Introduction to Pygame-ce

2D Game Programming

Ralf Adams

Version 1.2.1 (February 14, 2026)

Contents

1 Goals

In this script, you will learn how to program simple 2D games using the programming language Python and the game library Pygame-ce.

The main goal is not to create a perfect or finished game. Instead, this script focuses on helping you understand the basic ideas and principles behind game programming.

You will learn, step by step,

- how a simple game is structured,
- how graphics are drawn on the screen,
- how bitmaps are drawn on the screen,
- how to move game elements,
- how to use the classes `Sprite` and `Group`,
- how keyboard and mouse input work,
- how to produce text outputs by fonts and bitmaps,
- how sounds and music can be used,
- how system events and user defined events work,
- how game objects like figures or obstacles interact i.e. collision detection,
- how to implement time based logic,
- and many small details about pygame-ce.

I will also present some programming techniques in the chapter *Techniques* that you may find useful. This chapter is still fairly thin, but it already contains an introduction to the topics: animation, tile-based graphics, and how to handle very large game worlds. Additional techniques – such as a 3D-style visual effect for passing landscapes – are currently being developed.

In the final chapter, I introduce a few smaller games in order to demonstrate the concrete application of these techniques: the classic example *Pong*, a bubble sticking one, and the *Moonlander*.

You can find all source code and resources on `github.com/adamsralf/pygame_book` and will be updated regularly. Each section kicks off with a link to the relevant GitHub subdirectory.

What is *not* part of this script:

- camera, controller, touch pad, joystick as input devices
- clipboard support
- test module
- freetype font
- interacting with other languages like C/C++
- other platforms like phone, web browser, etc.
- client-server communication
- midi sound
- direct usage of SDL

One thing I'm really not good at is creating visually appealing game worlds. And if I'm being honest, I've always cared far more about programming than about game design. So if you're looking for a deep dive into everything from sketchbooks and graphics tools to a polished final game, you'll be better off turning to other authors.

This script is especially designed for beginners. You do not need any previous experience with game programming. Basic knowledge of Python is required.

Many examples are kept short and simple. You are encouraged to **try things out, experiment, and change the code**. Making mistakes is part of learning — and often the best way to understand how things work. At the end of this script, you should be able to create your own small 2D games and continue learning on your own.

It is up to you which development environment you use; in this script, I use VS Code.

This script is based on the Pygame fork *Pygame Community Edition* (`Pygame-ce`). The source code examples are **not** checked for compatibility with the original Pygame. To keep things simple and easier to read, I will usually just say *Pygame* and will not make a distinction between the two versions.

Support my work

If you enjoyed this book and found it helpful, you're welcome to support my work with a small voluntary contribution. Writing, testing, and explaining things takes time – and occasionally coffee.
If you feel like buying me one (or helping fund the next version of this script), you can do so via PayPal: *adamsralf@outlook.de*.
Of course, this is entirely optional – but very much appreciated. Thank you for reading.

If you have any suggestions or feedback, feel free to get in touch: `adamsralf@outlook.de`

Have fun programming and creating your first games!
Ralf Adams

2 Basics

2.0 Installation and Organization

2.0.1 Installing Python and Pygame

2.0.1.1 Python

So . . . where are the files?

https://www.python.org/downloads/

I will not provide a detailed step-by-step installation guide for Python here. For that, it is best to consult the up-to-date instructions on the official Python website (see above). So, if you do not yet have Python installed, visit the homepage, download the current installation files, and run the installer.

The exact procedure differs depending on whether you are working on Windows, Linux, macOS, Android, or even with Docker images.

For the current edition of this book, I used Python 3.12.150 on Windows 10.0.26200. Python was installed using the official Python stand alone installer.

Python options

- During installation, you can choose whether Python should be added to the system search path (environment variable) or not. It is is highly recommended to do so, as it allows you to start a Python program simply with `python program.py`. Otherwise, you would always have to specify the full path to the Python executable.
- You will also be asked whether Python should be installed just for you or for all users. My recommendation is to always install it for yourself only. This allows other users to use different Python versions, and any customization remain local. It also makes it much harder to break things ;-)
- If you do not need multiple Python versions, consider removing older installations – this avoids a lot of unnecessary confusion.

2.0.1.2 Pygame Community Edition

So . . . where are the files?

- https://pyga.me/
- https://github.com/adamsralf/pygame_book/blob/main/src/00%20Introduction/00%20Setup/start.py

The installation itself is very simple. Before we continue, however, we should make sure that the original Pygame (not Pygame-ce) is not already installed:

```
pip uninstall pygame
WARNING: Skipping pygame as it is not installed.
```

If you receive this response, everything is fine. Otherwise, you will be asked whether all Pygame files should be removed, which you can safely confirm with Yes.

After that, install Pygame-ce using:

```
pip install pygame-ce
Collecting pygame-ce
Using cached pygame_ce-2.5.6-cp312-cp312-win_amd64.whl.metadata (12 kB)
Using cached pygame_ce-2.5.6-cp312-cp312-win_amd64.whl (10.4 MB)
Installing collected packages: pygame-ce
Successfully installed pygame-ce-2.5.6
```

And that's it – everything is in place :-)

You might also see the following output:

```
pip install pygame-ce
Requirement already satisfied: pygame-ce in
    .\AppData\Local\Programs\Python\Python312\Lib\site-packages (2.5.6)
```

In this case, everything is fine as well. You should only check the version number. If it is not the desired or the most recent one, simply uninstall Pygame-ce and then install it again.

After Python and Pygame have been installed, you should run a quick installation check. To do so, download a minimal program and try to start it.

Listing 2.1: Pygame installation test

```python
import pygame  # Pygame-Modul (also for pygame-ce!)


def main():
    pygame.init()
    window = pygame.Window(size=(400, 100))
    window.title = "Pygame test"
    screen = window.get_surface()

    running = True
```

```
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
        screen.fill("white")
        window.flip()

    pygame.quit()


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

If everything is configured correctly, you should see a window like the one in figure 2.1.

Figure 2.1: Pygame installation test

Error reason number 1

This is by far the most common problem I see with my students: The editor in Visual Studio Code, the word `pygame` is underlined in the line `import pygame`, accompanied by an error message claiming that the module cannot be found.
In the vast majority of cases, this has nothing to do with Pygame itself. The real reason is almost always that multiple Python installations exist on the system.
Pygame is installed into one specific Python installation, usually in the subdirectory
`.\AppData\Local\Programs\Python\Python`*xyz*`\Lib\site-packages`
If the development environment happens to use a different Python interpreter, it will simply not find the Pygame files – even though they are actually installed.

2.0.2 A Recommended Project Structure

At the beginning, it is tempting to put everything into a single file called `main.py`. This works – for about five minutes.

As soon as your game grows beyond a few hundred lines, structure becomes important. A good directory layout makes your code easier to understand, easier to extend, and much easier to debug.

Here is a simple and proven project structure that we will *loosely* follow throughout this book.

Do not worry if this feels like overkill right now. It will make sense very soon. But what goes where? Let us briefly go through the most important parts.

