PySDL2 Documentation

Release 0.9.5

Marcus von Appen

Jul 19, 2017

Contents

PySDL2 is a wrapper around the SDL2 library and as such similar to the discontinued PySDL project. In contrast to PySDL, it has no licensing restrictions, nor does it rely on C code, but uses `ctypes` instead.

CHAPTER 1

Contents

Installing PySDL2

This section provides an overview and guidance for installing PySDL2 on various target platforms.

Getting the sources

You can download the official releases of PySDL2 from https://bitbucket.org/marcusva/py-sdl2/downloads. Download the most recent release, unpack it and make sure that you installed the relevant prerequisites before continuing with the installation.

Prerequisites

PySDL2 relies on some 3rd party packages to be fully usable and to provide you full access to all of its features.

You must have at least one of the following Python versions installed:

- Python 2.7, 3.2+ (http://www.python.org)
- PyPy 1.8.0+ (http://www.pypy.org)

Other Python versions or Python implementations might work, but are (currently) not officially tested or supported by the PySDL2 distribution.

You need to have a working SDL2 library on your target system. You can obtain the source code (to build it yourself) or a prebuilt version at http://www.libsdl.org.

PySDL2 also offers support for the following SDL-related libraries:

- SDL2_image (http://www.libsdl.org/projects/SDL_image/)
- SDL2_mixer (http://www.libsdl.org/projects/SDL_mixer/)
- SDL2_ttf (http://www.libsdl.org/projects/SDL_ttf/)
- SDL2_gfx (http://www.ferzkopp.net/Software/SDL_gfx-2.0/)

Those are optional though and only necessary, if you want to use *sdl2.sdlimage*, *sdl2.sdlmixer*, *sdl2.sdlttf* or *sdl2.sdlgfx*.

Installation

You can either use the python way of installing the package or the make command using the Makefile on POSIX-compatible platforms, such as Linux or BSD, or the make.bat batch file on Windows platforms.

Simply type

```
python setup.py install
```

for the traditional python way or

```
make install
```

for using the Makefile or make.bat. Both will try to perform a default installation with as many features as possible.

Trying out

You also can test out PySDL2 without actually installing it. You just need to set up your `PYTHONPATH` to point to the location of the source distribution package. On Windows-based platforms, you might use something like

```
set PYTHONPATH=C:\path\to\pysdl2\:%PYTHONPATH%
```

to define the `PYTHONPATH` on a command shell. On Linux/Unix, use

```
export PYTHONPATH=/path/to/pysdl2:$PYTHONPATH
```

for bourne shell compatibles or

```
setenv PYTHONPATH /path/to/pysdl2:$PYTHONPATH
```

for C shell compatibles. You can omit the *:$PYTHONPATH*', if you did not use it so far and if your environment settings do not define it.

Note: If you are using IronPython, use `IRONPYTHONPATH` instead of `PYTHONPATH`.

Note: If you did not install SDL2 using the preferred way for your operation system, please read the information about *Importing* in the section *Integrating PySDL2*.

Notes on Mercurial usage

The Mercurial version of PySDL2 is not intended to be used in a production environment. Interfaces may change from one checkin to another, methods, classes or modules can be broken and so on. If you want more reliable code, please refer to the official releases.

Integrating PySDL2

PySDL2 consists of two packages, *sdl2*, which is a plain 1:1 API wrapper around the SDL2 API, and *sdl2.ext*, which offers enhanced functionality for *sdl2*.

The *sdl2* package is implemented in a way that shall make it easy for you to integrate and deploy it with your own software projects. You can rely on PySDL2 as third-party package, so that the user needs to install it before he can use your software. Alternatively, you can just copy the whole package into your project to ship it within your own project bundle.

Importing

The *sdl2* package relies on an external SDL2 library for creating the wrapper functions. This means that the user needs to have SDL2 installed or that you ship a SDL2 library with your project.

If the user has a SDL2 library installed on the target system, the `ctypes` hooks of *sdl2* try to find it in the OS-specific standard locations via `ctypes.util.find_library()`. If you are going to ship your own SDL2 library with the project or can not rely on the standard mechanism of `ctypes`, it is also possible to set the environment variable `PYSDL2_DLL_PATH`, which shall point to the directory of the SDL2 library or consist of a list of directories, in which the SDL2 libraries can be found.

Note: `PYSDL2_DLL_PATH` is preferred over the standard mechanism. If the module finds a SDL2 library in `PYSDL2_DLL_PATH`, it will try to use that one in the first place, before using any SDL2 library installed on the target system.

Let's assume, you ship your own library *SDL2.dll* within your project location *fancy_project/third_party*. You can set the environment variable `PYSDL2_DLL_PATH` before starting Python.

```
# Win32 platforms
set PYSDL2_DLL_PATH=C:\path\to\fancy_project\third_party

# Unix/Posix-alike environments - bourne shells
export PYSDL2_DLL_PATH=/path/to/fancy_project/third_party

# Unix/Posix-alike environments - C shells
setenv PYSDL2_DLL_PATH /path/to/fancy_project/third_party

# Define multiple paths to search for the libraries - Win32
set PYSDL2_DLL_PATH=C:\first\path;C:\second\path
```

You also can set the environment variable within Python using `os.environ`.

```
os.environ["PYSDL2_DLL_PATH"] = "C:\\path\\to\\fancy_project\\third_party"
os.environ["PYSDL2_DLL_PATH"] = "/path/to/fancy_project/third_party"
```

Note: If you aim to integrate `sdl` directly into your software and do not want or are not allowed to change the environment variables, you can also change the `os.getenv("PYSDL2_DLL_PATH")` query within the *sdl2/dll.py* (or *sdl2/sdlimage.py*, *sdl2/sdlttf.py*, *sdl2/sdlgfx.py*) file to point to the directory, in which you keep the DLL.

Using different SDL2 versions

PySDL2 tries to provide interfaces to the most recent versions of the SDL2 libraries. Sometimes this means that PySDL2 tries to test for functions that might not be available for your very own project or that are not available on the target system due to a version of the specific library. To check, if the SDL2 libraries do not provide certain functions, you can enable the specific warnings for them.

```
>>> python -W"module"::ImportWarning:sdl2.dll yourfile.py
```

Learn to fly - the tutorials

PySDL2 is easy to learn and a powerful multimedia programming framework. It features efficient high- and low-level structures and an excellent object-oriented programming layout.

The following tutorials will guide you through your first applications written with PySDL2 and introduces certain parts of the PySDL2 packages to you. They will most likely *not* cover each single part of PySDL2, but instead show you the most noteworthy features.

Hello World

Ahhh, the great tradition of saying "Hello World" in a programming language. To whet your appetite, we will do this with a most simple application, which will display an image. It is not important to understand everything at once, which will be used by the example. Nearly all parts used now are explained in later chapters, so do not hesitate, if the one or other explanation is missing.

Importing

Let's start with importing some basic modules, which are necessary to display a small nice window and to do some basic drawing within that window.

```
import sys
import sdl2.ext

RESOURCES = sdl2.ext.Resources(__file__, "resources")
```

We need some resources from the `resources` folder, so that we have a test image around to display on the window later on. In your own applications, it is unlikely that you will ever need to import them, but we need them here, so we use the *sdl2.ext.Resources* class to have them available.

Window creation and image loading

Any graphical application requires access to the screen, mostly in form of a window, which basically represents a portion of the screen, the application has access to and the application can manipulate. In most cases that portion has a border and title bar around it, allowing the user to move it around on the screen and reorganise everything in a way to fit his needs.

Once we have imported all necessary parts, let's create a window to have access to the screen, so we can display the logo and thus represent it to the user.

```
sdl2.ext.init()

window = sdl2.ext.Window("Hello World!", size=(640, 480))
window.show()

factory = sdl2.ext.SpriteFactory(sdl2.ext.SOFTWARE)
sprite = factory.from_image(RESOURCES.get_path("hello.bmp"))

spriterenderer = factory.create_sprite_render_system(window)
spriterenderer.render(sprite)
```

First, we initialise the *sdl2.ext* internals to gain access to the screen and to be able to create windows on top of it. Once done with that, *sdl2.ext.Window* will create the window for us and we supply a title to be shown on the window's border along with its initial size. Since *sdl2.ext.Window* instances are not shown by default, we have to tell the operating system and window manager that there is a new window to display by calling *sdl2.ext.Window.show()*.

Afterwards, we get an image from the resources folder and create a *sdl2.ext.Sprite* from it, which can be easily shown later on. This is done via a *sdl2.ext.SpriteFactory*, since the factory allows us to switch between texture-based, hardware-accelerated, and software-based sprites easily.

To display the image, we will use a *sdl2.ext.SpriteRenderSystem*, which supports the sprite type (texture- or software-based) and can copy the image to the window for display. The *sdl2.ext.SpriteRenderSystem* needs to know, where to copy to, thus we have to supply the window as target for copy and display operations.

All left to do is to initiate the copy process by calling *sdl2.ext.SpriteRenderSystem.render()* with the image we created earlier.

Tip: You will notice that the sprite used above will always be drawn at the top-left corner of the *sdl2.ext.Window*. You can change the position of where to draw it by changing its *sdl2.ext.Sprite.position* value.

```
# will cause the renderer to draw the sprite 10px to the right and
# 20 px to the bottom
sprite.position = 10, 20

# will cause the renderer to draw the sprite 55px to the right and
# 10 px to the bottom
sprite.position = 55, 10
```

Experiment with different values to see their effect. Do not forget to do this *before* `spriterenderer.render(sprite)` is called.

Making the application responsive

We are nearly done now. We have an image to display, we have a window, where the image should be displayed on, so we can execute the written code, not?

Well, yes, but the only thing that will happen is that we will notice a short flickering before the application exits. Maybe we can even see the window with the image for a short moment, but that's not what we want, do we?

To keep the window on the screen and to make it responsive to user input, such as closing the window, react upon the mouse cursor or key presses, we have to add a so-called event loop. The event loop will deal with certain types of actions happening on the window or while the window is focused by the user and - as long as the event loop is running - will keep the window shown on the screen.

```
processor = sdl2.ext.TestEventProcessor()
processor.run(window)
```

Since this is a very first tutorial, we keep things simple here and use a dummy class for testing without actually dealing with the event loop magic. By calling *sdl2.ext.TestEventProcessor.run()*, we implicitly start an event loop, which takes care of the most important parts for us.

And here it ends...

The window is shown, the image is shown, great! All left to do is to clean up everything, once the application finishes. Luckily the *sdl2.ext.TestEventProcessor* knows when the window is closed, so it will exit from the event loop. Once it exits, we should clean up the video internals, we initialised at the beginning. Thus, a final call to

```
sdl2.ext.quit()
```

should be made.

The Pong Game

The following tutorial will show you some capabilities of the component-based approach, PySDL2 features. We will create the basics of a simple Pong game implementation here. The basics of creating a event loop, dealing with user input, moving images around and creating a rendering function are covered in this tutorial.

Getting started

We start with creating the window and add a small event loop, so we are able to close the window and exit the game.

```
import sys
import sdl2
import sdl2.ext


def run():
    sdl2.ext.init()
    window = sdl2.ext.Window("The Pong Game", size=(800, 600))
    window.show()
    running = True
    while running:
        events = sdl2.ext.get_events()
        for event in events:
            if event.type == sdl2.SDL_QUIT:
                running = False
                break
        window.refresh()
    return 0

if __name__ == "__main__":
    sys.exit(run())
```

The import statements, video initialisation and window creation were discussed previously in the *Hello World* tutorial. We import everything from the *sdl2* package here, too, to have all SDL2 functions available.

Instead of some integrated event processor, a new code fragment is introduced, though.

```
running = True
while running:
    events = sdl2.ext.get_events()
    for event in events:
        if event.type == sdl2.SDL_QUIT:
            running = False
            break
    window.refresh()
```

The while loop above is the main event loop of our application. It deals with all kinds of input events that can occur when working with the window, such as mouse movements, key strokes, resizing operations and so on. SDL handles a lot for us when it comes to events, so all we need to do is to check, if there are any events, retrieve each event one by one, and handle it, if necessary. For now, we will just handle the `sdl2.SDL_QUIT` event, which is raised when the window is about to be closed.

In any other case we will just refresh the window's graphics buffer, so it is updated and visible on-screen.

Adding the game world

The window is available and working. Now let's take care of creating the game world, which will manage the player paddles, ball, visible elements and everything else. We are going to use an implementation layout loosely based on a COP[1] pattern, which separates data structures and functionality from each other. This allows us to change or enhance functional parts easily without having to refactor all classes we are implementing.

We start with creating the two player paddles and the rendering engine that will display them.

```
[...]

WHITE = sdl2.ext.Color(255, 255, 255)

class SoftwareRenderer(sdl2.ext.SoftwareSpriteRenderSystem):
    def __init__(self, window):
        super(SoftwareRenderer, self).__init__(window)

    def render(self, components):
        sdl2.ext.fill(self.surface, sdl2.ext.Color(0, 0, 0))
        super(SoftwareRenderer, self).render(components)


class Player(sdl2.ext.Entity):
    def __init__(self, world, sprite, posx=0, posy=0):
        self.sprite = sprite
        self.sprite.position = posx, posy


def run():
    ...

    world = sdl2.ext.World()

    spriterenderer = SoftwareRenderer(window)
    world.add_system(spriterenderer)

    factory = sdl2.ext.SpriteFactory(sdl2.ext.SOFTWARE)
    sp_paddle1 = factory.from_color(WHITE, size=(20, 100))
```

[1] Component-Oriented Programming

```
    sp_paddle2 = factory.from_color(WHITE, size=(20, 100))

    player1 = Player(world, sp_paddle1, 0, 250)
    player2 = Player(world, sp_paddle2, 780, 250)

    running = True
    while running:
        events = sdl2.ext.get_events()
        for event in events:
            if event.type == sdl2.SDL_QUIT:
                running = False
                break
        world.process()

if __name__ == "__main__":
    sys.exit(run())
```

The first thing to do is to enhance the *sdl2.ext.SoftwareSpriteRenderSystem* so that it will paint the whole window screen black on every drawing cycle, before drawing all sprites on the window.

Afterwards, the player paddles will be implemented, based on an *sdl2.ext.Entity* data container. The player paddles are simple rectangular sprites that can be positioned anywhere on the window.

In the main program function, we put those things together by creating a *sdl2.ext.World*, in which the player paddles and the renderer can live and operate.

Within the main event loop, we allow the world to process all attached systems, which causes it to invoke the `process()` methods for all *sdl2.ext.System* instances added to it.

Moving the ball

We have two static paddles centred vertically on the left and right of our window. The next thing to do is to add a ball that can move around within the window boundaries.

```
[...]
class MovementSystem(sdl2.ext.Applicator):
    def __init__(self, minx, miny, maxx, maxy):
        super(MovementSystem, self).__init__()
        self.componenttypes = Velocity, sdl2.ext.Sprite
        self.minx = minx
        self.miny = miny
        self.maxx = maxx
        self.maxy = maxy

    def process(self, world, componentsets):
        for velocity, sprite in componentsets:
            swidth, sheight = sprite.size
            sprite.x += velocity.vx
            sprite.y += velocity.vy

            sprite.x = max(self.minx, sprite.x)
            sprite.y = max(self.miny, sprite.y)

            pmaxx = sprite.x + swidth
            pmaxy = sprite.y + sheight
            if pmaxx > self.maxx:
                sprite.x = self.maxx - swidth
```

```
            if pmaxy > self.maxy:
                sprite.y = self.maxy - sheight


class Velocity(object):
    def __init__(self):
        super(Velocity, self).__init__()
        self.vx = 0
        self.vy = 0


class Player(sdl2.ext.Entity):
    def __init__(self, world, posx=0, posy=0):
        [...]
        self.velocity = Velocity()


class Ball(sdl2.ext.Entity):
    def __init__(self, world, sprite, posx=0, posy=0):
        self.sprite = sprite
        self.sprite.position = posx, posy
        self.velocity = Velocity()


def run():
    [...]
    sp_ball = factory.from_color(WHITE, size=(20, 20))
    [...]
    movement = MovementSystem(0, 0, 800, 600)
    spriterenderer = SoftwareRenderer(window)

    world.add_system(movement)
    world.add_system(spriterenderer)

    [...]

    ball = Ball(world, sp_ball, 390, 290)
    ball.velocity.vx = -3

    [...]
```

Two new classes are introduced here, `Velocity` and `MovementSystem`. The `Velocity` class is a simple data bag. It does not contain any application logic, but consists of the relevant information to represent the movement in a certain direction. This allows us to mark in-game items as being able to move around.

The `MovementSystem` in turn takes care of moving the in-game items around by applying the velocity to their current position. Thus, we can simply enable any `Player` instance to be movable or not by adding or removing a velocity attribute to them, which is a `Velocity` component instance.

Note: The naming is important here. The EBS implementation as described in *Working with component-based entities* requires every in-application or in-game item attribute bound to a *`sdl2.ext.Entity`* to be the lowercase class name of its related component.

```
Player.vel = Velocity(10, 10)
```

for example would raise an exception, since the system expects `Player.vel` to be an instance of a `Vel` component.