```
project_name/
├── main.py
├── config.py
├── assets/
│   ├── images/
│   ├── sounds/
│   ├── fonts/
│   └── ...
├── scenes/
│   ├── menu.py
│   ├── game.py
│   ├── pause.py
│   └── ...
├── sprites/
│   ├── player.py
│   ├── enemies.py
│   └── ...
├── utils/
│   ├── helpers.py
│   └── ...
└── ...
```

`main.py` This is the entry point of your game. It starts Pygame, creates the main window, and runs the main loop. Ideally, `main.py` does not contain a lot of game logic. Think of it as the conductor, not the orchestra.

`assets/` description Everything that is not Python code goes here: images, sounds, fonts. Keeping assets separate from code avoids clutter and makes it easier to move or reuse them later.

`scenes/` Scenes (or states) represent different phases of the game: main menu, actual gameplay, pause screen, game over screen, etc. Separating scenes into individual files keeps each part manageable and avoids huge, unreadable files.

`sprites/` This directory contains sprite classes such as the player, enemies, or other game objects. If an object moves, animates, or collides with something, it usually belongs here.

`utils/` Helper functions, small tools, or reusable code that does not belong to a specific scene or sprite can live here.

Not every project needs this directory – but many projects end up needing it sooner or later. In this book i. e. I skip the level `assets/`.

A note on file paths

One important habit: Always use relative paths, and always load assets relative to your project directory. Hard-coded absolute paths like `C:\Users\...` or `/home/...` will break as soon as you move the project to another computer. We will look at clean and safe ways to handle paths later in the book (see source code 2.85 on page 115).

If you have reached this point and Pygame starts without errors, the test window opens and your project directory looks reasonably clean, then you are perfectly prepared for the chapters ahead.

From here on, we can finally focus on what this book is really about: writing games instead of fighting your setup.

2.1 Kind of *Hello World!*

So . . . where are the files?

- https://github.com/adamsralf/pygame_book/blob/main/src/00%20Introduction/01%20TheBeginning/v01
- https://github.com/adamsralf/pygame_book/blob/main/src/00%20Introduction/01%20TheBeginning/v02

2.1.1 The Very First Steps

Listing 2.2: My first *Game*, Version 1.0

```python
1  import pygame  # Pygame-Modul (also for pygame-ce!)
2
3
4  def main():
5      pygame.init()                                  # Start subsystem
6      window = pygame.Window(size=(600, 400))        # Create Window
7      window.title = "My first Pygame program"       # Set window title
8      window.position = (10, 50)                     # Set window position
9      screen = window.get_surface()                  # Get the window's bitmap surface
10
11     running = True
12     while running:                                 # Main program loop: start
13         for event in pygame.event.get():           # Retrieve events
14             if event.type == pygame.QUIT:          # Window X clicked?
15                 running = False
16         screen.fill((0, 255, 0))                   # Fill playground
17         window.flip()                              # Swap double buffer
18
19     pygame.quit()                                  # Shut down subsystem
20
21
22 if __name__ == "__main__":
23     main()
```

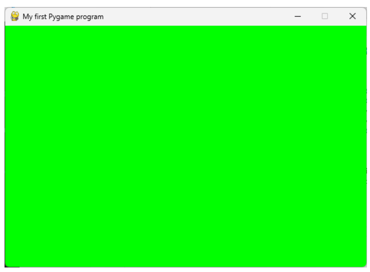

Fig. 2.2: Playground

When you start the application now, you will see a nicely designed window with a green background (figure 2.2). At the moment, however, not much is happening. The only thing you can do is close the window by clicking on the *X* button in the upper right corner of the window frame.

In order to use Pygame, the module `pygame` must be imported into the program (line 1). This makes the constants, functions, events, and classes of the namespace available.

Pygame is not just about calling functions or creating objects; a whole subsystems must be initialized explicitly. In this example, this is done using the static function init()

`pygame.init()`. Pygame is now connected to parts of the Operating System (OS) system, to deliver and receive required information and actions. In line 5 the Pygame engine is started by calling `init()` . It ist also possible to start only parts of the engine e.g. the sound subsystem by `pygame.mixer.init()`.

Window

get_surface()

For our games, we need a *playfield*/a window in which everything takes place. The class `pygame.Window` represents such a playfield. In line 6, the constructor receives one argument – namely the width and the height of the window as the 2-tuple `size`. Our window is therefore 600 px wide and 400 px high (see Pixel (px)). The method `get_surface()` in line 9 returns a `pygame.Surface` object , which is roughly something like a bitmap.

screen

title

In line 9, I store this return value in the variable named `screen`. I can then assign a title to the window using the attribute `Window.title` (see line 7) and set the position of the window relative to the desktop using the attribute `Window.position` (see line 8).

main loop

The game itself – just like all future games – runs inside a main loop. The loop starts in line 12 and ends in line 19.

Inside this loop, three things will always happen in the future:

- Reading and processing events: As shown in line 13f., mouse, keyboard, or game events are detected and passed on to the game elements. In our case, only clicking the X in the upper right corner of the window is registered.
- Updating the state of the game elements: Based on the events detected above and the current states of the game elements, the new states are determined (the player moves, a projectile bounces off, points increase, etc.). In our case, only the flag `running` of the main program loop is set to `False`.
- Drawing the bitmaps of the game elements: The game elements have a new position or a new appearance and must therefore be redrawn. In this minimal example, only the background of the playfield is colored in line 16, and afterwards the double buffer is swapped using `Window.flip()` in line 17.

By calling `pygame.init()`, Pygame places a kind of listener inside the operating system. More precisely, Pygame listens to the *message queue*. This is where the operating system collects all messages that are triggered by events. These can include USB connection messages, SSD error messages, mouse actions, program starts or crashes, and many others.

event.get()

Pygame now retrieves from the message queue, using `pygame.event.get()`, all events that could be relevant to the game. Using a `for`-loop, I then iterate over these events starting at line 13 and pick out the ones that are relevant to me.

event.type

QUIT

First, I check which type of event (`pygame.event.type`) is being offered. At the moment, only the type `pygame.QUIT` is important to me. This type is triggered when the operating system sends a *quit* message to the application. If I receive such a message, I set the flag `running` to `False`, so that the main program loop is terminated.

If I do not receive this signal, the main program loop continues to run happily and fills the entire playfield with a color in line 16 using `screen.fill()` – in this case, green. Please note that, similar to line 6, the function expects one argument – namely a 3-tuple. This 3-tuple encodes the color using RGB values between 0 and 255. Predefined color names such as *green* can also be used here.

RGB

color names

What remains is line 17: Here, the function `pygame.quit()` is called. This function is essentially the opposite of `pygame.init()` in line 5. All reserved resources are released again, and the Pygame listeners are removed from the system. You should always make sure to call this function at the end of your application; do not simply terminate the game. The difference is similar to just running out of your apartment versus properly turning off the lights and locking the door when leaving.

quit()

If we take a look at the game in the task manager (see figure 2.3), we might be a bit surprised: around 30 % of the CPU time is being used by this *IAmActuallyDoingNothing* game.

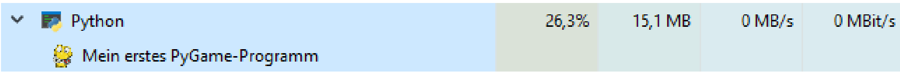

Figure 2.3: Resource usage without timing control

However, if we take a closer look at the main program loop, this should not really be surprising. A bitmap is being drawn onto the screen without any limitation and without interruption. It would be better to allow enough time in each loop iteration to collect events, calculate the new states, and only then generate the screen output. The screen output itself should also not happen arbitrarily fast or too often; in general, about 60 Frames Per Second (FPS) are sufficient for motion to be perceived as smooth.

fps

Listing 2.3: My first *Game*, Version 1.1

```python
import pygame


def main():
    pygame.init()
    window = pygame.Window(size=(600, 400),
        title="My first Pygame program",           # via function parameter
        position=(10, 50))
    screen = window.get_surface()
    clock = pygame.time.Clock()                    # Clock object

    running = True
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
        screen.fill((0, 255, 0))
        window.flip()
        clock.tick(60)                             # Limit framerate to 60 fps

```

```
21     pygame.quit()
22
23
24 if __name__ == "__main__":
25     main()
```

Clock

tick()

In line 10, a `pygame.time.Clock` object is created for timing control. With the help of this object, various time-related tasks can be handled; for the moment, however, we only need it for timing in line 19. There, `pygame.time.Clock.tick()` is called with a frame rate measured in fps. This function ensures that the application now runs at a maximum of 60 fps. This can be seen in the significantly reduced CPU usage shown in figure 2.4.

Hint

The Pygame documentation points out that the function `tick()` is very resource-efficient, but somewhat imprecise. If accuracy is important for timing, the function `tick__loop()` is recommended instead. Its disadvantage, however, is that it consumes significantly more processing time than `tick()`.

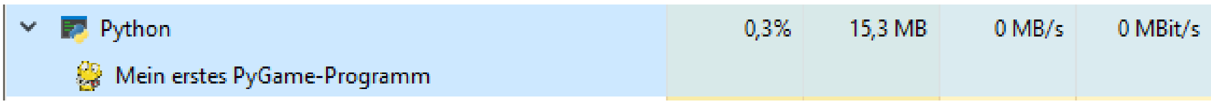

Figure 2.4: Resource usage with timing control

2.1.2 More Input

2.1.2.1 Multiple Windows

So . . . where are the files?

https://github.com/adamsralf/pygame_book/blob/main/src/00%20Introduction/01%20TheBeginning/v03/start.py

You can also create multiple windows for a game (see https://pyga.me/docs/ref/window.html).

Listing 2.4: Multiple Windows

```python
def main():
    pygame.init()
    window_first = pygame.Window(size=(300, 50),
        title="Main Window",
        position=(500, 50))
    window_second = pygame.Window(size=(300, 50),
        title="Side Window",
        position=(820, 50))
    screen_first = window_first.get_surface()
    screen_second = window_second.get_surface()
    clock = pygame.time.Clock()


    running = True
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
            elif event.type == pygame.WINDOWCLOSE:
                running = False
                event.window.destroy()
        if running:
            screen_first.fill((0, 255, 0))
            window_first.flip()
            screen_second.fill((255, 0, 0))
            window_second.flip()
            clock.tick(60)

    pygame.quit()
```

Figure 2.5: Multiple Windows

2.1.2.2 Information About the Graphics Environment

So . . . where are the files?

https://github.com/adamsralf/pygame_book/blob/main/src/00%20Introduction/01%20TheBeginning/v04/start.py

Info()

Sometimes it is necessary to know information about the graphics environment: perhaps to identify performance problems, or perhaps to find out which display features are available. Using the function `pygame.display.Info()`, various parameters can be queried. On my system, calling this function produced the output shown in figure 2.6 on the next page.

Please refer to table 2.1 for the meaning of the values (source: https://pyga.me/docs/ref/display.html#pygame.display.Info).

Table 2.1: Fields of `pygame.display.Info()`

Field	**Description**
`hw`	1 if the display is hardware accelerated.
`wm`	1 if windowed display modes can be used.
`video_mem`	The amount of video memory on the display in megabytes. This value is `0` if the amount is unknown.
`bitsize`	Number of bits used to store each pixel.
`bytesize`	Number of bytes used to store each pixel.
`masks`	Four values used to pack RGBA values into pixels.
`shifts`	Four values used to pack RGBA values into pixels.
`losses`	Four values used to pack RGBA values into pixels.
`blit_hw`	1 if hardware Surface blitting is accelerated.
`blit_hw_CC`	1 if hardware Surface colorkey blitting is accelerated.
`blit_hw_A`	1 if hardware Surface pixel alpha blitting is accelerated.
`blit_sw`	1 if software Surface blitting is accelerated.
`blit_sw_CC`	1 if software Surface colorkey blitting is accelerated.
`blit_sw_A`	1 if software Surface pixel alpha blitting is accelerated.
`current_w`, `current_h`	Width and height of the current video mode, or of the desktop mode if called before `display.set_mode()`. The values are `-1` on error.
`pixel_format`	The pixel format of the display surface as a string, for example `PIXELFORMAT_RGB888`.

Listing 2.5: `pygame.display.Info()`

```
4 def main():
5     pygame.init()
6     screen=pygame.display.set_mode([200, 200])
7     info = pygame.display.Info() #pygame.display.get_wm_info()
8     pygame.display.message_box("Window System Information", repr(info), "info")
9     pygame.quit()
```

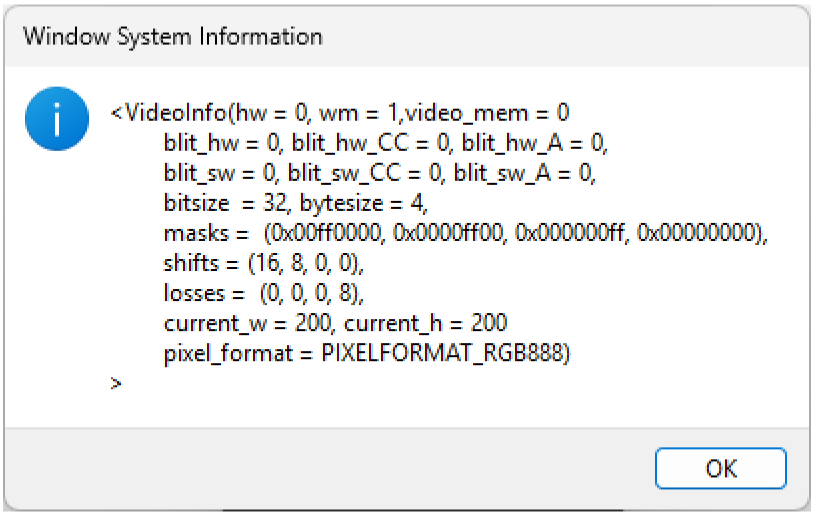

Figure 2.6: Infos about the graphical environment

2.1.2.3 Using `pygame.display` instead of `pygame.Window`

So … where are the files?

https://github.com/adamsralf/pygame_book/blob/main/src/00%20Introduction/01%20TheBeginning/v04/start.py

Based on feedback from the community, it became clear that there is a need to also show how to create a window using `pygame.display`. This seems to cause issues especially on Linux systems running Wayland.

Creating the window happens like this (or something very close to it) in line 6. Using `pygame.display.set_mode()`, you pass the window size in pixels as a pair of numbers – here 400 px × 600 px. The function returns a `Surface` object, which we can then use to draw everything we need (or want) onto the screen. set_mode()

Number pair!

A very common beginner mistake is passing the size as two separate numbers instead of a number pair. So don't forget the parentheses – even if they look a bit odd at first!

Listing 2.6: Window by `pygame.display`