The MovementSystem is a specialised *sdl2.ext.System*, a *sdl2.ext.Applicator*, which can operate on combined sets of data. When the *sdl2.ext.Applicator.process()* method is called, the passed componentsets iterable will contain tuples of objects that belong to an instance and feature a certain type. The MovementSystem's process() implementation hence will loop over sets of Velocity and Sprite instances that belong to the same *sdl2.ext.Entity*. Since we have a ball and two players currently available, it typically would loop over three tuples, two for the individual players and one for the ball.

The *sdl2.ext.Applicator* thus enables us to process combined data of our in-game items, without creating complex data structures.

Note: Only entities that contain *all* attributes (components) are taken into account. If e.g. the Ball class would not contain a Velocity component, it would not be processed by the MovementSystem.

Why do we use this approach? The *sdl2.ext.Sprite* objects carry a position, which defines the location at which they should be rendered, when processed by the SoftwareRenderer. If they should move around (which is a change in the position), we need to apply the velocity to them.

We also define some more things within the MovementSystem, such as a simple boundary check, so that the players and ball cannot leave the visible window area on moving around.

Bouncing

We have a ball that can move around as well as the general game logic for moving things around. In contrast to a classic OO approach we do not need to implement the movement logic within the Ball and Player class individually, since the basic movement is the same for all (yes, you could have solved that with inheriting Ball and Player from a MovableObject class in OO).

The ball now moves and stays within the bounds, but once it hits the left side, it will stay there. To make it *bouncy*, we need to add a simple collision system, which causes the ball to change its direction on colliding with the walls or the player paddles.

```
[...]
class CollisionSystem(sdl2.ext.Applicator):
    def __init__(self, minx, miny, maxx, maxy):
        super(CollisionSystem, self).__init__()
        self.componenttypes = Velocity, sdl2.ext.Sprite
        self.ball = None
        self.minx = minx
        self.miny = miny
        self.maxx = maxx
        self.maxy = maxy

    def _overlap(self, item):
        pos, sprite = item
        if sprite == self.ball.sprite:
            return False

        left, top, right, bottom = sprite.area
        bleft, btop, bright, bbottom = self.ball.sprite.area

        return (bleft < right and bright > left and
                btop < bottom and bbottom > top)

    def process(self, world, componentsets):
        collitems = [comp for comp in componentsets if self._overlap(comp)]
        if collitems:
```

```
            self.ball.velocity.vx = -self.ball.velocity.vx


def run():
    [...]
    world = World()

    movement = MovementSystem(0, 0, 800, 600)
    collision = CollisionSystem(0, 0, 800, 600)
    spriterenderer = SoftwareRenderer(window)

    world.add_system(movement)
    world.add_system(collision)
    world.add_system(spriterenderer)

    [...]
    collision.ball = ball

    running = True
    while running:
        events = sdl2.ext.get_events()
        for event in events:
            if event.type == sdl2.SDL_QUIT:
                running = False
                break
        sdl2.SDL_Delay(10)
        world.process()

if __name__ == "__main__":
    sys.exit(run())
```

The `CollisionSystem` only needs to take care of the ball and objects it collides with, since the ball is the only unpredictable object within our game world. The player paddles will only be able to move up and down within the visible window area and we already dealt with that within the `MovementSystem` code.

Whenever the ball collides with one of the paddles, its movement direction (velocity) should be inverted, so that it *bounces* back.

Additionally, we won't run at the full processor speed anymore in the main loop, but instead add a short delay, using the `sdl2.SDL_Delay()` function. This reduces the overall load on the CPU and makes the game a bit slower.

Reacting on player input

We have a moving ball that bounces from side to side. The next step would be to allow moving one of the paddles around, if the player presses a key. The SDL event routines allow us to deal with a huge variety of user and system events that could occur for our application, but right now we are only interested in key strokes for the Up and Down keys to move one of the player paddles up or down.

```
[...]
def run():
    [...]
    running = True
    while running:
        events = sdl2.ext.get_events()
        for event in events:
            if event.type == sdl2.SDL_QUIT:
                running = False
```

```
                break
            if event.type == sdl2.SDL_KEYDOWN:
                if event.key.keysym.sym == sdl2.SDLK_UP:
                    player1.velocity.vy = -3
                elif event.key.keysym.sym == sdl2.SDLK_DOWN:
                    player1.velocity.vy = 3
            elif event.type == sdl2.SDL_KEYUP:
                if event.key.keysym.sym in (sdl2.SDLK_UP, sdl2.SDLK_DOWN):
                    player1.velocity.vy = 0
        sdl2.SDL_Delay(10)
        world.process()

if __name__ == "__main__":
    sys.exit(run())
```

Every event that can occur and that is supported by SDL2 can be identified by a static event type code. This allows us to check for a key stroke, mouse button press, and so on. First, we have to check for `sdl2.SDL_KEYDOWN` and `sdl2.SDL_KEYUP` events, so we can start and stop the paddle movement on demand. Once we identified such events, we need to check, whether the pressed or released key is actually the Up or Down key, so that we do not start or stop moving the paddle, if the user presses R or G or whatever.

Whenever the Up or Down key are pressed down, we allow the left player paddle to move by changing its velocity information for the vertical direction. Likewise, if either of those keys is released, we stop moving the paddle.

Improved bouncing

We have a moving paddle and we have a ball that bounces from one side to another, which makes the game ... quite boring. If you played Pong before, you know that most variations of it will cause the ball to bounce in a certain angle, if it collides with a paddle. Most of those implementations achieve this by implementing the paddle collision as if the ball collides with a rounded surface. If it collides with the center of the paddle, it will bounce back straight, if it hits the paddle near the center, it will bounce back with a pointed angle and on the corners of the paddle it will bounce back with some angle close to 90 degrees to its initial movement direction.

```
class CollisionSystem(sdl2.ext.Applicator):
    [...]

    def process(self, world, componentsets):
        collitems = [comp for comp in componentsets if self._overlap(comp)]
        if collitems:
            self.ball.velocity.vx = -self.ball.velocity.vx

            sprite = collitems[0][1]
            ballcentery = self.ball.sprite.y + self.ball.sprite.size[1] // 2
            halfheight = sprite.size[1] // 2
            stepsize = halfheight // 10
            degrees = 0.7
            paddlecentery = sprite.y + halfheight
            if ballcentery < paddlecentery:
                factor = (paddlecentery - ballcentery) // stepsize
                self.ball.velocity.vy = -int(round(factor * degrees))
            elif ballcentery > paddlecentery:
                factor = (ballcentery - paddlecentery) // stepsize
                self.ball.velocity.vy = int(round(factor * degrees))
            else:
                self.ball.velocity.vy = - self.ball.velocity.vy
```

The reworked processing code above simulates a curved paddle by creating segmented areas, which cause the ball to be reflected in different angles. Instead of doing some complex trigonometry to calculate an accurate angle and transform it on a x/y plane, we simply check, where the ball collided with the paddle and adjust the vertical velocity.

If the ball now hits a paddle, it can be reflected at different angles, hitting the top and bottom window boundaries... and will stay there. If it hits the window boundaries, it should be reflected, too, but not with a varying angle, but with the exact angle, it hit the boundary with. This means that we just need to invert the vertical velocity, once the ball hits the top or bottom.

```
class CollisionSystem(sdl2.ext.Applicator):
    [...]

    def process(self, world, componentsets):
        [...]

        if (self.ball.sprite.y <= self.miny or
            self.ball.sprite.y + self.ball.sprite.size[1] >= self.maxy):
            self.ball.velocity.vy = - self.ball.velocity.vy

        if (self.ball.sprite.x <= self.minx or
            self.ball.sprite.x + self.ball.sprite.size[0] >= self.maxx):
            self.ball.velocity.vx = - self.ball.velocity.vx
```

Creating an enemy

Now that we can shoot back the ball in different ways, it would be nice to have an opponent to play against. We could enhance the main event loop to recognise two different keys and manipulate the second paddle's velocity for two people playing against each other. We also could create a simple computer-controlled player that tries to hit the ball back to us, which sounds more interesting.

```
class TrackingAIController(sdl2.ext.Applicator):
    def __init__(self, miny, maxy):
        super(TrackingAIController, self).__init__()
        self.componenttypes = PlayerData, Velocity, sdl2.ext.Sprite
        self.miny = miny
        self.maxy = maxy
        self.ball = None

    def process(self, world, componentsets):
        for pdata, vel, sprite in componentsets:
            if not pdata.ai:
                continue

            centery = sprite.y + sprite.size[1] // 2
            if self.ball.velocity.vx < 0:
                # ball is moving away from the AI
                if centery < self.maxy // 2:
                    vel.vy = 3
                elif centery > self.maxy // 2:
                    vel.vy = -3
                else:
                    vel.vy = 0
            else:
                bcentery = self.ball.sprite.y + self.ball.sprite.size[1] // 2
                if bcentery < centery:
                    vel.vy = -3
                elif bcentery > centery:
```

```
                    vel.vy = 3
                else:
                    vel.vy = 0


class PlayerData(object):
    def __init__(self):
        super(PlayerData, self).__init__()
        self.ai = False


class Player(sdl2.ext.Entity):
    def __init__(self, world, sprite, posx=0, posy=0, ai=False):
        self.sprite = sprite
        self.sprite.position = posx, posy
        self.velocity = Velocity()
        self.playerdata = PlayerData()
        self.playerdata.ai = ai


def run():
    [...]
    aicontroller = TrackingAIController(0, 600)

    world.add_system(aicontroller)
    world.add_system(movement)
    world.add_system(collision)
    world.add_system(spriterenderer)

    player1 = Player(world, sp_paddle1, 0, 250)
    player2 = Player(world, sp_paddle2, 780, 250, True)
    [...]
    aicontroller.ball = ball

    [...]
```

We start by creating a component `PlayerData` that flags a player as being AI controlled or not. Afterwards, a `TrackingAIController` is implemented, which, depending on the information of the `PlayerData` component, will move the specific player paddle around by manipulating its velocity information.

The AI is pretty simple, just following the ball's vertical movement, trying to hit it at its center, if the ball moves into the direction of the AI-controlled paddle. As soon as the ball moves away from the paddle, the paddle will move back to the vertical center.

Tip: Add `True` as last parameter to the first `Player()` constructor to see two AIs playing against each other.

Next steps

We created the basics of a Pong game, which can be found in the examples folder. However, there are some more things to do, such as

- resetting the ball to the center with a random vertical velocity, if it hits either the left or right window bounds
- adding the ability to track the points made by either player, if the ball hit the left or right side
- drawing a dashed line in the middle to make the game field look nicer

- displaying the points made by each player

It is your turn now to implement these features. Go ahead, it is not as complex as it sounds.

- you can reset the ball's position in the `CollisionSystem` code, by changing the code for the `minx` and `maxx` test
- you could enhance the `CollisionSystem` to process `PlayerData` components and add the functionality to add points there (or write a small processor that keeps track of the ball only and processes only the `PlayerData` and `video.SoftSprite` objects of each player for adding points). Alternatively, you could use the *`sdl2.ext.EventHandler`* class to raise a score count function within the `CollisionSystem`, if the ball collides with one of the paddles.
- write an own render sytem, based on *`sdl2.ext.Applicator`*, which takes care of position and sprite sets

```
StaticRepeatingSprite(Entity):
    ...
    self.positions = Positions((400, 0), (400, 60), (400, 120), ...)
    ...
```

- draw some simple images for 0-9 and render them as sprites, depending on the points a player made.

PySDL2 for Pygamers

Care to move to a newer SDL with your Pygame knowledge? Then you should know one thing or two about PySDL2 before hacking code, since it is completely different from Pygame. Do not let that fact scare you away, the basics with graphics and sound are still the same (as they are fundamental), but you will not find many similarities to the Pygame API within PySDL2.

Todo

More details, examples, etc.

Technical differences

Pygame is implemented as a mixture of Python, C and Assembler code, wrapping 3rd party libraries with CPython API interfaces. PySDL2 in contrast is written in pure Python, using `ctypes` to interface with the C interfaces of 3rd party libraries.

API differences

pygame

pygame	sdl2
`init()`	`sdl2.SDL_Init()` where appropriate
`quit()`	`sdl2.SDL_Quit()` where appropriate
`error`	No equivalent
`get_error()`	`sdl2.SDL_GetError()`
`set_error()`	`sdl2.SDL_SetError()`
`get_sdl_version()`	`sdl2.SDL_GetVersion()`
`get_sdl_byteorder()`	`sdl2.SDL_BYTEORDER`
`register_quit()`	No equivalent planned
`encode_string()`	No equivalent planned
`encode_file_path()`	No equivalent planned

pygame.cdrom

PySDL2 does not feature any CD-ROM related interfaces. They were removed in SDL2 and PySDL2 does not provide its own facilities.

pygame.Color

You can find a similar class in *sdl2.ext.Color*. It does not feature a `set_length()` or `correct_gamma()` method, though.

pygame.cursors

PySDL2 does not feature any pre-defined cursor settings at the moment.

pygame.display

pygame.display	sdl2
`init()`	*sdl2.ext.init()*
`quit()`	*sdl2.ext.quit()*
`get_init()`	`sdl2.SDL_WasInit()`
`set_mode()`	*sdl2.ext.Window*
`get_surface()`	*sdl2.ext.Window.get_surface()*
`flip()`	*sdl2.ext.Window.refresh()*
`update()`	*sdl2.ext.Window.refresh()*
`get_driver()`	`sdl2.SDL_GetCurrentVideoDriver()`
`Info`	No equivalent
`get_wm_info()`	`sdl2.SDL_GetWindowWMInfo()`
`list_modes()`	`sdl2.SDL_GetNumDisplayModes()`
`mode_ok()`	`sdl2.SDL_GetClosestDisplayMode()`
`gl_get_attribute()`	`sdl2.SDL_GL_GetAttribute()`
`gl_set_attribute()`	`sdl2.SDL_GL_SetAttribute()`
`get_active()`	No equivalent
`iconify()`	*sdl2.ext.Window.minimize()*
`toggle_fullscreen()`	`sdl2.SDL_SetWindowFullscreen()`
`set_gamma()`	`sdl2.SDL_SetWindowBrightness()`
`set_gamma_ramp()`	`sdl2.SDL_SetWindowGammaRamp.()`
`set_icon()`	`sdl2.SDL_SetWindowIcon()`
`set_caption()`	*sdl2.ext.Window.title*
`get_caption()`	*sdl2.ext.Window.title*
`set_palette()`	`sdl2.SDL_SetSurfacePalette()`

pygame.draw

Drawing primitives can be accessed through either the `sdl2.SDL_RenderDraw*()` and `sdl2.SDL_RenderFill*()` functions or the more powerful *sdl2.sdlgfx* module,

pygame.event

pygame.event	sdl2
`pump()`	`sdl2.SDL_PumpEvents()`
`get()`	`sdl2.SDL_PollEvent()` or *sdl2.ext.get_events()*
`poll()`	`sdl2.SDL_PollEvent()`
`wait()`	`sdl2.SDL_WaitEvent()`
`peek()`	`sdl2.SDL_PeepEvents()`
`clear()`	`sdl2.SDL_FlushEvents()`
`event_name()`	No equivalent
`set_blocked()`	`sdl2.SDL_EventState()`
`get_blocked()`	`sdl2.SDL_EventState()`
`set_allowed()`	`sdl2.SDL_EventState()`
`set_grab()`	`sdl2.SDL_SetWindowGrab()`
`get_grab()`	`sdl2.SDL_GetWindowGrab()`
`post()`	`sdl2.SDL_PeepEvents()`
`Event`	`sdl2.SDL_Event`

pygame.font

pygame.font	sdl2
`init()`	`sdl2.sdlttf.TTF_Init()`
`quit()`	`sdl2.sdlttf.TTF_Quit()`
`get_init()`	`sdl2.sdlttf.TTF_WasInit()`
`get_default_font()`	No equivalent planned[1]
`get_fonts()`	No equivalent planned[1]
`match_font()`	No equivalent planned[1]
`SysFont`	No equivalent planned[1]
`Font`	No equivalent planned[1]

pygame.freetype

PySDL2 does not feature direct FreeType support.

pygame.gfxdraw

PySDL2 offers SDL_gfx support through the *`sdl2.sdlgfx`* module.

pygame.image

pygame.image	sdl2
`load()`	`sdl2.sdlimage.IMG_Load()`, *`sdl2.ext.load_image()`*
`save()`	`sdl2.surface.SDL_SaveBMP()`, `sdl2.sdlimage.IMG_SavePNG()`
`get_extended()`	`sdl2.sdlimage.IMG_isBMP()` et al.
`tostring()`	No equivalent yet
`fromstring()`	No equivalent yet
`frombuffer()`	No equivalent yet

pygame.joystick

pygame.joystick	sdl2
`init()`	`sdl2.SDL_Init()`
`quit()`	`sdl2.SDL_Quit()`
`get_init()`	`sdl2.SDL_WasInit()`
`get_count()`	`sdl2.joystick.SDL_NumJoysticks()`
`Joystick()`	`sdl2.joystick.SDL_Joystick` and related functions

[1] Check https://bitbucket.org/marcusva/python-utils for an easy to use system font detection module

pygame.key

pygame.key	sdl2
`get_focused()`	`sdl2.keyboard.SDL_GetKeyboardFocus()`
`get_pressed()`	`sdl2.keyboard.SDL_GetKeyboardState()`
`get_mods()`	`sdl2.keyboard.SDL_GetModState()`
`set_mods()`	`sdl2.keyboard.SDL_SetModState()`
`set_repeat()`	Based on the OS/WM settings, no equivalent
`get_repeat()`	Based on the OS/WM settings, no equivalent
`name()`	`sdl2.keyboard.SDL_GetKeyName()`

pygame.locals

Constants in PySDL2 are spread across the different packages and modules, depending on where they originate from.