```
1  import pygame
2
3
4  def main():
5      pygame.init()
6      screen = pygame.display.set_mode((400,600))     # Create window
7      pygame.display.set_caption("Window by Display Module") # Set window title
8
9      running = True
10     while running:
11         for event in pygame.event.get():
12             if event.type == pygame.QUIT:
13                 running = False
14
15         screen.fill("white")                        # Fill background with white
16         pygame.display.flip()                       # Update display
17
18     pygame.quit()
19
20
21 if __name__ == "__main__":
22     main()
```

The function `set_mode()` can take a couple of additional parameters. For example, you can decide whether the window should be resizable (see table 2.2 on page 26) or specify which color depth to use.

set_caption()

flip()

The window title is set in line 7 using `pygame.display.set_caption()`. Finally, `pygame.display.flip()` pushes the complete contents of `screen` to the display in one go.

2.1.3 What was new?

To start a minimal Pygame application

- Import the Pygame library.
- Initialize the Pygame system.
- Create a window / a playfield.
- Set up a main program loop:
  1. Poll events.

2. Update game objects.
3. Render the screen contents.
4. Control the timing of the loop iterations.

- Shut down the Pygame system when exiting.

The following Pygame elements were introduced

- `import pygame`:
  https://pyga.me/docs/tutorials/en/import-init.html
- `pygame.display.flip()`:
  https://pyga.me/docs/ref/display.html#pygame.display.flip
- `pygame.display.set_caption()`:
  https://pyga.me/docs/ref/display.html#pygame.display.set_caption
- `pygame.display.set_mode()`:
  https://pyga.me/docs/ref/display.html#pygame.display.set_mode
- `pygame.init()`:
  https://pyga.me/docs/ref/pygame.html#pygame.init
- `pygame.quit()`:
  https://pyga.me/docs/ref/pygame.html#pygame.quit
- `pygame.QUIT`:
  https://pyga.me/docs/ref/event.html#pygame.event.EventType.type
- `pygame.WINDOWCLOSE`:
  https://pyga.me/docs/ref/event.html#pygame.event.EventType.type
- `pygame.event.get()`:
  https://pyga.me/docs/ref/event.html#pygame.event.get
- `pygame.event.type`:
  https://pyga.me/docs/ref/event.html#pygame.event.EventType.type
- `pygame.time.Clock`:
  https://pyga.me/docs/ref/time.html#pygame.time.Clock
- `pygame.time.Clock.tick()`:
  https://pyga.me/docs/ref/time.html#pygame.time.Clock.tick

- `pygame.time.Clock.tick_busy_loop()`: https://pyga.me/docs/ref/time.html#pygame.time.Clock.tick_busy_loop
- `pygame.Surface.fill()`: https://pyga.me/docs/ref/surface.html#pygame.Surface.fill
- `pygame.Window`: https://pyga.me/docs/ref/window.html
- `pygame.Window.destroy()`: https://pyga.me/docs/ref/window.html#pygame.Window.destroy
- `pygame.Window.flip()`: https://pyga.me/docs/ref/window.html#pygame.Window.flip
- `pygame.Window.get_surface()`: https://pyga.me/docs/ref/window.html#pygame.Window.get_surface
- `pygame.Window.title`: https://pyga.me/docs/ref/window.html#pygame.Window.title
- `pygame.Window.position`: https://pyga.me/docs/ref/window.html#pygame.Window.position

Table 2.2: Predefined Display Modes

Constant	Description
`FULLSCREEN`	create a fullscreen display
`DOUBLEBUF`	only applicable with OPENGL
`HWSURFACE`	(obsolete in pygame 2) hardware accelerated, only in FULLSCREEN
`OPENGL`	create an OpenGL-renderable display
`RESIZABLE`	display window should be sizeable
`NOFRAME`	display window will have no border or controls
`SCALED`	resolution depends on desktop size and scale graphics
`SHOWN`	window is opened in visible mode (default)
`HIDDEN`	window is opened in hidden mode

2.1.4 Homework

Please have a look at `https://pyga.me/docs/ref/window.html` and then try to solve the following small exercises:

1. Set up a working environment for your game programming. Install Python, your

preferred editor, and the latest Pygame-ce version. Open the directory containing your Pygame source code and try to run `start01.py`.

2. Change the background color of the window. Use RGB values as well as named colors.
3. Change the size of the window.
4. Change the position of the window. Use position values and also try `WINDOWPOS_CENTERED` and `WINDOWPOS_UNDEFINED` .
5. Create the window as a resizable window and try to resize it.
6. Define a minimum an maximum window size, show the actual size in the title bar, and try to resize the window to its limits.
7. Show the actual window position in the title bar.
8. Play a little bit with the `opacity` attribute of the window.
9. Change the title of the window during runtime according to a counter. Shut down the program if counter is greater 600.
10. Create a borderless window.
11. Try a fullscreen window.
12. Try to arrange three windows in a row. Compute the x-position of the second and third window based on the window size and a useful gap between them.

WINDOWPOS_CENTERED

WINDOWPOS_UNDEFINED

2.2 Graphic Primitives

2.2.1 Introduction

So ... where are the files?

https://github.com/adamsralf/pygame_book/blob/main/src/00%20Introduction/02%20Primitives/primitives00.py

Graphic primitives are simple graphical shapes that are drawn, such as lines, points, circles, and so on. They do not play a very important role in game programming, because most graphics are predefined bitmaps, but they can be quite useful. For this reason, I will only introduce a few of them here.

Listing 2.7: Graphic Primitives

```python
import pygame
import pygame.gfxdraw  # !


def main():
    pygame.init()
    window = pygame.Window(size=(530, 530), title = "Grafic Primitives", position = (10, 50))
    screen = window.get_surface()
    clock = pygame.time.Clock()

    mygrey = pygame.Color(200, 200, 200)                    # Custom color
    myrectangle1 = pygame.Rect(10, 10, 20, 30)              # Rectangle object
    myrectangle2 = pygame.Rect(60, 10, 20, 30)
    points1 = ((120, 10), (160, 10), (140, 90))             # List of points
    points2 = ((180, 10), (220, 10), (200, 90))

    running = True
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
        screen.fill(mygrey)
        pygame.draw.rect(screen, "red", myrectangle1)                 # Filled rectangle
        pygame.draw.rect(screen, "red", myrectangle2, 3, 5)           # Rectangle outline
        pygame.draw.polygon(screen, "green", points1)                 # Filled polygon
        pygame.draw.polygon(screen, "green", points2, 1)              # Polygon outline
        pygame.draw.line(screen, "red", (5, 230), (240, 230), 3)      # Line
        pygame.draw.circle(screen, "blue", (40, 150), 30)             # Filled circle
        pygame.draw.circle(screen, "blue", (110, 150), 30, 2)         # Circle outline
        pygame.draw.circle(screen, "blue", (180, 150), 30, 5, True)   # Arc segment
        for i in range(255):
            for j in range(255):
                screen.set_at((265+i, 10+j), (255, i, j))                 # Points variant 1
                screen.fill((i, j, 255), ((10+i, 265+j), (1, 1)) )        # Variant 2
                pygame.gfxdraw.pixel(screen, 265+i, 265+j, (i, 255, j))   # Variant 3

        window.flip()
        clock.tick(60)

    pygame.quit()
```

The basic structure is the same as in source code 2.3 on page 19. The differences begin in line 11. The class `pygame.Color` can encode color information in various formats, including an alpha channel (alpha blending, transparency); more about this will follow later in section 2.3 on page 45. Here, I use RGB encoding with color channel values between 0 and 255. Color

In most cases, however, I do not need to define my own colors. Pygame provides a really extensive list of 664 predefined color names . Wherever color values are expected, I can pass either a `Color` object, a numeric color code, or a color name as a string. Color names

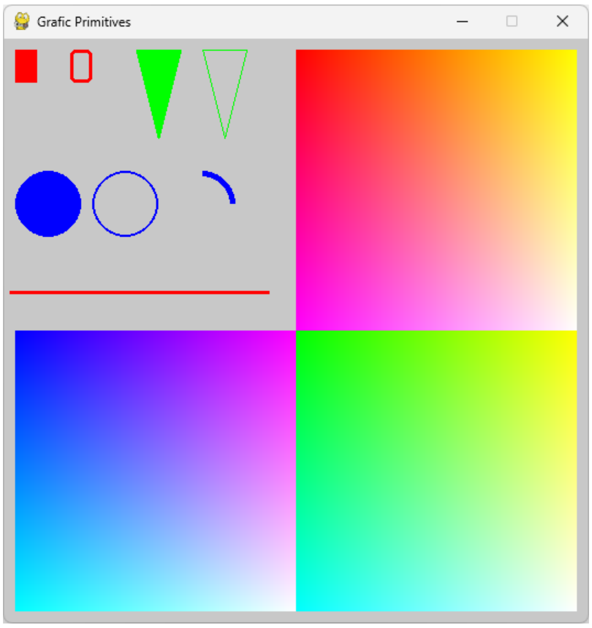

Fig. 2.7: Some graphic primitives

Let us go through the individual shapes one by one and start with the rectangle. There are several ways to define a rectangle in Pygame. Since we will need it very often later on, I would like to introduce the class `pygame.Rect` here. It is defined by four parameters: the upper-left corner, its width, and its height. In line 12, a rectangle is therefore defined at the position $(10, 10)$ with a width of 20 px and a height of 30 px. Rect

In line 23, `pygame.draw.rect()` draws a filled rectangle. The Semantics of the parameters should be self-explanatory. The call in line 24, however, is different. The first parameter after the rectangle – here `3` – specifies the thickness of the line. If this parameter is given and greater than 0, the rectangle is no longer filled. The value `10` specifies the rounding of the corners. Here, a value between 0 and $min(width, height)/2$ can be used, as this value corresponds to the radius of the corner rounding. rect()

Hint

The class `Rect` is not a drawn rectangle, but merely a container for information that is relevant for a rectangle. More in chapter 2.4.1 on page 59.

More general than a rectangle is a Polygon. A polygon is a closed chain of lines that is defined in Pygame by its points. Similar to rectangles, there are filled (line 25) and unfilled (line 26) variants. Both are drawn using `pygame.draw.polygon()`. polygon()

Warning

Be careful with the line thickness: the lines grow outward, which can quickly lead to ugly offsets at the corners. Try it out by changing the value `2` to `5`.

line()

lines()

For individual lines, there is `pygame.draw.line()`, and for a polyline – without an example here – there is `pygame.draw.lines()`. An example can be found in line 27.

circle()

A circle is defined by two values: its center point and its radius. In line 28, a filled circle with the center at $(40, 150)$ and a radius of 30 px is drawn using `pygame.draw.circle()`. As with rectangles and polygons, there are also unfilled variants (line 29). Of particular interest is the circular arc segment in line 30. Here, Boolean variables are used to control which section of the circular arc is drawn (for more details, see the Pygame reference).

Finally, one small color experiment. Strangely enough, Pygame does not provide a dedicated function for drawing a single point or pixel. Here, I have implemented three workarounds that I found. One could think of additional ones as well: a line with $start = end$, a circle with a radius of 1 px, and so on.

set_at()

pixel()

In line 33, a point is drawn by setting a single color value at a specific position using `pygame.Surface.set_at()`. Alternatively, the `fill()` surface function used earlier can be applied with an area of only one pixel in width and height (line 34). Another way to set a pixel using a graphics library is the experimental `gfxdraw` module. In line 35, a single pixel is set using `pygame.gfxdraw.pixel()`. The `gfxdraw` module is not imported automatically by `import pygame` (see line 2).

2.2.2 More Input

2.2.2.1 Example: Particle Swarm

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/02%20Primitives/circles

Using graphic primitives, it is possible to create dynamic effects, such as particle swarms. Here, I would like to present a very simple example of a mouse-controlled fountain made of circles.

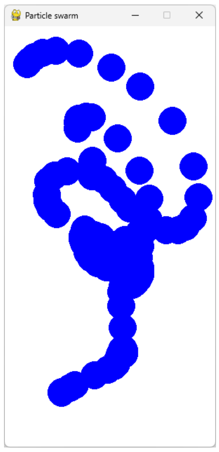

Fig. 2.8: Not a particle swarm

Let us first build a small program that draws a circle at the mouse position. The class `Circle` (see line 4) contains all the information I need to draw circles: position, radius, and color. The position is defined via a constructor argument. In the method `draw()`, the screen output is encapsulated.

The function `main()` now contains a lot of familiar elements, but also a few new ones. In line 4, the screen size is stored in a list, because we still need this information at another place, namely in line 18. Below that, in line 24, a list for storing the circles is defined.

Inside the main loop, line 32 checks whether the left mouse button has been pressed. If so, a circle is drawn at the mouse position. After that, the screen is filled with white color and the circles stored in the container are drawn.

get_pressed()

get_pos()

The result is not very impressive yet (see figure 2.8 on the preceding page) and is more reminiscent of a drawing program like Paint.

Listing 2.8: Particle swarm, Version 1.0