pygame.mixer

pygame.mixer	sdl2
`init()`	`sdl2.sdlmixer.Mix_Init()`
`quit()`	`sdl2.sdlmixer.Mix_Quit()`
`get_init()`	No equivalent planned
`stop()`	`sdl2.sdlmixer.Mix_HaltChannel()`, `sdl2.sdlmixer.Mix_HaltGroup()`, `sdl2.sdlmixer.Mix_HaltMusic()`
`pause()`	`sdl2.sdlmixer.Mix_Pause()`, `sdl2.sdlmixer.Mix_PauseMusic()`
`unpause()`	`sdl2.sdlmixer.Mix_Resume()`, `sdl2.sdlmixer.Mix_ResumeMusic()`
`fadeout()`	`sdl2.sdlmixer.Mix_FadeOutChannel()`, `sdl2.sdlmixer.Mix_FadeOutGroup()`, `sdl2.sdlmixer.Mix_FadeOutMusic()`
`set_num_channels()`	`sdl2.sdlmixer.Mix_AllocateChannels()`
`get_num_channels()`	`sdl2.sdlmixer.Mix_AllocateChannels()`
`set_reserved()`	`sdl2.sdlmixer.Mix_ReserveChannels()`
`find_channel()`	No equivalent planned
`get_busy()`	`sdl2.sdlmixer.Mix_ChannelFinished()`
`Sound`	`sdl2.sdlmixer.Mix_Chunk`
`Channel`	No equivalent, use the channel functions instead

pygame.mixer.music

See *pygame.mixer*.

pygame.mouse

pygame.mouse	sdl2
`get_pressed()`	`sdl2.mouse.SDL_GetMouseState()`
`get_pos()`	`sdl2.mouse.SDL_GetMouseState()`
`get_rel()`	`sdl2.mouse.SDL_GetRelativeMouseState()`
`set_pos()`	`sdl2.mouse.SDL_WarpMouseInWindow()`
`set_visible()`	`sdl2.mouse.SDL_ShowCursor()`
`get_focused()`	`sdl2.mouse.SDL_GetMouseFocus()`
`set_cursor()`	`sdl2.mouse.SDL_GetCursor()`
`get_cursor()`	`sdl2.mouse.SDL_SetCursor()`

pygame.movie

No such module is planned for PySDL2.

pygame.Overlay

You can work with YUV overlays by using the `sdl2.render` module with `sdl2.render.SDL_Texture` objects.

pygame.PixelArray

You can access pixel data of sprites and surfaces directly via the *`sdl2.ext.PixelView`* class. It does not feature comparison or extractions methods.

pygame.Rect

No such functionality is available for PySDL2. Rectangles are represented via `sdl2.rect.SDL_Rect` for low-level SDL2 wrappers or 4-value tuples.

pygame.scrap

PySDL2 offers basic text-based clipboard access via the `sdl2.clipboard` module. A feature-rich clipboard API as for Pygame does not exist yet.

pygame.sndarray

No such module is available for PySDL2 yet.

pygame.sprite

PySDL2 uses a different approach of rendering and managing sprite objects via a component-based system and the *`sdl2.ext.Sprite`* class. A sprite module as for Pygame is not planned.

pygame.Surface

pygame.Surface	sdl2
blit()	sdl2.surface.SDL_BlitSurface(), *sdl2.ext.SpriteRenderSystem*
convert()	sdl2.surface.SDL_ConvertSurface()
convert_alpha()	sdl2.surface.SDL_ConvertSurface()
copy()	sdl2.surface.SDL_ConvertSurface()
fill()	sdl2.surface.SDL_FillRect(), sdl2.surface.SDL_FillRects(), *sdl2.ext.fi*
scroll()	No equivalent planned
set_colorkey()	sdl2.surface.SDL_SetColorKey()
get_colorkey()	sdl2.surface.SDL_GetColorKey()
set_alpha()	sdl2.surface.SDL_SetSurfaceAlphaMod()
get_alpha()	sdl2.surface.SDL_GetSurfaceAlphaMod()
lock()	sdl2.surface.SDL_LockSurface()
unlock()	sdl2.surface.SDL_UnlockSurface()
mustlock()	sdl2.surface.SDL_MUSTLOCK()
get_locked()	sdl2.surface.SDL_Surface.locked
get_locks()	No equivalent planned
get_at()	Direct access to the pixels for surfaces can be achieved via the *sdl2.ext.PixelView* class
set_at()	Direct access to the pixels for surfaces can be achieved via the *sdl2.ext.PixelView* class
get_at_mapped()	No equivalent planned
get_palette()	via sdl2.surface.SDL_Surface.format and the sdl2.pixels.SDL_PixelFormat.p
get_palette_at()	sdl2.pixels.SDL_Palette.colors[offset]
set_palette()	sdl2.surface.SDL_SetSurfacePalette()
set_palette_at()	sdl2.pixels.SDL_Palette.colors[offset]
map_rgb()	sdl2.pixels.SDL_MapRGB()
unmap_rgb()	sdl2.pixels.SDL_GetRGB()
set_clip()	sdl2.surface.SDL_SetClipRect()
get_clip()	sdl2.surface.SDL_GetClipRect()
subsurface()	*sdl2.ext.subsurface()*
get_parent()	No equivalent yet
get_abs_parent()	As for get_parent
get_offset()	As for get_parent
get_abs_offset()	As for get_parent
get_size()	*sdl2.ext.Sprite.size*, sdl2.surface.SDL_Surface.w, sdl2.surface.SDL_Surf
get_width()	sdl2.ext.Sprite.size[0], sdl2.surface.SDL_Surface.w,
get_height()	sdl2.ext.Sprite.size[1], sdl2.surface.SDL_Surface.h
get_rect()	No equivalent planned
get_bitsize()	sdl2.pixels.SDL_PixelFormat.BitsPerPixel
get_bytesize()	sdl2.pixels.SDL_PixelFormat.BytesPerPixel
get_flags()	sdl2.surface.SDL_Surface.flags
get_pitch()	sdl2.surface.SDL_Surface.pitch
get_masks()	sdl2.pixels.SDL_PixelFormat.Rmask, ...
get_shifts()	sdl2.pixels.SDL_PixelFormat.Rshift, ...
get_losses()	sdl2.pixels.SDL_PixelFormat.Rloss, ...
get_bounding_rect()	No equivalent planned
get_view()	*sdl2.ext.PixelView*
get_buffer()	*sdl2.ext.PixelView* or sdl2.surface.SDL_Surface.pixels

pygame.surfarray

2D and 3D pixel access can be achieved via the *sdl2.ext.PixelView* class in environments without numpy. Simplified numpy-array creation with direct pixel access (similar to `pygame.surfarray.pixels2d()` and `pygame.surfarray.pixels3d()`) is available via *sdl2.ext.pixels2d()* and *sdl2.ext.pixels3d()*.

pygame.time

pygame.time	sdl2
`get_ticks()`	`sdl2.timer.SDL_GetTicks()`
`wait()`	`sdl2.timer.SDL_Delay()`
`delay()`	`sdl2.timer.SDL_Delay()`
`Clock`	No equivalent planned

pygame.transform

The are no transformation helpers in PySDL2 at moment. Those might be implemented later on via numpy helpers, the Python Imaging Library or other 3rd party packages.

pygame.version

pygame.version	sdl2
`ver`	`sdl2.__version__`
`vernum`	`sdl2.version_info`

API reference

This is the core documentation of the various modules, classes and functions PySDL2 offers. If you want to have a quick overview about the modules, use the modindex. If you just want to look up a specific class, method or function, use the genindex or search.

sdl2 - SDL2 library wrapper

The *sdl2* package is a `ctypes`-based wrapper around the SDL2 library. It wraps nearly all publicly accessible structures and functions of the SDL2 library to be accessible from Python code.

A detailed documentation about the behaviour of the different functions can found within the SDL2 documentation.

Usage

You can use *sdl2* in nearly exactly the same way as you would do with the SDL library and C code.

A brief example in C code:

```c
#include <SDL.h>

int main(int argc, char *argv[]) {
    int running;
    SDL_Window *window;
    SDL_Surface *windowsurface;
    SDL_Surface *image;
    SDL_Event event;

    SDL_Init(SDL_INIT_VIDEO);

    window = SDL_CreateWindow("Hello World",
                              SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED,
                              592, 460, SDL_WINDOW_SHOWN);
    windowsurface = SDL_GetWindowSurface(window);

    image = SDL_LoadBMP("exampleimage.bmp");
    SDL_BlitSurface(image, NULL, windowsurface, NULL);

    SDL_UpdateWindowSurface(window);
    SDL_FreeSurface(image);

    running = 1;
    while (running) {
        while (SDL_PollEvent(&event) != 0) {
            if (event.type == SDL_QUIT) {
                running = 0;
                break;
            }
        }
    }
    SDL_DestroyWindow(window);
    SDL_Quit();
    return 0;
}
```

Doing the same in Python:

```python
import sys
import ctypes
from sdl2 import *

def main():
    SDL_Init(SDL_INIT_VIDEO)
    window = SDL_CreateWindow(b"Hello World",
                              SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED,
                              592, 460, SDL_WINDOW_SHOWN)
    windowsurface = SDL_GetWindowSurface(window)

    image = SDL_LoadBMP(b"exampleimage.bmp")
    SDL_BlitSurface(image, None, windowsurface, None)

    SDL_UpdateWindowSurface(window)
    SDL_FreeSurface(image)

    running = True
    event = SDL_Event()
    while running:
```

```
        while SDL_PollEvent(ctypes.byref(event)) != 0:
            if event.type == SDL_QUIT:
                running = False
                break

    SDL_DestroyWindow(window)
    SDL_Quit()
    return 0

if __name__ == "__main__":
    sys.exit(main())
```

You can port code in a straightforward manner from one language to the other, though it is important to know about the limitations and slight differences mentioned below. Also, PySDL2 offers advanced functionality, which also feels more *'pythonic'*, via the *sdl2.ext* package.

Missing interfaces

The following functions, classes, constants and macros of SDL2 are *not* available within *sdl2*.

- SDL_REVISION and SDL_REVISION_NUMBER from SDL_revision.h
- SDL_NAME() from SDL_name.h
- SDL_MostSignificantBitIndex32() from SDL_bits.h
- Everything from SDL_main.h
- Everything from SDL_system.h
- Everything from SDL_assert.h
- Everything from SDL_thread.h
- Everything from SDL_atomic.h
- Everything from SDL_opengl.h
- Everything from SDL_mutex.h

Additional interfaces

The following functions, classes, constants and macros are are *not* part of SDL2, but were introduced by *sdl2*.

sdl2.**ALL_PIXELFORMATS**
: Tuple containing all SDL2 pixel format constants (SDL_PIXELFORMAT_INDEX1LSB, ..., SDL_PIXELFORMAT_RGB565, ...).

sdl2.**AUDIO_FORMATS**
: Set containing all SDL2 audio format constants (AUDIO_U8, AUDIO_S8, ... AUDIO_F32LSB, ...).

sdl2.**rw_from_object**(*obj : object*) → SDL_RWops
: Creates a SDL_RWops from any Python object. The Python object must at least support the following methods:

  read(length) -> data

  > length is the size in bytes to be read. A call to len(data) must return the correct amount of bytes for the data, so that len(data) / [size in bytes for a single element from data] returns the amount of elements. Must raise an error on failure.

  seek(offset, whence) -> int

offset denotes the offset to move the read/write pointer of the object to. whence indicates the movement behaviour and can be one of the following values:

- RW_SEEK_SET - move to offset from the start of the file
- RW_SEEK_CUR - move by offset from the relative location
- RW_SEEK_END - move to offset from the end of the file

If it could not move read/write pointer to the desired location, an error must be raised.

tell() -> int

Must return the current offset. This method must only be provided, if seek() does not return any value.

close() -> None

Closes the object(or its internal data access methods). Must raise an error on failure.

write(data) -> None

Writes the passed data(which is a string of bytes) to the object. Must raise an error on failure.

Note: The write() method is optional and only necessary, if the passed object should be able to write data.

The returned `sdl2.SDL_RWops` is a pure Python object and **must not** be freed via `sdl2.SDL_FreeRW()`.

sdl2.sdlgfx - SDL2_gfx library wrapper

The *`sdl2.sdlgfx`* module is a `ctypes`-based wrapper around the SDL2_gfx library. It wraps nearly all publicly accessible structures and functions of the SDL2_gfx library to be accessible from Python code.

A detailed documentation about the behaviour of the different functions can found on the SDL2_gfx project website.

sdl2.sdlimage - SDL2_image library wrapper

The *`sdl2.sdlimage`* module is a `ctypes`-based wrapper around the SDL2_image library. It wraps nearly all publicly accessible structures and functions of the SDL2_image library to be accessible from Python code.

A detailed documentation about the behaviour of the different functions can found within the SDL2_image documentation.

sd2.sdlmixer - SDL2_mixer library wrapper

The *`sdl2.sdlmixer`* module is a `ctypes`-based wrapper around the SDL2_mixer library. It wraps nearly all publicly accessible structures and functions of the SDL2_mixer library to be accessible from Python code.

A detailed documentation about the behaviour of the different functions can found within the SDL2_mixer documentation.

sd2.sdlttf - SDL2_ttf library wrapper

The *sdl2.sdlttf* module is a ctypes-based wrapper around the SDL2_ttf library. It wraps nearly all publicly accessible structures and functions of the SDL2_ttf library to be accessible from Python code.

A detailed documentation about the behaviour of the different functions can found within the SDL2_ttf documentation.

sdl2.ext - Python extensions for SDL2

The *sdl2.ext* package provides advanced functionality for creating applications using SDL2 and Python. It offers a rich set of modules, classes and functions, such as easy image loading, basic user interface elements, resource management and sprite and (on-screen) scene systems.

Learn more about

Common algorithms

sdl2.ext.**cohensutherland**(*left : int*, *top : int*, *right : int*, *bottom : int*, *x1 : int*, *y1 : int*, *x2 : int*, *y2 : int*) → int, int, int, int

This implements the Cohen-Sutherland line clipping algorithm. *left*, *top*, *right* and *bottom* denote the clipping area, into which the line defined by *x1*, *y1* (start point) and *x2*, *y2* (end point) will be clipped.

If the line does not intersect with the rectangular clipping area, four `None` values will be returned as tuple. Otherwise a tuple of the clipped line points will be returned in the form `(cx1, cy1, cx2, cy2)`.

sdl2.ext.**liangbarsky**(*left : int*, *top : int*, *right : int*, *bottom : int*, *x1 : int*, *y1 : int*, *x2 : int*, *y2 : int*) → int, int, int, int

This implements the Liang-Barsky line clipping algorithm. *left*, *top*, *right* and *bottom* denote the clipping area, into which the line defined by *x1*, *y1* (start point) and *x2*, *y2* (end point) will be clipped.

If the line does not intersect with the rectangular clipping area, four `None` values will be returned as tuple. Otherwise a tuple of the clipped line points will be returned in the form `(cx1, cy1, cx2, cy2)`.

sdl2.ext.**clipline**(*left : int*, *top : int*, *right : int*, *bottom : int*, *x1 : int*, *y1 : int*, *x2 : int*, *y2 : int*[, *method=liangbarsky*]) → int, int, int, int

Clips a line to a rectangular area.

sdl2.ext.**point_on_line**(*p1 : iterable*, *p2 : iterable*, *point : iterable*) → bool

Checks, if *point*, a two-value tuple, is on the line segment defined by *p1* and *p2*.

Converting sequences

This module provides various functions and classes to access sequences and buffer-style objects in different ways. It also provides conversion routines to improve the interoperability of sequences with ctypes data types.

Providing read-write access for sequential data

Two classes allow you to access sequential data in different ways. The *CTypesView* provides byte-wise access to iterable objects and allows you to convert the object representation to matching byte-widths for ctypes or other modules.

Depending on the the underlying object and the chosen size of each particular item of the object, the *CTypesView* allows you to operate directly on different representations of the object's contents.

```
>>> text = bytearray("Hello, I am a simple ASCII string!")
>>> ctview = CTypesView(text, itemsize=1)
>>> ctview.view[0] = 0x61
>>> print(text)
aello, I am a simple ASCII string!"
>>> ctview.to_uint16()[3] = 0x6554
>>> print(text)
aello,Te am a simple ASCII string!"
```

The snippet above provides a single-byte sized view on a `bytearray()` object. Afterwards, the first item of the view is changed, which causes a change on the `bytearray()`, on the first item as well, since both, the *CTypesView* and the `bytearray()` provide a byte-wise access to the contents.