```
1  import pygame
2
3
4  class Circle:                                        # Very helpful
5      def __init__(self, pos) -> None:
6          self.posx = pos[0]
7          self.posy = pos[1]
8          self.radius = 20
9          self.color = "blue"
10
11     def draw(self, screen: pygame.Surface) -> None:
12         pygame.draw.circle(screen, self.color, (self.posx, self.posy), self.radius)
13
14
15 def main():
16     size = (300, 600)                                # Screen size
17     pygame.init()
18     window = pygame.Window( size=size,               # Create window
19                             title = "Particle swarm",
20                             position = (10, 50))
21     screen = window.get_surface()
22
23     clock = pygame.time.Clock()
24     circles = []                                     # Container for circles
25
26     running = True
27     while running:
28         for event in pygame.event.get():
29             if event.type == pygame.QUIT:
30                 running = False
31
32         if pygame.mouse.get_pressed()[0]:            # Left mouse button?
33             circles.append(Circle(pygame.mouse.get_pos()))
34
35         screen.fill("white")
36         for p in circles:
37             p.draw(screen)
38
39         window.flip()
40         clock.tick(60)
41
42     pygame.quit()
43
44
45 if __name__ == '__main__':
46     main()
```

In the next step, we want to turn the bulky circles into colorful particles. These particles should also no longer appear exactly at the mouse position, but be scattered around it. To achieve this, only minimal changes need to be made to the `Circle` class.

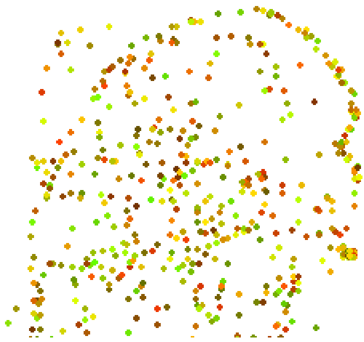

Fig. 2.9: Particle swarm Version 2

The two position values are now extended by a random value between -2 and $+2$. The radius is also reduced to 2 px. The color is likewise varied using random values. I tried out several combinations here, and I quite like this color variation. Feel free to experiment with the color channels and the random values yourself. The result shown in figure 2.9 already looks much better.

Listing 2.9: Particle swarm, Version 2.0

```
    def __init__(self, pos) -> None:
        self.posx = pos[0] + randint(-2, 2)
        self.posy = pos[1] + randint(-2, 2)
        self.radius = 2
        self.color = [randint(100, 255), randint(50, 255), 0]
```

Now we want to add a bit of dynamics to the game. The particles should first rise upward and then fall down again. To achieve this, I added the vertical velocity `speedy` to the `Circle` class and assigned it a random initial value (line 14). The division by 10.1 ensures that no smooth, rounded values are created. Here as well, feel free to experiment with the values to see the different effects.

Gravity

The class also needs to be extended by the method `update()`. In this method, the new vertical position `posy` is calculated based on the vertical velocity `speedy`, and the velocity is in turn modified with respect to gravity. In order for all particles to be subject to the same gravitational force, I defined `GRAVITY` as a static attribute (line 7).

Listing 2.10: Particle swarm, Version 3.0, Class `Circle`

```
6  class Circle:
7      GRAVITY = 0.3                          # Gravity as a static attribute
8
9      def __init__(self, pos) -> None:
10         self.posx = pos[0] + randint(-2, 2)
11         self.posy = pos[1] + randint(-2, 2)
12         self.radius = 2
13         self.color = [randint(100, 255), randint(50, 255), 0]
14         self.speedy = randint(-100, 0) / 10.01      # Initial vertical speed
15
16     def update(self) -> None:
17         self.speedy += Circle.GRAVITY
18         self.posy += self.speedy
```

All that remains is the call of `update()` inside the main program loop.

Listing 2.11: Particle swarm, Version 3.0, Call of `update()`

```
        if pygame.mouse.get_pressed()[0]:
            circles.append(Circle(pygame.mouse.get_pos()))

        for p in circles:
            p.update()
```

The fountain is still not really lively yet. So let us also scatter the particles horizontally. For this purpose, the attribute `speedx` is added in the constructor. The upper and lower bounds of the random number generator determine the width of the particle fountain. Try out values here that match your own sense of aesthetics. In `update()`, the new horizontal position `posx` then has to be calculated.

The horizontal velocity does not need to be adjusted, since `GRAVITY` is only supposed to act downward.

Listing 2.12: Particle swarm, Version 4.0, `Circle.update()`

```
9       def __init__(self, pos) -> None:
10          self.posx = pos[0] + randint(-2, 2)
11          self.posy = pos[1] + randint(-2, 2)
12          self.radius = 2
13          self.color = [randint(100, 255), randint(50, 255), 0]
14          self.speedx = randint(-10, 10) / 10.01
15          self.speedy = randint(-100, 0) / 10.01
16
17      def update(self) -> None:
18          self.speedy += Circle.GRAVITY
19          self.posx += self.speedx
20          self.posy += self.speedy
```

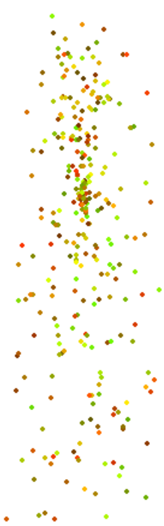

Fig. 2.10: Particle swarm, Version 5: nearly finished

After some time, the list `circles` contains many particles that are no longer displayed at all. We want to remove these particles. To do this, the `Circle` class needs to determine whether the `object` can be deleted.

As a first step, we add the deletion flag `todelete` to the class (see line 16), which is initialized to `False`; a new particle should of course not be deleted immediately.

In line 22, it is checked whether the right edge of the particle (center point plus radius) lies outside the screen on the left. If this is the case, the deletion flag must be set to `True`. Analogously, the right and the bottom edges of the screen are checked in line 24 and line 26.

For this purpose, the attribute `pygame.Window.size` is used to determine the width and height of the screen. This attribute returns the screen size as a 2-tuple. The zeroth value represents the width, and the first value represents the height. A check to see whether the

size

particle has disappeared upward is not necessary, since it will eventually fall down again and thus become visible once more.

Listing 2.13: Particle swarm, Version 5.0, Class `Circle`

```python
class Circle:
    GRAVITY = 0.3

    def __init__(self, pos) -> None:
        self.posx = pos[0] + randint(-2, 2)
        self.posy = pos[1] + randint(-2, 2)
        self.radius = 2
        self.color = [randint(100, 255), randint(50, 255), 0]
        self.speedx = randint(-10, 10) / 10.01
        self.speedy = randint(-100, 0) / 10.01
        self.todelete = False                       # Delete flag

    def update(self, window: pygame.Window) -> None:
        self.speedy += Circle.GRAVITY
        self.posx += self.speedx
        self.posy += self.speedy
        if self.posx - self.radius < 0:             # Left side out
            self.todelete = True
        elif self.posx + self.radius > window.size[0]:  # Right side out
            self.todelete = True
        elif self.posy - self.radius > window.size[1]:  # Bottom out
            self.todelete = True

    def draw(self, screen: pygame.Surface) -> None:
        pygame.draw.circle(screen, self.color, (self.posx, self.posy), self.radius)
```

In the main program, I now need to implement suitable deletion logic. But first, I want my fountain to have a bit more *oomph*: In line 48, not just one particle is created, but always five at once.

In line 51, an empty list is created that will contain the particles to be deleted. Inside the update loop, it is now additionally checked whether a particle should be deleted (line 54). If so, this particle is added to the list `todelete`. After the update loop has finished, the particles to be deleted are removed from the list `circles` starting at line 56.

In figure 2.10 on the previous page, you can see a fountain. It really starts to look cool only when you move the mouse while it is running.

Listing 2.14: Particle swarm, Version 5.0, Main loop

```python
41     running = True
42     while running:
43         for event in pygame.event.get():
44             if event.type == pygame.QUIT:
45                 running = False
46 
47         if pygame.mouse.get_pressed()[0]:
48             for i in range(5):                      # 5 particles at once
49                 circles.append(Circle(pygame.mouse.get_pos()))
50 
51         todelete = []                               # Temporary storag
52         for p in circles:
53             p.update(window)
54             if p.todelete:                          # Marked for deletion?
```

```python
                todelete.append(p)
        for p in todelete:                          # Delete
            circles.remove(p)

        screen.fill("white")
        for p in circles:
            p.draw(screen)

        window.flip()
        clock.tick(60)
```

Why do I not call `remove()` already inside the update loop?

Because: *Never increase or decrease the length of a list while you are iterating over it.* Very strange effects can occur.

Try to guess the number of loop iterations of the following program:

```python
        values = [1, 2, 3]
        for a in values:
                values.append(a*10)
        print(vlues)
```

Even small changes to the parameters can already produce interesting visual effects. Unfortunately, these cannot be shown very well using images here, so: program it yourself and try it out.

Listing 2.15: Particle swarm, Version 6.0

```python
class Circle:
    GRAVITY = 0.3
    RADIUS_INC = -0.1                                     # Radius increment

    def __init__(self, pos) -> None:
        self.posx = pos[0] + randint(-4, 4)               # Changed spread
        self.posy = pos[1] + randint(-4, 4)
        self.radius = 8
        self.color = [randint(100, 255), randint(50, 255), 0]
        self.speedx = randint(-15, 15) / 10.01            # Fountain wider
        self.speedy = randint(-100, 0) / 10.01
        self.todelete = False

    def update(self, window: pygame.Window) -> None:
        self.speedy -= Circle.GRAVITY
        self.posx += self.speedx
        self.posy += self.speedy
        self.radius += Circle.RADIUS_INC
        if self.posx - self.radius < 0:
            self.todelete = True
        elif self.posx + self.radius > window.size[0]:
            self.todelete = True
        elif self.posy - self.radius > window.size[1]:
            self.todelete = True
        elif self.radius <= 0.0:                          # Can be removed
            self.todelete = True

    def draw(self, screen: pygame.Surface) -> None:
        pygame.draw.circle(screen, self.color, (self.posx, self.posy), self.radius)

```

```python

def main():
    size = (300, 600)
    pygame.init()
    window = pygame.Window( size=size, title = "Particle swarm", position = (10, 50))
    screen = window.get_surface()
    clock = pygame.time.Clock()
    circles = []

    running = True
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False

        if pygame.mouse.get_pressed()[0]:
            for i in range(5):
                circles.append(Circle(pygame.mouse.get_pos()))

        todelete = []
        for p in circles:
            p.update(window)
            if p.todelete:
                todelete.append(p)
        for p in todelete:
            circles.remove(p)

        screen.fill("white")
        for p in circles:
            p.draw(screen)

        window.flip()
        clock.tick(60)

    pygame.quit()
```

2.2.2.2 Example: Landscape

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/02%20Primitives/landscape

In this example, we combine graphic primitives, object-oriented design, and simple mathematics to create a small animated scene.

I would like to create a small animated landscape that should look like in figure 2.11 on the facing page. The sun should rise on the left, move across the sky, and set on the right. The blue color of the sky should also change depending on the time of day.

Let us start with a basic framework that I want to expand step by step. The elements of the source code 2.16 on the next page should be self-explanatory. The variable `horizon` is meant to control the boundary between the sky and the meadow – in other words, it forms the horizon. The sky, sun, tree, house, and meadow should all align with this boundary.

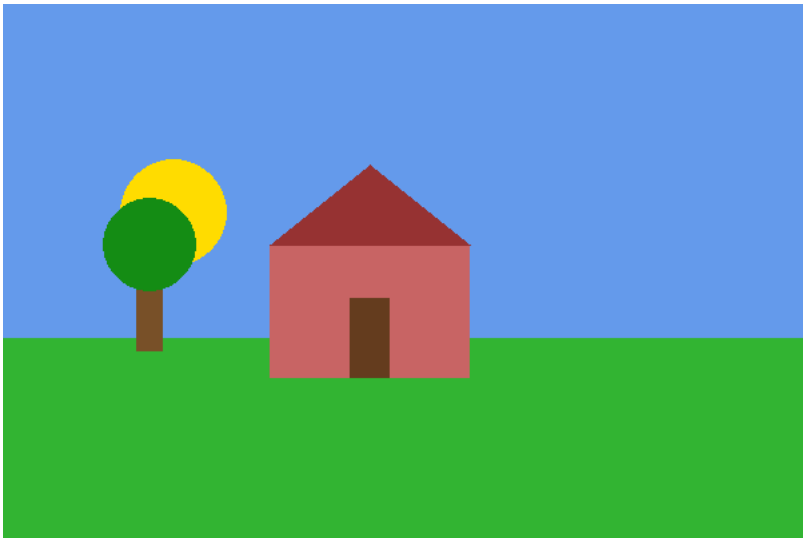

Figure 2.11: Example: Drawing a Landscape

Listing 2.16: Landscape, Version 1.0

```python
import pygame


def main():
    size = (600, 400)
    pygame.init()
    window = pygame.Window( size=size, title = "A Peaceful Day")
    clock = pygame.time.Clock()
    horizon = 250


    running = True
    while running:
        # Watch for events
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False

        # Updates

        # Draw
        window.flip()
        clock.tick(60)

    pygame.quit()


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

I extend the program with the very simple class `Meadow`. In the constructor, a reference to the window and the horizon are stored, and the color is defined — in this case, a custom shade of green. After that, the upper-left corner and the size of the meadow are calculated. Both values take the horizon into account. The method `draw()` then draws the meadow as a green rectangle into the window.

Listing 2.17: Landscape, Version 2.0, Class `Meadow`

```
class Meadow:
    def __init__(self, window: pygame.window.Window, horizon: int) -> None:
        self.window = window
        self.horizon = horizon
        self.color = (50, 180, 50)
        self.lefttop = 0, self.horizon
        self.widthheight= self.window.size[0], self.window.size[1] - self.horizon

    def draw(self) -> None:
        screen = self.window.get_surface()
        pygame.draw.rect(screen, self.color, (self.lefttop, self.widthheight))
```

In `main()`, an object of the class `Meadow` is now created in line 23, and in line 36 the meadow is drawn using the `draw()` method. The result looks like the one shown in figure 2.12 on the facing page.