By using *CTypesView.to_uint16()*, we change the access representation to a 2-byte unsigned integer `ctypes` pointer and change the fourth 2-byte value, *I* to something else.

```
>>> text = bytearray("Hello, I am a simple ASCII string!")
>>> ctview = CTypesView(text, itemsize=2)
>>> ctview.view[0] = 0x61
>>> print(text)
aello, I am a simple ASCII string!"
>>> ctview.to_uint16()[3] = 0x6554
>>> print(text)    aello,Te am a simple ASCII string!"
```

If the encapsuled object does not provide a (writable) `buffer()` interface, but is iterable, the *CTypesView* will create an internal copy of the object data using Python's `array` module and perform all operations on that copy.

```
>>> mylist = [18, 52, 86, 120, 154, 188, 222, 240]
>>> ctview = CTypesView(mylist, itemsize=1, docopy=True)
>>> print(ctview.object)
array('B', [18, 52, 86, 120, 154, 188, 222, 240])
>>> ctview.view[3] = 0xFF
>>> print(mylist)
[18, 52, 86, 120, 154, 188, 222, 240]
>>> print(ctview.object)
array('B', [18, 52, 86, 255, 154, 188, 222, 240])
```

As for directly accessible objects, you can define your own itemsize to be used. If the iterable does not provide a direct byte access to their contents, this won't have any effect except for resizing the item widths.

```
>>> mylist = [18, 52, 86, 120, 154, 188, 222, 240]
>>> ctview = CTypesView(mylist, itemsize=4, docopy=True)
>>> print(ctview.object)
array('I', [18L, 52L, 86L, 120L, 154L, 188L, 222L, 240L])
```

Accessing data over multiple dimensions

The second class, *MemoryView* provides an interface to access data over multiple dimensions. You can layout and access a simple byte stream over e.g. two or more axes, providing a greater flexibility for functional operations and complex data.

Let's assume, we are reading image data from a file stream into some buffer object and want to access and manipulate the image data. Images feature two axes, one being the width, the other being the height, defining a rectangular graphics area.

When we read all data from the file, we have an one-dimensional view of the image graphics. The *MemoryView*

allows us to define a two-dimensional view over the image graphics, so that we can operate on both, rows and columns of the image.

```
>>> imagedata = bytearray("some 1-byte graphics data")
>>> view = MemoryView(imagedata, 1, (5, 5))
>>> print(view)
[[s, o, m, e,  ], [1, -, b, y, t], [e,  , g, r, a], [p, h, i, c, s], [ , d, a, t, a]]
>>> for row in view:
...     print(row)
...
[s, o, m, e,  ]
[1, -, b, y, t]
[e,  , g, r, a]
[p, h, i, c, s]
[ , d, a, t, a]
>>> for row in view:
...     row[1] = "X"
...     print row
...
[s, X, m, e,  ]
[1, X, b, y, t]
[e, X, g, r, a]
[p, X, i, c, s]
[ , X, a, t, a]
>>> print(imagedata)
sXme 1XbyteXgrapXics Xata
```

On accessing a particular dimension of a *MemoryView*, a new *MemoryView* is created, if it does not access a single element.

```
>>> firstrow = view[0]
>>> type(firstrow)
<class 'sdl2.ext.array.MemoryView'>
>>> type(firstrow[0])
<type 'bytearray'>
```

A *MemoryView* features, similar to Python's builtin memoryview, dimensions and strides, accessible via the *MemoryView.ndim* and *MemoryView.strides* attributes.

```
>>> view.ndim
2
>>> view.strides
(5, 5)
```

The *MemoryView.strides*, which have to be passed on creating a new *MemoryView*, define the layout of the data over different dimensions. In the example above, we created a 5x5 two-dimensional view to the image graphics.

```
>>> twobytes = MemoryView(imagedata, 2, (5, 1))
>>> print(twobytes)
[[sX, me,  1, Xb, yt], [eX, gr, ap, Xi, cs]]
```

Array API

class sdl2.ext.**CTypesView**(*obj : iterable*[, *itemsize=1*[, *docopy=False*[, *objsize=None*]]])

A proxy class for byte-wise accessible data types to be used in ctypes bindings. The CTypesView provides a read-write access to arbitrary objects that are iterable.

In case the object does not provide a `buffer()` interface for direct access, the CTypesView can copy the object's contents into an internal buffer, from which data can be retrieved, once the necessary operations have been performed.

Depending on the item type stored in the iterable object, you might need to provide a certain *itemsize*, which denotes the size per item in bytes. The *objsize* argument might be necessary of iterables, for which len() does not return the correct amount of objects or is not implemented.

bytesize
Returns the length of the encapsuled object in bytes.

is_shared
Indicates, if changes on the CTypesView data effect the encapsuled object directly. if not, this means that the object was copied internally and needs to be updated by the user code outside of the CTypesView.

object
The encapsuled object.

view
Provides a read-write aware view of the encapsuled object data that is suitable for usage from `ctypes`.

to_bytes() → ctypes.POINTER
Returns a byte representation of the encapsuled object. The return value allows a direct read-write access to the object data, if it is not copied. The `ctypes.POINTER()` points to an array of `ctypes.c_ubyte`.

to_uint16() → ctypes.POINTER
Returns a 16-bit representation of the encapsuled object. The return value allows a direct read-write access to the object data, if it is not copied. The `ctypes.POINTER()` points to an array of `ctypes.c_ushort`.

to_uint32() → ctypes.POINTER
Returns a 32-bit representation of the encapsuled object. The return value allows a direct read-write access to the object data, if it is not copied. The `ctypes.POINTER()` points to an array of `ctypes.c_uint`.

to_uint64() → ctypes.POINTER
Returns a 64-bit representation of the encapsuled object. The return value allows a direct read-write access to the object data, if it is not copied. The `ctypes.POINTER()` points to an array of `ctypes.c_ulonglong`.

class sdl2.ext.**MemoryView**(*source : object*, *itemsize : int*, *strides : tuple*[, *getfunc=None*[, *setfunc=None*[, *srcsize=None*]]])
The *MemoryView* provides a read-write access to arbitrary data objects, which can be indexed.

itemsize denotes the size of a single item. *strides* defines the dimensions and the length (n items * *itemsize*) for each dimension. *getfunc* and *setfunc* are optional parameters to provide specialised read and write access to the underlying *source*. *srcsize* can be used to provide the correct source size, if `len(source)` does not return the absolute size of the source object in all dimensions.

Note: The MemoryView is a pure Python-based implementation and makes heavy use of recursion for multi-dimensional access. If you aim for speed on accessing a n-dimensional object, you want to consider using a specialised library such as numpy. If you need n-dimensional access support, where such a library is not supported, or if you need to provide access to objects, which do not fulfill the requirements of that particular libray, *MemoryView* can act as solid fallback solution.

---

itemsize
The size of a single item in bytes.

ndim
The number of dimensions of the *MemoryView*.

size
The size in bytes of the underlying source object.

source
The underlying data source.

strides
A tuple defining the length in bytes for accessing all elements in each dimension of the *MemoryView*.

sdl2.ext.**to_ctypes**(*dataseq : iterable*, *dtype*[, *mcount=0*]) → array, int
Converts an arbitrary sequence to a ctypes array of the specified *dtype* and returns the ctypes array and amount of items as two-value tuple.

Raises a TypeError, if one or more elements in the passed sequence do not match the passed *dtype*.

sdl2.ext.**to_list**(*dataseq : iterable*) → list
Converts a ctypes array to a list.

sdl2.ext.**to_tuple**(*dataseq : iterable*) → tuple
Converts a ctypes array to a tuple.

sdl2.ext.**create_array**(*obj : object*, *itemsize : int*) → array.array
Creates an array.array based copy of the passed object. *itemsize* denotes the size in bytes for a single element within *obj*.

Color handling

class sdl2.ext.**Color**(*r=255*, *g=255*, *b=255*, *a=255*)
A simple RGBA-based color implementation. The Color class uses a byte-wise representation of the 4 channels red, green, blue and alpha transparency, so that the values range from 0 to 255. It allows basic arithmetic operations, e.g. color addition or subtraction and conversions to other color spaces such as HSV or CMY.

r
The red channel value of the Color.

g
The green channel value of the Color.

b
The blue channel value of the Color.

a
The alpha channel value of the Color.

cmy
The CMY representation of the Color. The CMY components are in the ranges C = [0, 1], M = [0, 1], Y = [0, 1]. Note that this will not return the absolutely exact CMY values for the set RGB values in all cases. Due to the RGB mapping from 0-255 and the CMY mapping from 0-1 rounding errors may cause the CMY values to differ slightly from what you might expect.

hsla
The HSLA representation of the Color. The HSLA components are in the ranges H = [0, 360], S = [0, 100], L = [0, 100], A = [0, 100]. Note that this will not return the absolutely exact HSL values for the set RGB values in all cases. Due to the RGB mapping from 0-255 and the HSL mapping from 0-100 and 0-360 rounding errors may cause the HSL values to differ slightly from what you might expect.

hsva
The HSVA representation of the Color. The HSVA components are in the ranges H = [0, 360], S = [0, 100], V = [0, 100], A = [0, 100]. Note that this will not return the absolutely exact HSV values for the

set RGB values in all cases. Due to the RGB mapping from 0-255 and the HSV mapping from 0-100 and 0-360 rounding errors may cause the HSV values to differ slightly from what you might expect.

i1i2i3
The I1I2I3 representation of the Color. The I1I2I3 components are in the ranges I1 = [0, 1], I2 = [-0.5, 0.5], I3 = [-0.5, 0.5]. Note that this will not return the absolutely exact I1I2I3 values for the set RGB values in all cases. Due to the RGB mapping from 0-255 and the I1I2I3 from 0-1 rounding errors may cause the I1I2I3 values to differ slightly from what you might expect.

normalize() -> (*float*, *float*, *float*, *float*)
Returns the normalised RGBA values of the Color as floating point values in the range [0, 1].

__add__(*self*, *color*) → Color
__sub__(*self*, *color*) → Color
__mul__(*self*, *color*) → Color
__div__(*self*, *color*) → Color
__truediv__(*self*, *color*) → Color
__mod__(*self*, *color*) → Color
Basic arithmetic functions for *Color* values. The arithmetic operations `+, -, *, /, %` are supported by the *Color* class and work on a per-channel basis. This means, that the operation

```
color = color1 + color2
```

is the same as

```
color = Color()
color.r = min(color1.r + color2.r, 255)
color.g = min(color1.g + color2.g, 255)
...
```

The operations guarantee that the channel values stay in the allowed range of [0, 255].

sdl2.ext.**argb_to_color**(*v : int*) → Color
sdl2.ext.**ARGB**(*v : int*) → Color
Converts an integer value to a Color, assuming the integer represents a 32-bit ARGB value.

sdl2.ext.**convert_to_color**(*v : object*) → Color
sdl2.ext.**COLOR**(*v : object*) → Color
Tries to convert the passed value to a Color object. The value can be an arbitrary Python object, which is passed to the different other conversion functions. If one of them succeeds, the Color will be returned to the caller. If none succeeds, a `ValueError` will be raised.

If the color is an integer value, it is assumed to be in ARGB layout.

sdl2.ext.**rgba_to_color**(*v : int*) → Color
sdl2.ext.**RGBA**(*v : int*) → Color
Converts an integer value to a Color, assuming the integer represents a 32-bit RGBA value.

sdl2.ext.**is_rgb_color**(*v : object*) → bool
Checks, if the passed value is an item that could be converted to a RGB color.

sdl2.ext.**is_rgba_color**(*v : object*) → bool
Checks, if the passed value is an item that could be converted to a RGBA color.

sdl2.ext.**string_to_color**(*v : string*) → Color
Converts a hex color string or color name to a Color value. Supported hex values are:

- #RGB
- #RGBA

- #RRGGBB
- #RRGGBBAA
- 0xRGB
- 0xRGBA
- 0xRRGGBB
- 0xRRGGBBAA

sdl2.ext.colorpalettes - predefined sets of colors

Indexed color palettes. Each palette is a tuple of *sdl2.ext.Color* objects.

The following palettes are currently available:

Palette Identifier	Description
MONOPALETTE	1-bit monochrome palette (black and white).
GRAY2PALETTE	2-bit grayscale palette with black, white and two shades of gray.
GRAY4PALETTE	4-bit grayscale palette with black, white and 14 shades shades of gray.
GRAY8PALETTE	8-bit grayscale palette with black, white and 254 shades shades of gray.
RGB3PALETTE	3-bit RGB color palette with pure red, green and blue and their complementary colors as well as black and white.
CGA-PALETTE	CGA color palette.
EGAPALETTE	EGA color palette.
VGA-PALETTE	8-bit VGA color palette.
WEB-PALETTE	"Safe" web color palette with 225 colors.

Initialization routines

TODO

API

exception `sdl2.ext.`**`SDLError`**(*msg=None*)
An SDL2 specific `Exception` class. if no *msg* is provided, the message will be set to the value of `sdl2.error.SDL_GetError()`

`sdl2.ext.`**`init`**`()` → None
Initialises the underlying SDL2 video subsystem. Raises a *SDLError*, if the SDL2 video subsystem could not be initialised.

`sdl2.ext.`**`quit`**`()` → None
Quits the underlying SDL2 video subysystem. If no other SDL2 subsystems are active, this will also call *quit()*, `sdl2.sdlttf.TTF_Quit()` and `sdl2.sdlimage.IMG_Quit()`.

`sdl2.ext.`**`get_events`**`()` → [SDL_Event, SDL_Event, ...]
Gets all SDL events that are currently on the event queue.

class `sdl2.ext.`**`TestEventProcessor`**
A simple event processor for testing purposes.

run(*window : Window*) → None
Starts an event loop without actually processing any event. The method will run endlessly until a SDL_QUIT event occurs.

sdl2.ext.compat - Python compatibility helpers

The *sdl2.ext.compat* module is for internal purposes of the *sdl2* package and should not be used outside of the package. Classes, methods and interfaces might change between versions and there is no guarantee of API compatibility on different platforms and python implementations or between releases.

sdl2.ext.compat.**ISPYTHON2**
True, if executed in a Python 2.x compatible interpreter, False otherwise.

sdl2.ext.compat.**ISPYTHON3**
True, if executed in a Python 3.x compatible interpreter, False otherwise.

sdl2.ext.compat.**long**([*x*[, *base*]])

Note: Only defined for Python 3.x, for which it is the same as int().

---

sdl2.ext.compat.**unichr**(*i*)

Note: Only defined for Python 3.x, for which it is the same as chr().

---

sdl2.ext.compat.**unicode**(*string*[, *encoding*[, *errors*]])

Note: Only defined for Python 3.x, for which it is the same as str().

---

sdl2.ext.compat.**callable**(*x*) → bool

Note: Only defined for Python 3.x, for which it is the same as isinstance(x, collections.Callable)

---

sdl2.ext.compat.**byteify**(*x : string*, *enc : string*) → bytes
Converts a string to a bytes() object.

sdl2.ext.compat.**stringify**(*x : bytes*, *enc : string*) → string
Converts a bytes() to a string object.

sdl2.ext.compat.**isiterable**(*x*) → bool
Shortcut for isinstance(x, collections.Iterable).

sdl2.ext.compat.**platform_is_64bit**() → bool
Checks, if the interpreter is 64-bit capable.

@sdl2.ext.compat.**deprecated**
A simple decorator to mark functions and methods as deprecated. This will print a deprecation message each time the function or method is invoked.

sdl2.ext.compat.**deprecation**(*message : string*) → None
Prints a deprecation message using the warnings.warn() function.

exception sdl2.ext.compat.**UnsupportedError**(*obj : object*[, *msg=None*])
Indicates that a certain class, function or behaviour is not supported in the specific execution environment.

@sdl2.ext.compat.**experimental**
A simple decorator to mark functions and methods as experimental. This will print a warning each time the function or method is invoked.

exception sdl2.ext.compat.**ExperimentalWarning**(*obj : object*[, *msg=None*])
Indicates that a certain class, function or behaviour is in an experimental state.

2D drawing routines for software surfaces

Note: The drawing functions within this module are unoptimised and should not be considered fast. If you want improved drawing of 2D primitives, including hardware acceleration, you should use the methods of the *Renderer* instead.

---

sdl2.ext.**prepare_color**(*color : object*, *target : object*) → int
Prepares the passed *color* for a specific *target*. *color* can be any object type that can be processed by *convert_to_color()*. *target* can be any sdl2.SDL_PixelFormat, sdl2.SDL_Surface or *SoftwareSprite* instance.

The returned integer will be a color value matching the target's pixel format.

sdl2.ext.**fill**(*target : object*, *color : object*[, *area=None*]) → None
Fills a certain area on the passed *target* with a *color*. If no *area* is provided, the entire target will be filled with the passed color. If an iterable item is provided as *area* (such as a list or tuple), it will be first checked, if the item denotes a single rectangular area (4 integer values) before assuming it to be a sequence of rectangular areas to fill with the color.

target can be any sdl2.SDL_Surface or *SoftwareSprite* instance.

sdl2.ext.**line**(*target : object*, *color : object*[, *width=1*]) → None
Draws one or multiple lines on the passed *target*. *line* can be a sequence of four integers for a single line in the form (x1, y1, x2, y2) or a sequence of a multiple of 4 for drawing multiple lines at once, e.g. (x1, y1, x2, y2, x3, y3, x4, y4, ...).

target can be any sdl2.SDL_Surface or *SoftwareSprite* instance.

Working with component-based entities

sdl2.ext supports a component oriented programming pattern to separate object instances, carried data and processing logic within applications or games. It uses an entity based approach, in which object instances are unique identifiers, while their data is managed within components, which are stored separately. For each individual component type a processing system will take care of all necessary updates on running the application.

Component-based patterns

Component-based means that - instead of a traditional OOP approach - object information are split up into separate data bags for reusability and that those data bags are separated from any application logic.

Behavioural design

Imagine a car class in traditional OOP, which might look like

```
class Car:
    def __init__(self):
        self.color = "red"
        self.position = 0, 0
        self.velocity = 0, 0
        self.sprite = get_some_car_image()
        ...
    def drive(self, timedelta):
        self.position[0] = self.velocity[0] * timedelta
        self.position[1] = self.velocity[1] * timedelta
        ...
    def stop(self):
        self.velocity = 0, 0
        ...
    def render(self, screen):
        screen.display(self.sprite)

mycar = new Car()
mycar.color = "green"
mycar.velocity = 10, 0
```

The car features information stored in attributes (`color`, `position`, ...) and behaviour (application logic, `drive()`, `stop()` ...).

A component-based approach aims to split and reduce the car to a set of information and external systems providing the application logic.

```
class Car:
    def __init__(self):
        self.color = "red"
        self.position = 0, 0
        self.velocity = 0, 0
        self.sprite = get_some_car_image()

class CarMovement:
    def drive(self, car, timedelta):
        car.position[0] = car.velocity[0] * timedelta
        car.position[1] = car.velocity[1] * timedelta
        ...
    def stop(self):
        car.velocity = 0, 0

class CarRenderer:
    def render(self, car, screen):
        screen.display(car.sprite)
```

At this point of time, there is no notable difference between both approaches, except that the latter one adds additional overhead.

The benefit comes in, when you

- use subclassing in your OOP design
- want to change behavioural patterns on a global scale or based on states
- want to refactor code logic in central locations

- want to cascade application behaviours

The initial `Car` class from above defines, how it should be displayed on the screen. If you now want to add a feature for rescaling the screen size after the user activates the magnifier mode, you need to refactor the `Car` and all other classes that render things on the screen, have to consider all subclasses that override the method and so on. Refactoring the `CarRenderer` code by adding a check for the magnifier mode sounds quite simple in contrast to that, not?

The same applies to the movement logic - inverting the movement logic requires you to refactor all your classes instead of a single piece of application code.

Information design

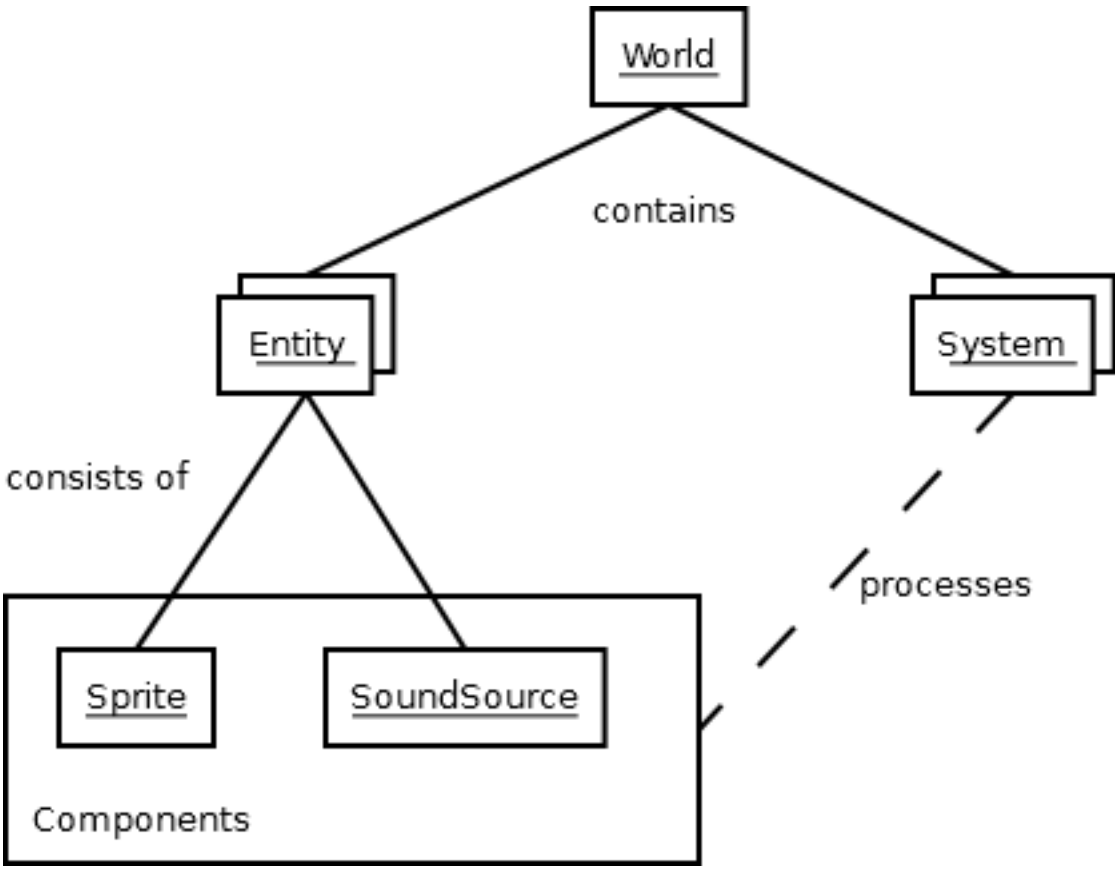
Subclassing with traditional OOP for behavioural changes also might bloat your classes with unnecessary information, causing the memory footprint for your application to rise without any need. Let's assume you have a `Truck` class that inherits from `Car`. Let's further assume that all trucks in your application look the same. Why should any of those carry a `sprite` or `color` attribute? You would need to refactor your `Car` class to get rid of those superfluous information, adding another level of subclassing. If at a later point of time you decide to give your trucks different colors, you need to refactor everything again.

Wouldn't it be easier to deal with colors, if they are available on the truck and leave them out, if they are not? We initially stated that the component-based approach aims to separate data (information) from code logic. That said, if the truck has a color, we can handle it easily, if it has not, we will do as usual.

Also, checking for the color of an object (regardless, if it is a truck, car, aeroplane or death star) allows us to apply the same or similar behaviour for every object. If the information is available, we will process it, if it is not, we will not do anything.

All in all

Once we split up the previously OOP-style classes into pure data containers and some separate processing code for the behaviour, we are talking about components and (processing) systems. A component is a data container, ideally grouping related information on a granular level, so that it is easy to (re)use. When you combine different components to build your in-application objects and instantiate those, we are talking about entities.



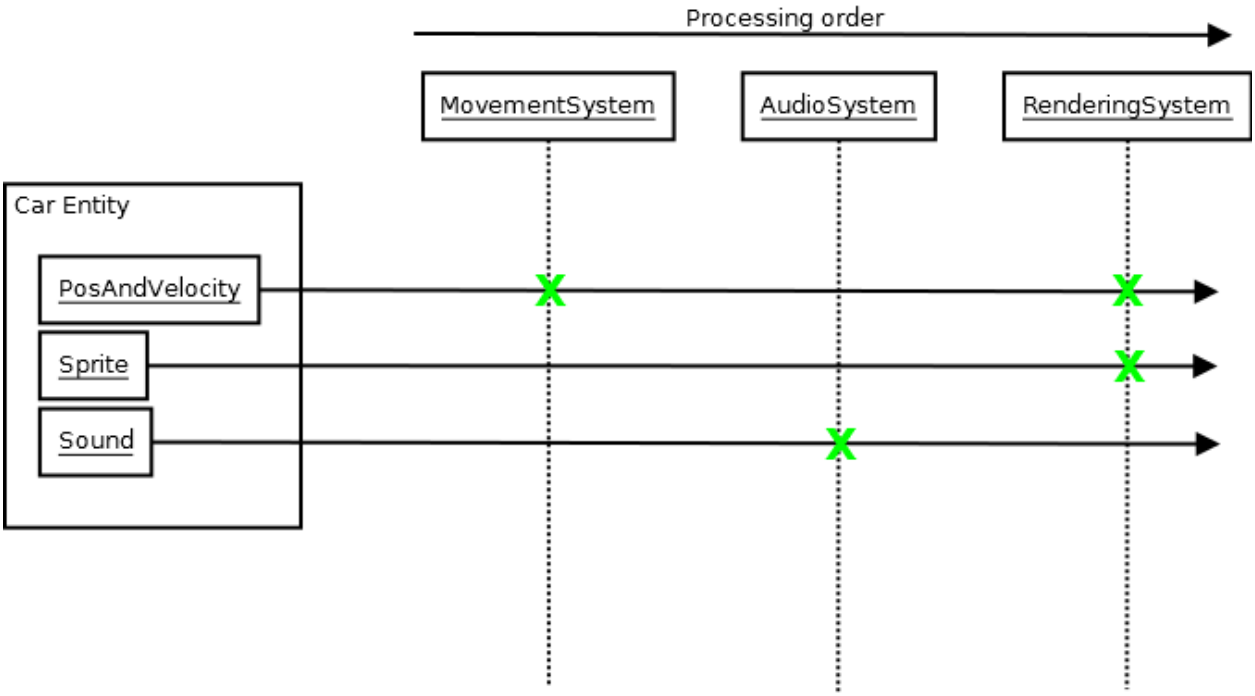
Component provides information (data bag)

Entity In-application instance that consists of *component* items

System Application logic for working with *Entity* items and their *component* data

World The environment that contains the different *System* instances and all *Entity* items with their *component* data

Within a strict COP design, the application logic (ideally) only knows about data to process. It does not know anything about entities or complex classes and only operates on the data.



To keep things simple, modular and easy to maintain and change, you usually create small processing systems, which perform the necessary operations on the data they shall handle. That said, a `MovementSystem` for our car entity would only operate on the position and velocity component of the car entity. It does not know anything about the the car's sprite or sounds that the car makes, since *this is nothing it has to deal with*.

To display the car on the screen, a `RenderingSystem` might pick up the sprite component of the car, maybe along with the position information (so it knows, where to place the sprite) and render it on the screen.

If you want the car to play sounds, you would add an audio playback system, that can perform the task. Afterwards you can add the necessary audio information via a sound component to the car and it will make noise.

Component-based design with sdl2.ext

Note: This section will deal with the specialities of COP patterns and provide the bare minimum of information. If you are just starting with such a design, it is recommended to read through the *The Pong Game* tutorial.

---

sdl2.ext provides a *World* class in which all other objects will reside. The *World* will maintain both, *Entity* and component items, and allows you to set up the processing logic via the *System* and *Applicator* classes.

```
>>> appworld = World()
```

Components can be created from any class that inherits from the `object` type and represent the data bag of information for the entity and application world. Ideally, they should avoid any application logic (except from getter and setter properties).

```
class Position2D(object):
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
```

Entity objects define the in-application objects and only consist of component-based attributes. They also require a *World* at object instantiation time.

```
class CarEntity(Entity):
    def __init__(self, world, x=0, y=0):
        self.position2d = Position2D(x, y)
```

Note: The *world* argument in `__init__()` is necessary. It will be passed to the internal `__new__()` constructor of the *Entity* and stores a reference to the *World* and also allows the *Entity* to store its information in the *World*.

The *Entity* also requries its attributes to be named exactly as their component class name, but in lowercase letters. If you name a component `MyAbsolutelyAwesomeDataContainer`, an *Entity* will force you to write the following:

```
class SomeEntity(Entity):
    def __init__(self, world):
        self.myabsolutelyawesomedatacontainer = MyAbsolutelyAwesomeDataContainer()
```

Note: This is not entirely true. A reference of the object will be stored on a per-class-in-mro basis. This means that if `MyAbsolutelyAwesomeDataContainer` inherits from `ShortName`, you can also do:

```
class SomeEntity(Entity):
    def __init__(self, world):
        self.shortname = MyAbsolutelyAwesomeDataContainer()
```

Components should be as atomic as possible and avoid complex inheritance. Since each value of an *Entity* is stored per class in its mro list, components inheriting from the same class(es) will overwrite each other on conflicting classes:

```
class Vector(Position2D):
    def __init__(self, x=0, y=0, z=0):
        super(Vector, self).__init__(x, y)


class SomeEntity(Entity):
    def __init__(self, world):
        # This will associate self.position2d with the new Position2D
        # value, while the previous Vector association is overwritten
        self.position2d = Position2D(4, 4)

        # self.vector will also associate a self.position2d attribute
        # with the Entity, since Vector inherits from Position2D. The
        # original association will vanish, and each call to
        # entity.position2d will effectively manipulate the vector!
        self.vector = Vector(1,2,3)
```

API

class `sdl2.ext.`**`Entity`**`(`*world : World*`)`

An entity is a specific object living in the application world. It does not carry any data or application logic, but merely acts as identifier label for data that is maintained in the application world itself.

As such, it is a composition of components, which would not exist without the entity identifier. The entity itself is non-existent to the application world as long as it does not carry any data that can be processed by a system within the application world.

id
The id of the Entity. Every Entity has a unique id, that is represented by a `uuid.UUID` instance.

world
The *World* the entity resides in.

delete() → None
Deletes the *Entity* from its *World*. This basically calls *World.delete()* with the *Entity*.

class sdl2.ext.**Applicator**
A processing system for combined data sets. The *Applicator* is an enhanced *System* that receives combined data sets based on its set *System.componenttypes*

is_applicator
A boolean flag indicating that this class operates on combined data sets.

componenttypes
A tuple of class identifiers that shall be processed by the *Applicator*.

process(*world : World*, *componentsets : iterable*)
Processes tuples of component items. *componentsets* will contain object tuples, that match the *componenttypes* of the *Applicator*. If, for example, the *Applicator* is defined as

```
class MyApplicator(Applicator):
    def __init__(self):
        self.componenttypes = (Foo, Bar)
```

its process method will receive `(Foo, Bar)` tuples

```
def process(self, world, componentsets):
    for foo_item, bar_item in componentsets:
        ...
```

Additionally, the *Applicator* will not process all possible combinations of valid components, but only those, which are associated with the same *Entity*. That said, an *Entity* *must* contain a `Foo` as well as a `Bar` component in order to have them both processed by the *Applicator* (while a *System* with the same `componenttypes` would pick either of them, depending on their availability).

class sdl2.ext.**System**
A processing system within an application world consumes the components of all entities, for which it was set up. At time of processing, the system does not know about any other component type that might be bound to any entity.

Also, the processing system does not know about any specific entity, but only is aware of the data carried by all entities.

componenttypes
A tuple of class identifiers that shall be processed by the *System*

process(*world : World*, *components : iterable*)
Processes component items.

This method has to be implemented by inheriting classes.

class sdl2.ext.**World**
An application world defines the combination of application data and processing logic and how the data will be processed. As such, it is a container object in which the application is defined.

The application world maintains a set of entities and their related components as well as a set of systems that process the data of the entities. Each processing system within the application world only operates on a certain set of components, but not all components of an entity at once.

The order in which data is processed depends on the order of the added systems.

systems
The processing system objects bound to the world.

add_system(*system : object*)
Adds a processing system to the world. The system will be added as last item in the processing order.

The passed system does not have to inherit from *System*, but must feature a `componenttypes` attribute and a `process()` method, which match the signatures of the *System* class

```
class MySystem(object):
    def __init__(self):
        # componenttypes can be any iterable as long as it
        # contains the classes the system should take care of
        self.componenttypes = [AClass, AnotherClass, ...]

    def process(self, world, components):
        ...
```

If the system shall operate on combined component sets as specified by the *Applicator*, the class instance must contain a `is_applicator` property, that evaluates to `True`

```
class MyApplicator(object):
    def __init__(self):
        self.is_applicator = True
        self.componenttypes = [...]

    def process(self, world, components):
        pass
```

The behaviour can be changed at run-time. The `is_applicator` attribute is evaluated for every call to *World.process()*.

delete(*entity : Entity*)
Removes an *Entity* from the World, including all its component data.

delete_entities(*entities : iterable*)
Removes a set of *Entity* instances from the World, including all their component data.

insert_system(*index : int*, *system : System*)
Adds a processing *System* to the world. The system will be added at the specified position in the processing order.

get_entities(*component : object*) → [Entity, ...]
Gets the entities using the passed component.

Note: This will not perform an identity check on the component but rely on its `__eq__` implementation instead.

---

process()
Processes all component items within their corresponding *System* instances.

remove_system(*system : System*)
Removes a processing *System* from the world.

General purpose event handling routines

class sdl2.ext.**EventHandler**(*sender*)
A simple event handling class, which manages callbacks to be executed.

The EventHandler does not need to be kept as separate instance, but is mainly intended to be used as attribute in event-aware class objects.

```
>>> def myfunc(sender):
...     print("event triggered by %s" % sender)
...
>>> class MyClass(object):
...     def __init__(self):
...         self.anevent = EventHandler(self)
...
>>> myobj = MyClass()
>>> myobj.anevent += myfunc
>>> myobj.anevent()
event triggered by <__main__.MyClass object at 0x801864e50>
```

callbacks
A list of callbacks currently bound to the *EventHandler*.

sender
The responsible object that executes the *EventHandler*.

add(*callback : Callable*)
Adds a callback to the *EventHandler*.

remove(*callback : Callable*)
Removes a callback from the *EventHandler*.

__call__(**args*) → [...]
Executes all connected callbacks in the order of addition, passing the *sender* of the *EventHandler* as first argument and the optional args as second, third, ... argument to them.

This will return a list containing the return values of the callbacks in the order of their execution.

class sdl2.ext.**MPEventHandler**(*sender*)
An asynchronous event handling class based on *EventHandler*, in which callbacks are executed in parallel. It is the responsibility of the caller code to ensure that every object used maintains a consistent state. The *MPEventHandler* class will not apply any locks, synchronous state changes or anything else to the arguments or callbacks being used. Consider it a "fire-and-forget" event handling strategy.

Note: The *MPEventHandler* relies on the multiprocessing module. If the module is not available in the target environment, a *sdl2.ext.compat.UnsupportedError* is raised.

Also, please be aware of the restrictions that apply to the multiprocessing module; arguments and callback functions for example have to be pickable, etc.

__call__(**args*) → AsyncResult
Executes all connected callbacks within a multiprocessing.pool.Pool, passing the sender as first argument and the optional *args* as second, third, ... argument to them.

This will return a multiprocessing.pool.AsyncResult containing the return values of the callbacks in the order of their execution.

Text rendering routines

class sdl2.ext.**BitmapFont**(*surface : Sprite, size : iterable[, mapping=None*)

A bitmap graphics to character mapping. The *BitmapFont* class uses an image *surface* to find and render font character glyphs for text. It requires a mapping table, which denotes the characters available on the image.

The mapping table is a list of strings, where each string reflects a *line* of characters on the image. Each character within each line has the same size as specified by the size argument.

A typical mapping table might look like

```
[ '0123456789',
  'ABCDEFGHIJ',
  'KLMNOPQRST',
  'UVWXYZ    ',
  'abcdefghij',
  'klmnopqrst',
  'uvwxyz    ',
  ',;.:!?+-()' ]
```

surface

The sdl2.SDL_Surface containing the character bitmaps.

offsets

A dict containing the character offsets on the *surface*.

mapping

The character mapping table, a list of strings.

size

The size of an individual glyph bitmap on the font.

render(*text : string*[, *bpp=None*]) → Sprite

Renders the passed text on a new *Sprite* and returns it. If no explicit *bpp* are provided, the bpp settings of the *surface* are used.

render_on(*surface : Sprite, text : string[, offset=(0, 0)]) -> (int, int, int, int*)

Renders a text on the passed sprite, starting at a specific offset. The top-left start position of the text will be the passed *offset* and a 4-value tuple with the changed area will be returned.

contains(*c : string*) → bool

Checks, whether a certain character exists in the font.

can_render(*text : string*) → bool

Checks, whether all characters in the passed *text* can be rendered.

class sdl2.ext.**FontManager**(*font_path : str*[, *alias=None*[, *size=16*[, *color=Color(255, 255, 255)*[, *bg_color=Color(0, 0, 0)*[, *index=0*]]]]])

Manage fonts and rendering of text.

One font path must be given to initialise the FontManager. *default_font* will be set to this font. *size* is the default font size in pixels. *color* and *bg_color* will give the FontManager a default color. *index* will select a specific font face from a file containing multiple font faces. The first face is always at index 0. It can be used for TTC (TrueType Font Collection) fonts.

bg_color

The *sdl2.ext.Color* to be used as background color.

color

The *sdl2.ext.Color* to be used for rendering text.

default_font
Returns the name of the current default font being used by the *FontManager*. On assigning *default_font*, the value must be a loaded font alias.

size
The default font size in pixels.

add(*font_path : str[, alias=None[, size=None[, index=0]]]*) → sdl2.sdlttf.TTF_Font
Add a font to the *FontManager*. *alias* is by default the font name, any other name can be passed, *size* is the font size in pixels and defaults to *size*. *index* selects a specific font face from a TTC (TrueType Font Collection) file. Returns the font pointer stored in `fonts`.

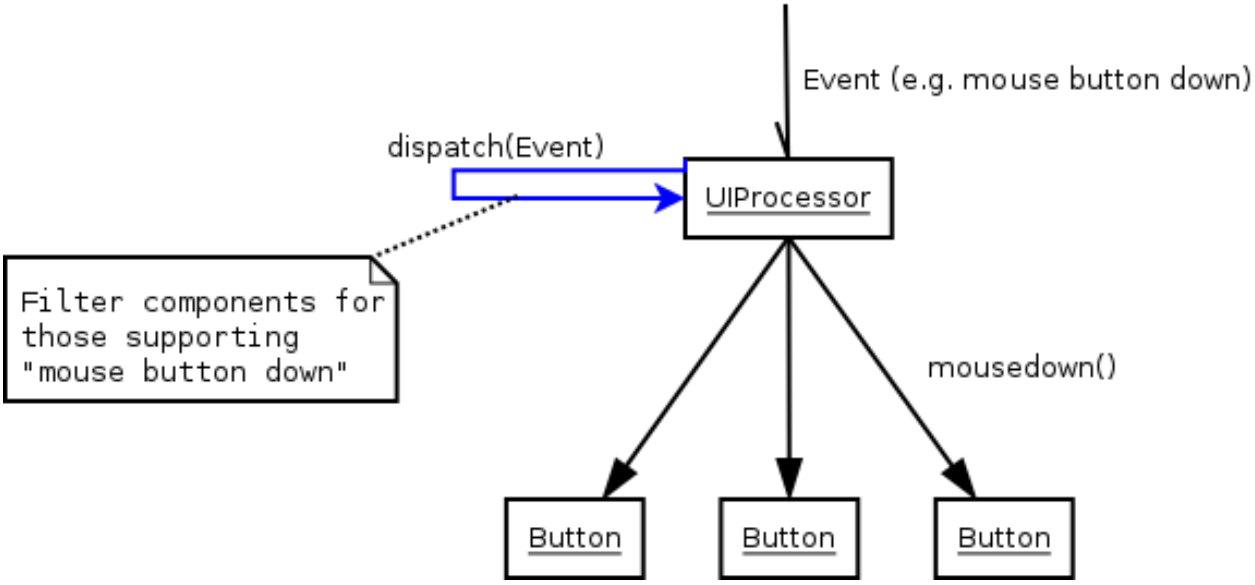
close()
Closes all fonts used by the *FontManager*.

render(*text : str*[, *alias=None*[, *size=None*[, *width=None*[, *color=None*[, *bg_color=None*[, ***kwargs*]]]]]]) → sdl2.SDL_Surface
Renders text to a surface. This method uses the font designated by the passed *alias* or, if *alias* is omitted, by the set *default_font*. A *size* can be passed even if the font was not loaded with this size. A *width* can be given for automatic line wrapping. If no *bg_color* or *color* are given, it will default to the FontManager's *bg_color* and *color*.

User interface elements

User interface elements within *sdl2.ext* are simple *Sprite* objects, which are enhanced by certain input hooks; as such, they are not classes on their own, but implemented as mixins. The user input itself is handled by an *UIProcessor* object, which takes care of delegating input events, such as mouse movements, clicks and keyboard input, to the correct UI element.

Depending on the event type (e.g. pressing a mouse button), the UIProcessor will execute its matching method (e.g. `mousedown()`) with only those UI elements, which support the event type.



UI element types

Every *sdl2.ext* UI element is a simple *Sprite* object, to which additional attributes and methods are bound.

Every UI element features the following attributes

element.uitype

The uitype attribute can have one of the following values, identifying the UI element:

- BUTTON - a UI element, which can react on mouse input
- CHECKBUTTON - as BUTTON, but it retains its state on clicks
- TEXTENTRY - a UI element that reacts on keyboard input

element.events

A dictionary containing the SDL2 event mappings. Each supported SDL2 event (e.g. SDL_MOUSEMOTION) is associated with a bound *EventHandler* acting as callback for user code (e.g. mousemotion()).

Depending on the exact type of the element, it will feature additional methods and attributes explained below.

Button elements

BUTTON UI elements feature a state attribute, which can be one of the following values.

state	Description
RELEASED	Indicates that the UI element is not pressed.
HOVERED	Indicates that the mouse cursor is currently hovering the UI element.
PRESSED	Indicates that a mouse button is pressed on the UI element.

BUTTON UI elements react with the following event handlers on events:

button.motion(event : sdl2.events.SDL_Event)

An *EventHandler* that is invoked, if the mouse moves around while being over the BUTTON.

button.pressed(event : sdl2.events.SDL_Event)

An *EventHandler* that is invoked, if a mouse button is pressed on the BUTTON.

button.released(event : sdl2.events.SDL_Event)

An *EventHandler* that is invoked, if a mouse button is released on the BUTTON.

button.click(event : sdl2.events.SDL_Event)

An *EventHandler* that is invoked, if a mouse button is pressed and released on the BUTTON.

Besides the BUTTON a special CHECKBUTTON UI element type exists, which enhances the BUTTON bindings by an additional checked attribute. The checked attribute switches its status (False to True and True to False) every time the UI element is clicked.

Text input elements

TEXTENTRY elements react on text input, once they are activated. Text being input, once a TEXTENTRY has been activated, is stored in its text attribute.

The TEXTENTRY reacts with the following event handlers on events:

textentry.motion(event : sdl2.events.SDL_Event)

An *EventHandler* that is invoked, if the mouse moves around while being over the TEXTENTRY.

textentry.pressed(event : sdl2.events.SDL_Event)

An *EventHandler* that is invoked, if a mouse button is pressed on the TEXTENTRY.

`textentry.released(event : sdl2.events.SDL_Event)`

An *EventHandler* that is invoked, if a mouse button is released on the `TEXTENTRY`.

`textentry.keydown(event : sdl2.events.SDL_Event)`

An *EventHandler* that is invoked on pressing a key.

`textentry.keyup(event : sdl2.events.SDL_Event)`

An *EventHandler* that is invoked on releasing a key.

`textentry.input(event : sdl2.events.SDL_Event)`

An *EventHandler* that is invoked on text input events. Text input events are automatically created, once the *UIProcessor* activates a `TEXTENTRY` UI element.

`textentry.editing(event : sdl2.events.SDL_Event)`

An *EventHandler* that is invoked on text editing events. Text editing events are automatically created, once the *UIProcessor* activates a `TEXTENTRY` UI element.

Text editing events are however only raised, if an IME system is involved, which combines glyphs and symbols to characters or word fragments.

API

class `sdl2.ext.`**`UIFactory`**(*spritefactory : SpriteFactory*[, ***kwargs*])

A factory class for creating UI elements. The *UIFactory* allows you to create UI elements based on the *Sprite* class. To do this, it requires a *SpriteFactory*, which will create the sprites, to which the *UIFactory* then binds the additional methods and attributes.

The additional *kwargs* are used as default arguments for creating **sprites** within the factory methods.

`default_args`

A dictionary containing the default arguments to be passed to the sprite creation methods of the bound *SpriteFactory*.

`spritefactory`

The *SpriteFactory* being used for creating new *Sprite* objects.

`create_button`(***kwargs*) → Sprite

Creates a new button UI element.

kwargs are the arguments to be passed for the sprite construction and can vary depending on the sprite type. See *SpriteFactory.create_sprite()* for further details.

`create_check_button`(***kwargs*) → Sprite

Creates a new checkbutton UI element.

kwargs are the arguments to be passed for the sprite construction and can vary depending on the sprite type. See *SpriteFactory.create_sprite()* for further details.

`create_text_entry`(***kwargs*) → Sprite

Creates a new textentry UI element.

kwargs are the arguments to be passed for the sprite construction and can vary depending on the sprite type. See *SpriteFactory.create_sprite()* for further details.

`from_color`(*color : object*, *size*) → Sprite

Creates a UI element with a specific color.

uitype must be one of the supported *UI element types* classifying the type of UI element to be created.

from_image(*uitype : int*, *fname : str*) → Sprite
Creates a UI element from an image file. The image must be loadable via *load_image()*.

uitype must be one of the supported *UI element types* classifying the type of UI element to be created.

from_object(*uitype : int*, *obj: object*) → Sprite
Creates a UI element from an object. The object will be passed through `sdl2.rwops_from_object()` in order to try to load image data from it.

uitype must be one of the supported *UI element types* classifying the type of UI element to be created.

from_surface(*uitype : int*, *surface : SDL_Surface*[, *free=False*]) → Sprite
Creates a UI element from the passed `sdl2.surface.SDL_Surface`. If *free* is set to `True`, the passed *surface* will be freed automatically.

uitype must be one of the supported *UI element types* classifying the type of UI element to be created.

class `sdl2.ext.`**UIProcessor**
A processing system for user interface elements and events.

handlers
A dict containing the mapping of SDL2 events to the available *EventHandler* bindings of the *UIProcessor*.

activate(*component : object*) → None
Activates a UI control to receive text input.

deactivate(*component : object*) → None
Deactivate the currently active UI control.

passevent(*component : object*, *event : SDL_Event*) → None
Passes the *event* to a *component* without any additional checks or restrictions.

mousemotion(*component : object*, *event : SDL_Event*) → None
Checks, if the event's motion position is on the *component* and executes the component's event handlers on demand. If the motion event position is not within the area of the *component*, nothing will be done. In case the component is a `BUTTON`, its `state` will be adjusted to reflect, if it is currently hovered or not.

mousedown(*component : object*, *event : SDL_Event*) → None
Checks, if the event's button press position is on the *component* and executes the component's event handlers on demand. If the button press position is not within the area of the component, nothing will be done.

In case the component is a `BUTTON`, its `state` will be adjusted to reflect, if it is currently pressed or not.

In case the component is a `TEXTENTRY` and the pressed button is the primary mouse button, the component will be marked as the next control to activate for text input.

mouseup(*self*, *component*, *event*) → None
Checks, if the event's button release position is on the *component* and executes the component's event handlers on demand. If the button release position is not within the area of the component, nothing will be done.

In case the component is a `BUTTON`, its `state` will be adjusted to reflect, whether it is hovered or not.

If the button release followed a button press on the same component and if the button is the primary button, the `click()` event handler is invoked, if the component is a `BUTTON`.

dispatch(*obj : object*, *event : SDL_Event*) → None
Passes an event to the given object. If *obj* is a *World* object, UI relevant components will receive the event, if they support the event type. If *obj* is a single object, `obj.events` **must** be a dict consisting of SDL event type identifiers and *EventHandler* instances bound to the object. If *obj* is a iterable, such as a list or set, every item within *obj* **must** feature an `events` attribute as described above.

process (*world : World*, *components : iterable*) → None
The *UIProcessor* class does not implement the *process()* method by default. Instead it uses *dispatch()* to send events around to components. *process()* does nothing.

Image loaders

sdl2.ext.**get_image_formats** (*) -> (*str*, *str*, ...)
Gets the formats supported by PySDL2 in the default installation.

sdl2.ext.**load_image** (*fname : str*[, *enforce=None*]) → sdl2.SDL_Surface
Creates a sdl2.SDL_Surface from an image file.

This function makes use of the Python Imaging Library, if it is available on the target execution environment. The function will try to load the file via *sdl2* first. If the file could not be loaded, it will try to load it via *sdl2.sdlimage* and PIL.

You can force the function to use only one of them, by passing the *enforce* as either "PIL" or "SDL".

Note: This will call sdl2.sdlimage.IMG_Init() implicitly with the default arguments, if the module is available and if sdl2.SDL_LoadBMP() failed to load the image.

---

sdl2.ext.particles - A simple particle system

class sdl2.ext.particles.**ParticleEngine**
A simple particle processing system. The *ParticleEngine* takes care of creating, updating and deleting particles via callback functions. It only decreases the life of the particles by itself and marks them as dead, once the particle's life attribute has reached 0 or below.

createfunc
Function for creating new particles. The function needs to take two arguments, the world argument passed to *process()* and a list of the particles considered dead (*Particle.life* <= 0).

```
def creation_func(world, deadparticles):
    ...
```

updatefunc
Function for updating existing, living particles. The function needs to take two arguments, the world argument passed to *process()* and a set of the still living particles.

```
def update_func(world, livingparticles):
    ...
```

deletefunc
Function for deleting dead particles. The function needs to take two arguments, the world argument passed to *process()* and a list of the particles considered dead (*Particle.life* <= 0).

```
def deletion_func(world, deadparticles):
    ...
```

process (*world : World*, *components : iterable*) → None
Processes all particle components, decreasing their life by 1.

Once the life of all particle components has been decreased properly and the particles considered dead (life <= 0) are identified, the creation, update and deletion callbacks are invoked.

The creation callback takes the passed world as first and the list of dead particles as second argument.

```
def particle_createfunc(world, list_of_dead_ones):
    ...
```

Afterwards the still living particles are passed to the update callback, which also take the passed world as first and the living particles as set as second argument.

```
def particle_updatefunc(world, set_of_living_ones):
    ...
```

Finally, the dead particles need to be deleted in some way or another, which is done by the deletion callback, taking the passed world as first and the list of dead particles as second argument.

```
def particle_deletefunc(world, list_of_dead_ones):
    ...
```

class sdl2.ext.particles.**Particle**(*x*, *y*, *life : int*)

A simple particle component type. It only contains information about a x- and y-coordinate and its current life time. The life time will be decreased by 1, every time the particle is processed by the *ParticleEngine*.

x

The x coordinate of the particle.

y

The y coordinate of the particle.

life

The remaining life time of the particle.

position

The x- and y-coordinate of the particle as tuple.

2D and 3D direct pixel access

class sdl2.ext.**PixelView**(*source : object*)

2D *MemoryView* for *SoftwareSprite* and sdl2.SDL_surface pixel access.

Note:

If necessary, the *source* surface will be locked for accessing its pixel data. The lock will be removed once the *PixelView* is garbage-collected or deleted.

The *PixelView* uses a y/x-layout. Accessing `view[N]` will operate on the Nth row of the underlying surface. To access a specific column within that row, `view[N][C]` has to be used.

Note: *PixelView* is implemented on top of the *MemoryView* class. As such it makes heavy use of recursion to access rows and columns and can be considered as slow in contrast to optimised ndim-array solutions such as numpy.

---

sdl2.ext.**pixels2d**(*source : object*)

Creates a 2D pixel array, based on numpy.ndarray, from the passed *source*. *source* can be a *SoftwareSprite* or sdl2.SDL_Surface. The SDL_Surface of the *source* will be locked and unlocked automatically.

The *source* pixels will be accessed and manipulated directly.

Note: *pixels2d()* is only usable, if the numpy package is available within the target environment. If numpy could not be imported, a *sdl2.ext.compat.UnsupportedError* will be raised.

---

sdl2.ext.**pixels3d**(*source : object*)

Creates a 3D pixel array, based on numpy.ndarray, from the passed *source*. *source* can be a *SoftwareSprite* or sdl2.SDL_Surface. The SDL_Surface of the *source* will be locked and unlocked automatically.

The *source* pixels will be accessed and manipulated directly.

Note: *pixels3d()* is only usable, if the numpy package is available within the target environment. If numpy could not be imported, a *sdl2.ext.compat.UnsupportedError* will be raised.

Resource management

Every application usually ships with various resources, such as image and data files, configuration files and so on. Accessing those files in the folder hierarchy or in a bundled format for various platforms can become a complex task. The *Resources* class allows you to manage different application data in a certain directory, providing a dictionary-style access functionality for your in-application resources.

Let's assume, your application has the following installation layout

```
Application Directory
    Application.exe
    Application.conf
    data/
        background.jpg
        button1.jpg
        button2.jpg
        info.dat
```

Within the Application.exe code, you can - completely system-agnostic - define a new resource that keeps track of all data items.

```
apppath = os.path.dirname(os.path.abspath(__file__))
appresources = Resources(os.path.join(apppath, "data"))
# Access some images
bgimage = appresources.get("background.jpg")
btn1image = appresources.get("button1.jpg")
...
```

To access individual files, you do not need to concat paths the whole time and regardless of the current directory, your application operates on, you can access your resource files at any time through the *Resources* instance, you created initially.

The *Resources* class is also able to scan an index archived files, compressed via ZIP or TAR (gzip or bzip2 compression), and subdiectories automatically.

```
Application Directory
    Application.exe
    Application.conf
    data/
```

```
        audio/
            example.wav
        background.jpg
        button1.jpg
        button2.jpg
        graphics.zip
            [tileset1.bmp
             tileset2.bmp
             tileset3.bmp
             ]
        info.dat

tilesimage = appresources.get("tileset1.bmp")
audiofile = appresources.get("example.wav")
```

If you request an indexed file via *Resources.get()*, you will receive a io.BytesIO stream, containing the file data, for further processing.

Note: The scanned files act as keys within the *Resources* class. This means that two files, that have the same name, but are located in different directories, will not be indexed. Only one of them will be accessible through the *Resources* class.

API

class sdl2.ext.**Resources**([*path=None*[, *subdir=None*[, *excludepattern=None*]]])

The Resources class manages a set of file resources and eases accessing them by using relative paths, scanning archives automatically and so on.

add(*filename : string*)

Adds a file to the resource container. Depending on the file type (determined by the file suffix or name) the file will be automatically scanned (if it is an archive) or checked for availability (if it is a stream or network resource).

add_archive(*filename : string*[, *typehint="zip"*])

Adds an archive file to the resource container. This will scan the passed archive and add its contents to the list of available and accessible resources.

add_file(*filename : string*)

Adds a file to the resource container. This will only add the passed file and do not scan an archive or check the file for availability.

get(*filename : string*) → BytesIO

Gets a specific file from the resource container.

Raises a KeyError, if the *filename* could not be found.

get_filelike(*filename : string*) → file object

Similar to *get()*, but tries to return the original file handle, if possible. If the found file is only available within an archive, a io.BytesIO instance will be returned.

Raises a KeyError, if the *filename* could not be found.

get_path(*filename : string*) → string

Gets the path of the passed *filename*. If *filename* is only available within an archive, a string in the form filename@archivename will be returned.

Raises a `KeyError`, if the *filename* could not be found.

scan(*path : string[, subdir=None[, excludepattern=None]*)
Scans a path and adds all found files to the resource container. If a file within the path is a supported archive (ZIP or TAR), its contents will be indexed aut added automatically.

The method will consider the directory part (`os.path.dirname`) of the provided *path* as path to scan, if the path is not a directory. If *subdir* is provided, it will be appended to the path and used as starting point for adding files to the resource container.

excludepattern can be a regular expression to skip directories, which match the pattern.

sdl2.ext.**open_tarfile**(*archive : string*, *filename : string*[, *directory=None*[, *ftype=None*]]) → BytesIO
Opens and reads a certain file from a TAR archive. The result is returned as `BytesIO` stream. *filename* can be a relative or absolute path within the TAR archive. The optional *directory* argument can be used to supply a relative directory path, under which *filename* will be searched.

ftype is used to supply additional compression information, in case the system cannot determine the compression type itself, and can be either **"gz"** for gzip compression or **"bz2"** for bzip2 compression.

If the filename could not be found or an error occurred on reading it, `None` will be returned.

Raises a `TypeError`, if *archive* is not a valid TAR archive or if *ftype* is not a valid value of ("gz", "bz2").

Note: If *ftype* is supplied, the compression mode will be enforced for opening and reading.

---

sdl2.ext.**open_url**(*filename : string*[, *basepath=None*]) → file object
Opens and reads a certain file from a web or remote location. This function utilizes the `urllib2` module for Python 2.7 and `urllib` for Python 3.x, which means that it is restricted to the types of remote locations supported by the module.

basepath can be used to supply an additional location prefix.

sdl2.ext.**open_zipfile**(*archive : string*, *filename : string*[, *directory : string*]) → BytesIO
Opens and reads a certain file from a ZIP archive. The result is returned as `BytesIO` stream. *filename* can be a relative or absolute path within the ZIP archive. The optional *directory* argument can be used to supply a relative directory path, under which *filename* will be searched.

If the filename could not be found, a `KeyError` will be raised. Raises a `TypeError`, if *archive* is not a valid ZIP archive.

Sprite, texture and pixel surface routines

sdl2.ext.**TEXTURE**
Indicates that texture-based rendering or sprite creation is wanted.

sdl2.ext.**SOFTWARE**
Indicates that software-based rendering or sprite creation is wanted.

class sdl2.ext.**Sprite**
A simple 2D object, implemented as abstract base class.

x
The top-left horizontal offset at which the *Sprite* resides.

y
The top-left vertical offset at which the *Sprite* resides.

position
The top-left position (*x* and *y*) as tuple.

size
The width and height of the *Sprite* as tuple.

Note: This is an abstract property and needs to be implemented by inheriting classes.

---

area
The rectangular area occupied by the *Sprite*.

depth
The layer depth on which to draw the *Sprite*. *Sprite* objects with higher *depth* values will be drawn on top of other *Sprite* values by the *SpriteRenderSystem*.

class sdl2.ext.**SoftwareSprite**
A simple, visible, pixel-based 2D object, implemented on top of SDL2 software surfaces.

surface
The sdl2.SDL_Surface containing the pixel data.

size
The size of the *SoftwareSprite* as tuple.

subsprite(*area : (int, int, int, int)*) → SoftwareSprite
Creates another *SoftwareSprite* from a part of the *SoftwareSprite*. The two sprites share pixel data, so if the parent sprite's surface is not managed by the sprite (free is False), you will need to keep it alive while the subsprite exists.

class sdl2.ext.**TextureSprite**
A simple, visible, pixel-based 2D object, implemented on top of SDL2 textures.

angle
The rotation angle for the *TextureSprite*.

center
The center to use for rotating the *TextureSprite*. *None* will reset the center to the default center of the *TextureSprite*.

flip
Allows the *TextureSprite* to be flipped over its horizontal or vertical axis via the appropriate SDL_FLIP_* value.

size
The size of the *TextureSprite* as tuple.

texture
The sdl2.SDL_Texture containing the texture data.

class sdl2.ext.**SpriteRenderSystem**
A rendering system for *Sprite* components. This is a base class for rendering systems capable of drawing and displaying *Sprite* based objects. Inheriting classes need to implement the rendering capability by overriding the render() method.

sortfunc
Sort function for the component processing order. The default sort order is based on the depth attribute of every sprite. Lower depth values will cause sprites to be drawn below sprites with higher depth values. If *sortfunc* shall be overridden, it must match the callback requirements for sorted().

process (*world : World*, *components : iterable*) → None
Renders the passed *Sprite* objects via the *render()* method. The *Sprite* objects are sorted via *sortfunc* before they are passed to *render()*.

render (*sprite : iterable*) → None
Renders the *Sprite* objects.

Note: This is a no-op function and needs to be implemented by inheriting classes.

---

class sdl2.ext.**SoftwareSpriteRenderSystem** (*window : object*)
A rendering system for *SoftwareSprite* components. The *SoftwareSpriteRenderSystem* class uses a sdl2.SDL_Window as drawing device to display *SoftwareSprite* surfaces. It uses the internal SDL surface of the *window* as drawing context, so that GL operations, such as texture handling or the usage of SDL renderers is not possible.

window can be either a *sdl2.ext.Window* or sdl2.SDL_Window instance.

window
The sdl2.SDL_Window that is used as drawing device.

surface
The sdl2.SDL_Surface that acts as drawing context for *window*.

render (*sprites : object*[, *x=None*[, *y=None*]]) → None
Draws the passed *sprites* on the *sdl2.ext.Window* surface. *x* and *y* are optional arguments that can be used as relative drawing location for *sprites*. If set to None, the location information of the *sprites* are used. If set and *sprites* is an iterable, such as a list of *SoftwareSprite* objects, *x* and *y* are relative location values that will be added to each individual sprite's position. If *sprites* is a single *SoftwareSprite*, *x* and *y* denote the absolute position of the *SoftwareSprite*, if set.

class sdl2.ext.**TextureSpriteRenderSystem** (*target : object*)
A rendering system for *TextureSprite* components. The *TextureSpriteRenderSystem* class uses a sdl2.SDL_Renderer as drawing device to display *Sprite* surfaces.

target can be a *sdl2.ext.Window*, sdl2.SDL_Window, a:class:*sdl2.ext.Renderer* or a sdl2.SDL_Renderer. If it is a *sdl2.ext.Window* or sdl2.SDL_Window instance, it will try to create a sdl2.SDL_Renderer with hardware acceleration for it.

sdlrenderer
The sdl2.SDL_Renderer that is used as drawing context.

rendertarget
The target for which the renderer was created, if any.

render (*sprites : object*[, *x=None*[, *y=None*]]) → None
Renders the passed *sprites* via the renderer. *x* and *y* are optional arguments that can be used as relative drawing location for *sprites*. If set to None, the location information of the *sprites* are used. If set and *sprites* is an iterable, such as a list of *TextureSprite* objects, *x* and *y* are relative location values that will be added to each individual sprite's position. If *sprites* is a single *TextureSprite*, *x* and *y* denote the absolute position of the *TextureSprite*, if set.

class sdl2.ext.**SpriteFactory** (*sprite_type=TEXTURE*, ***kwargs*)
A factory class for creating *Sprite* objects. The *SpriteFactory* can create *TextureSprite* or *SoftwareSprite* instances, depending on the *sprite_type* being passed to it, which can be SOFTWARE or TEXTURE. The additional *kwargs* are used as default arguments for creating sprites within the factory methods.

sprite_type
The sprite type created by the factory. This will be either SOFTWARE for *SoftwareSprite* or

TEXTURE for *TextureSprite* objects.

default_args
The default arguments to use for creating new sprites.

create_software_sprite(*size*, *bpp=32*, *masks=None*) → SoftwareSprite
Creates a software sprite. A *size* tuple containing the width and height of the sprite and a *bpp* value, indicating the bits per pixel to be used, need to be provided.

create_sprite(***kwargs*) → Sprite
Creates a *Sprite*. Depending on the *sprite_type*, this will return a *SoftwareSprite* or *TextureSprite*.

kwargs are the arguments to be passed for the sprite construction and can vary depending on the sprite type. Usually they have to follow the *create_software_sprite()* and *create_texture_sprite()* method signatures. *kwargs* however will be mixed with the set *default_args* so that one does not necessarily have to provide all arguments, if they are set within the *default_args*. If *kwargs* and *default_args* contain the same keys, the key-value pair of *kwargs* is chosen.

create_sprite_render_system(**args*, ***kwargs*) → SpriteRenderSystem
Creates a new *SpriteRenderSystem*, based on the set *sprite_type*. If *sprite_type* is TEXTURE, a *TextureSpriteRenderSystem* is created with the the `renderer` from the *default_args*. Other keyword arguments are ignored in that case.

Otherwise a *SoftwareSpriteRenderSystem* is created and *args* and *kwargs* are passed to it.

create_texture_sprite(*renderer : object*, *size*, *pformat=sdl2.SDL_PIXELFORMAT_RGBA8888*, *access=sdl2.SDL_TEXTUREACCESS_STATIC*) → TextureSprite
Creates a texture sprite. A *size* tuple containing the width and height of the sprite needs to be provided.

TextureSprite objects are assumed to be static by default, making it impossible to access their pixel buffer in favour for faster copy operations. If you need to update the pixel data frequently or want to use the texture as target for rendering operations, *access* can be set to the relevant SDL_TEXTUREACCESS_* flag.

from_color(*color : object*, *size*, *bpp=32*, *masks=None*) → Sprite
Creates a *Sprite* with a certain color.

from_image(*fname : str*) → Sprite
Creates a *Sprite* from an image file. The image must be loadable via *sdl2.ext.load_image()*.

from_object(*obj: object*) → Sprite
Creates a *Sprite* from an object. The object will be passed through `sdl2.rwops_from_object()` in order to try to load image data from it.

from_surface(*surface : SDL_Surface*[, *free=False*]) → Sprite
Creates a *Sprite* from the passed `sdl2.SDL_Surface`. If *free* is set to `True`, the passed *surface* will be freed automatically.

from_text(*text : str*[, ***kwargs*]) → Sprite
Creates a *Sprite* from a string of text. This method requires a *sdl2.ext.FontManager* to be in *kwargs* or *default_args*.

class `sdl2.ext.`**Renderer**(*target : obj*[, *logical_size=None*[, *index=-1*[, *flags=sdl2.SDL_RENDERER_ACCELERATED*]]])
A rendering context for windows and sprites that can use hardware or software-accelerated graphics drivers.

If target is a *sdl2.ext.Window* or `sdl2.SDL_Window`, *index* and *flags* are passed to the relevant `sdl2.SDL_CreateRenderer()` call. If *target* is a *SoftwareSprite* or `sdl2.SDL_Surface`, the *index* and *flags* arguments are ignored.

sdlrenderer
The underlying `sdl2.SDL_Renderer`.

rendertarget
The target for which the *Renderer* was created.

logical_size
The logical size of the renderer.

Setting this allows you to draw as if your renderer had this size, even though the target may be larger or smaller. When drawing, the renderer will automatically scale your contents to the target, creating letterboxing or sidebars if necessary.

To reset your logical size back to the target's, set it to `(0, 0)`.

Setting this to a lower value may be useful for low-resolution effects.

Setting this to a larger value may be useful for antialiasing.

color
The *sdl2.ext.Color* to use for draw and fill operations.

blendmode
The blend mode used for drawing operations (fill and line). This can be a value of

- `SDL_BLENDMODE_NONE` for no blending
- `SDL_BLENDMODE_BLEND` for alpha blending
- `SDL_BLENDMODE_ADD` for additive color blending
- `SDL_BLENDMODE_MOD` for multiplied color blending

scale
The horizontal and vertical drawing scale as two-value tuple.

clear(*[color=None]*)
Clears the rendering context with the currently set or passed *color*.

copy(*src : obj[, srcrect=None[, dstrect=None[, angle=0[, center=None[, flip=render.SDL_FLIP_NONE]]]]]*) → None
Copies (blits) the passed *src*, which can be a *TextureSprite* or `sdl2.SDL_Texture`, to the target of the *Renderer*. *srcrect* is the source rectangle to be used for clipping portions of *src*. *dstrect* is the destination rectangle. *angle* will cause the texture to be rotated around *center* by the given degrees. *flip* can be one of the SDL_FLIP_* constants and will flip the texture over its horizontal or vertical middle axis. If *src* is a *TextureSprite*, *angle*, *center* and *flip* will be set from *src*'s attributes, if not provided.

draw_line(*points : iterable[, color=None]*) → None
Draws one or multiple lines on the rendering context. If *line* consists of four values `(x1, y1, x2, y2)` only, a single line is drawn. If *line* contains more than four values, a series of connected lines is drawn.

draw_point(*points : iterable[, color=None]*) → None
Draws one or multiple points on the rendering context. The *points* argument contains the x and y values of the points as simple sequence in the form `(point1_x, point1_y, point2_x, point2_y, ...)`.

draw_rect(*rects : iterable[, color=None]*) → None
Draws one or multiple rectangles on the rendering context. *rects* contains sequences of four values denoting the x and y offset and width and height of each individual rectangle in the form `((x1, y1, w1, h1), (x2, y2, w2, h2), ...)`.

fill(*rects : iterable[, color=None]*) → None
Fills one or multiple rectangular areas on the rendering context with the current set or passed *color*. *rects*

contains sequences of four values denoting the x and y offset and width and height of each individual rectangle in the form `((x1, y1, w1, h1), (x2, y2, w2, h2), ...)`.

present() → None
Refreshes the rendering context, causing changes to the render buffers to be shown.

Software Surface manipulation

sdl2.ext.**subsurface**(*surface : SDL_Surface*, *area : (int, int, int, int)*) → SDL_Surface
Creates a surface from a part of another surface. The two surfaces share pixel data.

Note: The newly created surface *must not* be used after its parent has been freed!

Window routines to manage on-screen windows

class sdl2.ext.**Window**(*title : string*, *size : iterable*[, *position=None*[, *flags=None*]])
The Window class represents a visible on-screen object with an optional border and *title* text. It represents an area on the screen that can be accessed by the application for displaying graphics and receive and process user input.

The position to show the Window at is undefined by default, letting the operating system or window manager pick the best location. The behaviour can be adjusted through the `DEFAULTPOS` class variable.

```
Window.DEFAULTPOS = (10, 10)
```

The created Window is hidden by default, which can be overridden at the time of creation by providing other SDL window flags through the *flags* parameter. The default flags for creating Window instances can be adjusted through the `DEFAULTFLAGS` class variable.

```
Window.DEFAULTFLAGS = sdl2.SDL_WINDOW_SHOWN
```

window
The used `sdl2.SDL_Window`.

title
The title of the *Window*.

size
The size of the *Window*.

show() → None
Show the *Window* on the display.

hide() → None
Hide the *Window*.

maximize() → None
Maximizes the *Window* to the display's dimensions.

minimize() → None
Minimizes the *Window* to an iconified state in the system tray.

refresh() → None
Refreshes the entire *Window* surface.

Note: This only needs to be called, if a SDL_Surface was acquired via *get_surface()* and is used to display contents.

get_surface() → SDL_Surface
Gets the sdl2.SDL_Surface used by the *Window* to display 2D pixel data.

Note: Using this method will make the usage of GL operations, such as texture handling or the usage of SDL renderers impossible.

PySDL2 FAQ

This is a list of Frequently Asked Questions about PySDL2. If you think, something is missing, please suggest it!

On importing...

... my script fails and complains that a SDL2 library could not be found!

Do you have the libraries properly installed? Did you follow the operating system's way of installing or registering libraries? If you placed the libraries in some folder, make sure that the PYSDL2_DLL_PATH environment variable points to the correct location.

... my script fails complaining that the *found* SDL2 library can't be used!

Do you use a 64-bit operating system? Please make sure, that the Python interpreter *and* that the SDL2 libraries are either 64-bit ones *or* 32-bit ones. A 32-bit Python interpreter can't deal with a 64-bit library and vice versa.

Using...

... the sdl2 API is weird. Why do you use the SDL_ prefix all the time?

The low-level APIs for SDL2, SDL2_mixer, SDL2_ttf, ... shall represent a clean wrapping around the original C API calls. Thus, if you have to search for documentation or want to make a Python to C conversion (or C to Python), most of the code cleanly maps to the original API naming and layout and you do not have to think about whether you had to use SDL_ or TTF_ or whatever as prefix or suffix.

... the sdl2 API is does not comply to PEP-8. Please make it PEP-8 compatible.

Most of the API is PEP-8 compatible. The low-level bindings to SDL2 and related libraries however use the exact naming (including capital letters) as the functions or structures, they map to. See the previous entry for the reason of that.

How do I...

... save my surfaces as image files?

You can use sdl2.SDL_SaveBMP() to save them as bitmap files. Other formats are currently unsupported, but might be added to the *sdl2.ext* package in the future.

Font handling...

... is too hard. Why can't it work the same way as pygame does?

> The *sdl2.sdlttf* API does not know about platform-specific font locations and is unable to resolve font paths based on e.g. the font name or typeface. It's not its job and PySDL2 likewise does not provide such functionality. If you need improved font detection support, you might want to take a look at the sysfont module of the python-utils project, which can be found at https://bitbucket.org/marcusva/python-utils/.

Release News

This describes the latest changes between the PySDL2 releases.

0.9.5

Released on 2016-10-20.

- updated *sdl2* to include the latest changes of SDL2 (release 2.0.5)
- fixed issue #94: added support for TrueType font collection (TTC) files
- fixed issue #80: added flip and rotation support for TextureSprite objects
- renamed `sdl2.ext.Renderer.renderer` attribute to *sdl2.ext.Renderer.sdlrenderer*. The *renderer* attribute is deprecated and will be removed in a later version.

0.9.4

Released on 2016-07-07.

- updated *sdl2* to include the latest changes of SDL2 (release 2.0.4)
- updated *sdl2.sdlttf* to include the latest changes of SDL_ttf (release 2.0.14)
- new *sdl2.ext.Renderer.logical_size* attribute to set or retrieve the logical pixel size of a renderer
- fixed issue #48: be more noisy about DLL loading issues
- fixed issue #65: misleading documentation for *sdl2.ext.Renderer.draw_line()*
- fixed issue #67: Return a proper error code, when unittests running as subprocesses fail
- fixed issue #72: `sdl2.video.SDL_GL_DrawableSize()` not available on import
- fixed issue #76: define missing SDL_PRESSED and SDL_RELEASED constants
- fixed issue #82: examples/gui.py fails due to an attribute error
- fixed issue #83: fix compatibility with newer PIL versions in `sdl2.ext.image.load_image()`
- fixed issue #84: The setter of *sdl2.ext.Renderer.scale* works properly now
- fixed issue #85: fix environment-dependent unit tests
- fixed issue #87: fix incorrect MIX_INIT_* constants in *sdl2.sdlmixer*
- fixed issue #88: use PILs *Image.tobyte()instead of the deprecated 'Image.tostring()*
- fixed horizontical and vertical line drawing in *sdl2.ext.line()*

- fixed a bug in *sdl2.ext.Renderer.draw_line()* for odd numbers of points
- dropped IronPython support

0.9.3

Released on 2014-07-08.

- updated *sdl2* to include the latest changes of SDL2 (HG)
- new *sdl2.ext.Renderer.scale* attribute, which denotes the horizontal and vertical drawing scale
- new *sdl2.ext.point_on_line()* function to test, if a point lies on a line segment
- PYSDL2_DLL_PATH can contain multiple paths separated by `os.pathsep` to search for the libraries now
- *sdl2.ext.get_image_formats()* only returns BMP image support now, if SDL2_image and PIL are not found
- *sdl2.ext.load_image()* tries to use `sdl2.SDL_LoadBMP()` now, if SDL2_image and PIL are not found
- fixed issue #55: `sdl2.SDL_GameControllerAddMappingsFromFile()` does not raise a TypeError for Python 3.x anymore
- fixed issue #56: *sdl2.ext.Renderer.draw_line()* and *sdl2.ext.Renderer.draw_point()* handle multiple lines (or points) as arguments properly now
- fixed issue #57: if SDL2_image is not installed and PIL is used, the loaded pixel buffer of the image file is not referenced anymore after returning from *sdl2.ext.load_image()*, causing random segmentation faults
- fixed issue #58: raise a proper error, if *sdl2.ext.FontManager.render()* could not render a text surface
- fixed issue #59: The *sdl2.ext.TextureSpriteRenderSystem.sdlrenderer* attribute is correctly documented now
- fixed a local variable and module name collision in *sdl2.ext.FontManager.render()*

Thanks to Filip M. Nowak for the PYSDL2_DLL_PATH improvement.

0.9.2

Released on 2014-04-13.

- fixed issue #32: the line clipping algorithms do not run into precision errors anymore
- fixed issue #53 (again): `sdl2.video.SDL_GL_ResetAttributes()` is properly wrapped now to retain backwards compatibility with previous SDL2 releases
- fixed issue #54: text input is correctly converted for the text entry component
- updated the example BMP files, which could not be loaded properly on some systems with SDL2_image and PIL

0.9.1

Released on 2014-04-05.

- fixed issue #50: corrected the *sdl2.ext.load_image()* documentation

- fixed issue #52: *sdl2.ext.Renderer.fill()*, *sdl2.ext.Renderer.draw_rect()* and *sdl2.ext.Renderer.draw_point()* convert sequences correctly now
- fixed issue #53: provide backwards compatibility for previous SDL2 releases by adding a wrapper func for `sdl2.cpuinfo.SDL_HasAVX()`

0.9.0

Released on 2014-03-23.

IMPORTANT: This release breaks backwards-compatibility. See the notes for the issues #36 and #39.

- updated *sdl2* to include the latest changes of SDL2 (release 2.0.3)
- new *sdl2.ext.subsurface()* function to create subsurfaces from `sdl2.SDL_Surface` objects
- new *sdl2.ext.SoftwareSprite.subsprite()* method to create `sdl2.ext.SoftwarSprite` objects sharing pixel data
- the unit test runner features a *–logfile* argument now to safe the unit test output to a file
- issues #36, #39: the different render classes of sdl2.ext.sprite were renamed
  - the `sdl2.ext.RenderContext` class was renamed to *sdl2.ext.Renderer* to be consistent with with SDL2's naming scheme
  - `sdl2.ext.SpriteRenderer` was renamed to *sdl2.ext.SpriteRenderSystem*
  - `sdl2.ext.SoftwareSpriteRenderer` was renamed to *sdl2.ext.SoftwareSpriteRenderSystem*
  - `sdl2.ext.TextureSpriteRenderer` was renamed to *sdl2.ext.TextureSpriteRenderSystem*
  - `sdl2.ext.SpriteFactory.create_sprite_renderer()` was renamed to *sdl2.ext.SpriteFactory.create_sprite_render_system()*
- fixed `sdl2.audio.SDL_LoadWAV()` macro to provide the correct arguments
- fixed issue #44: use a slightly less confusing `ValueError`, if a renderer argument for the *sdl2.ext.SpriteFactory* is not provided
- fixed issue #43: improved the code reference for the improved bouncing section in the docs
- fixed issue #40: typo in a `RuntimeWarning` message on loading the SDL2 libraries
- fixed issue #38: the points arguments of `sdl2.ext.Renderer.draw_points()` are properly documented now
- fixed issue #37: `sdl2.SDL_GetRendererOutputSize()` is now acccessible via a wildcard import
- fixed issue #35: download location is now mentioned in the docs
- fixed issue #12: remove confusing try/except on import in the examples

0.8.0

Released on 2013-12-30.

- updated PD information to include the CC0 dedication, since giving software away is not enough anymore
- updated *sdl2* to include the latest changes of SDL2 (HG)
- fixed a wrong C mapping of `sdl2.rwops.SDL_FreeRW()`

- fixed various issues within the *sdl2.ext.BitmapFont* class
- issue #26: `sdl2.SDL_AudioSpec.callback` is a `SDL_AudioCallBack()` now
- issue #30: the SDL_Add/DelHintCallback() unittest works with PyPy now
- issue #31: `sdl2.sdlmixer.SDL_MIXER_VERSION()` returns the proper version now

Thanks to Sven Eckelmann, Marcel Rodrigues, Michael McCandless, Andreas Schiefer and Franz Schrober for providing fixes and improvements.

0.7.0

Released on 2013-10-27.

- updated *sdl2* to include the latest changes of SDL2 (release 2.0.1)
- fixed a bug in *sdl2.ext.FontManager.render()*, which did not apply the text color correctly
- issue #14: improved the error messages on failing DLL imports
- issue #19: the `sdl2.ext.TextureSpriteRenderer.render()` and `sdl2.ext.SoftwareSpriteRenderer.render()` methods do not misinterpret x and y arguments anymore, if set to 0
- issue #21: *sdl2.ext.load_image()* raises a proper `UnsupportedError`, if neither SDL_image nor PIL are usable

Thanks to Marcel Rodrigues, Roger Flores and otus for providing fixes and improvement ideas.

0.6.0

Released on 2013-09-01.

- new *sdl2.ext.FontManager.size* attribute, which gives a default size to be used for adding fonts or rendering text
- updated *sdl2* to include the latest changes of SDL2
- `sdl2.ext.RenderContext.copy()` accepts any 4-value sequence as source or destination rectangle now
- issue #11: throw an `ImportError` instead of a `RuntimeError`, if a third-party DLL could not be imported properly
- fixed a bug in the installation code, which caused `sdl2.examples` not to install the required resources

Thanks to Steven Johnson for his enhancements to the FontManager class. Thanks to Marcel Rodrigues for the improvements to RenderContext.copy().

0.5.0

Released on 2013-08-14.

- new *sdl2.ext.FontManager* class, which provides simple TTF font rendering.
- new *sdl2.ext.SpriteFactory.from_text()* method, which creates text sprites
- put the SDL2 dll path at the beginning of PATH, if a PYSDL2_DLL_PATH is provided to avoid loading issues for third party DLLs on Win32 platforms
- minor documentation fixes

Thanks to Dan Gillett for providing the FontManager and from_text() enhancements and his patience regarding all the small change requests. Thanks to Mihail Latyshov for providing fixes to the documentation.

0.4.1

Released on 2013-07-26.

- updated *sdl2* to include the latest changes of SDL2
- improved DLL detection for DLLs not being in a library path
- fixed a bug in `sdl2.ext.RenderContext.draw_rect()` for drawing a single rect
- fixed a bug in the `repr()` call for *sdl2.ext.SoftwareSprite*
- issue #4: fixed a bug in `sdl2.ext.RenderContext.fill()` for filling a single rect
- issue #5: fixed pip installation support
- issue #6: fixed a bug in *sdl2.ext.get_events()*, which did not handle more than 10 events in the queue correctly
- issue #8: *sdl2.ext.SpriteFactory.create_texture_sprite()* can create sprites to be used as rendering targets now
- issue #9: improved error messages on trying to bind non-existent library functions via ctypes
- minor documentation fixes

Thanks to Steven Johnson, Todd Rovito, Bil Bas and Dan McCombs for providing fixes and improvements.

0.4.0

Released on 2013-06-08.

- new *sdl2.sdlmixer* module, which provides access to the SDL2_mixer library
- issue #1: fixed libc loading for cases where libc.so is a ld script
- updated *sdl2* and *sdl2.sdlimage* to include the latest changes of the libraries, they wrap

0.3.0

Released on 2013-05-07.

- new *sdl2.sdlgfx* module, which provides access to the SDL2_gfx library
- new *sdl2.ext.UIFactory.from_color* method; it creates UI-supportive sprites from a color
- fixed color argument bugs in `sdl2.ext.RenderContext` methods
- fixed a module namespace issues in `sdl2.ext.pixelaccess`
- *sdl2.ext.SpriteFactory* methods do not use a default `size` argument anymore; it has to provided by the caller

0.2.0

Released on 2013-05-03.

- removed sdl2.ext.scene; it now lives in python-utils
- fixed `sdl2.haptic` module usage for Python 3
- fixed `sdl2.SDL_WindowGetData()` and `sdl2.SDL_WindowSetData()` wrappers
- fixed `sdl2.ext.RenderContext.copy()`
- fixed *sdl2.ext.line()*
- *sdl2* imports all submodules now
- improved documentation

0.1.0

Released on 2013-04-23.

- Initial Release

Further readings:

Todo list for PySDL2

General

- more unit tests

Windows

- Add support for SDL_SetWindowsMessageHook()
- Add SDL_TOUCH_MOUSEID constant

License

```
This software is distributed under the Public Domain. Since it is
not enough anymore to tell people: 'hey, just do with it whatever
you like to do', you can consider this software being distributed
under the CC0 Public Domain Dedication
(http://creativecommons.org/publicdomain/zero/1.0/legalcode.txt).

In cases, where the law prohibits the recognition of Public Domain
software, this software can be licensed under the zlib license as
stated below:

Copyright (C) 2012-2014 Marcus von Appen <marcus@sysfault.org>

This software is provided 'as-is', without any express or implied
```

```
warranty. In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgement in the product documentation would be
   appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.
```

Some files are **not** covered by the license above.

- For `examples/resources/tuffy.ttf`, `examples/resources/tuffy.copy.ttf` and `sdl2/test/resources/tuffy.ttf` the following terms apply:

  ```
  We, the copyright holders of this work, hereby release it into the
  public domain. This applies worldwide.

  In case this is not legally possible,

  We grant any entity the right to use this work for any purpose, without
  any conditions, unless such conditions are required by law.

  Thatcher Ulrich <tu@tulrich.com> http://tulrich.com
  Karoly Barta bartakarcsi@gmail.com
  Michael Evans http://www.evertype.com
  ```

- doc/python.inv

  An inventory index for linking to the proper places in the Python documentation, taken from http://docs.python.org/2/. Its copyright and license information can be found at http://docs.python.org/2/copyright.html and http://docs.python.org/2/license.html.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

CHAPTER 3

Documentation TODOs

Todo

More details, examples, etc.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pysdl2/checkouts/rel_0_9_5/doc/tutorial/pygamers.rs
line 11.)

Last generated on: Jul 19, 2017

Python Module Index

S

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

X

Y