Listing 2.18: Landscape, Version 2.0, `main()`

```
22     horizon = 250
23     meadow = Meadow(window, horizon)                #
24
25
26     running = True
27     while running:
28         # Watch for events
29         for event in pygame.event.get():
30             if event.type == pygame.QUIT:
31                 running = False
32
33         # Updates
34
35         # Draw
36         meadow.draw()                               #
37         window.flip()
```

The class `Sky` is similarly simple. Here as well, reference data is stored in the constructor, and in `draw()` a blue rectangle is drawn whose size depends on `horizon`.

Listing 2.19: Landscape, Version 3.0, Class `Sky`

```
class Sky:
    def __init__(self, window: pygame.window.Window, horizon: int) -> None:
        self.window = window
        self.horizon = horizon
        self.color = (100, 150, 255)

    def draw(self) -> None:
        screen = self.window.get_surface()
        pygame.draw.rect(screen, self.color, (0, 0, self.window.size[0], self.horizon))
```

All that remains is to integrate it into `main()` in the same way as `Meadow` (see figure 2.13 on the next page). Play around a bit with the variable `horizon` to see the effect.

Listing 2.20: Landscape, Version 3.0, `main()`

```
    horizon = 250
    meadow = Meadow(window, horizon)
    sky = Sky(window, horizon)                        #


    running = True
    while running:
        # Watch for events
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False

        # Updates

        # Draw
        meadow.draw()
        sky.draw()                                    #
        window.flip()
```

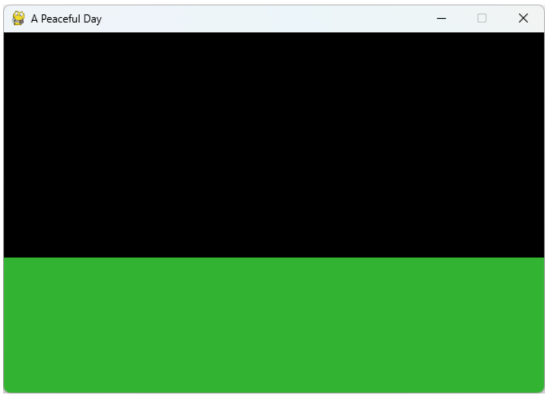

Figure 2.12: Drawing a Landscape (2)

Figure 2.13: Drawing a Landscape (3)

The class `Tree` consists of two parts: a tree trunk and a leafy crown. The tree trunk is created in `draw()` using a rectangle, and the leafy crown is created using a circle. I will not show the integration into `main()` here, since it is completely analogous to the integration of `Meadow` and `Sky`. Only the order needs to be considered, because the tree is supposed to appear in the foreground. The tree should look like the one shown in figure 2.14 on the following page.

Order matters

One of the most common beginner mistakes is drawing graphic objects in the wrong order. When that happens, objects often end up being partly or completely covered, and you start looking for the bug in the wrong place.

Listing 2.21: Landscape, Version 4.0, Class `Tree`

```
class Tree:
    def __init__(self, window: pygame.window.Window, horizon: int) -> None:
        self.window = window
        self.colors = [(120, 80, 40), (20, 140, 20)]
        self.start = (100, horizon - 50)

    def draw(self) -> None:
        screen = self.window.get_surface()
        pygame.draw.rect(screen, self.colors[0], (self.start, (20, 60)))
        pygame.draw.circle(screen, self.colors[1], (self.start[0]+10, self.start[1]-20), 35)
```

The basic principle of the class `House` is the same as for the other classes. It is just a bit more complex, since it consists of two rectangles and a triangle. Here as well, the integration into `main()` is trivial and is left to you. In figure 2.15, only the sun is missing now.

Listing 2.22: Landscape, Version 5.0, Class `House`

```
class House:
    def __init__(self, window: pygame.window.Window, horizon: int) -> None:
        self.window = window
        self.colors = [(200, 100, 100), (150, 50, 50), (100, 60, 30)]
        self.start = (200, horizon - 70)

    def draw(self) -> None:
        screen = self.window.get_surface()
        pygame.draw.rect(screen, self.colors[0], (self.start, (150, 100)))
        pygame.draw.polygon(
            screen,
            self.colors[1],
            [self.start, (self.start[0]+75, self.start[1]-60), (self.start[0]+150,
                self.start[1])]
        )
        pygame.draw.rect(screen, self.colors[2], (self.start[0]+60, self.start[1]+40, 30,
            60))
```

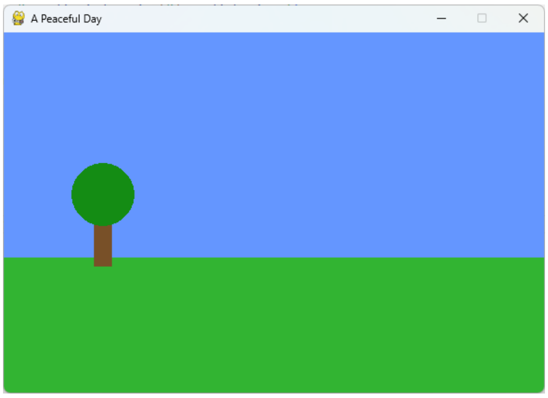

Figure 2.14: Drawing a Landscape (4)

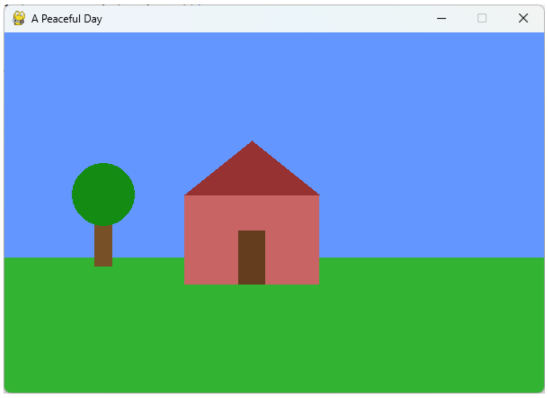

Figure 2.15: Drawing a Landscape (5)

In its basic shape, the sun is a simple yellow filled circle. However, we want it to move across the sky. Therefore, we need a start position below the horizon (see line 63) and a method `update()` that calculates the new position of the sun in each frame.

In `update()`, the new horizontal position is first calculated based on `speed`. After that, I calculate how far the sun has already progressed along its path. This value is relative and has a range of $[0, 1]$. If the sun has covered a quarter of the distance, the value of `progress` is 0.25, at halfway it is 0.5, and so on.

How do I calculate the height now? For the sake of simplicity, I let the sun follow the first half of the sine function. For this, the domain must be $[0, \pi]$; this is the hump of the sine function that lies above the x-axis. The range of the sine function from 0 to π is $[0, 1]$. If I multiply this value by the horizon, I obtain values from 0 to `horizon`. Finally, I add the radius so that the sun just touches the upper edge at its highest point.

The function `update()` returns the value of `progress` so that I can reuse this value to calculate the color of the sky, which still needs to be implemented. Everything clear? By the way, do not forget to add `import math` at the beginning because of the sine function!

Listing 2.23: Landscape, Version 6.0, Class `Sun`

```
class Sun:
    def __init__(self, window: pygame.window.Window, horizon: int) -> None:
        self.speed = 1
        self.radius = 40
        self.pos = [-self.radius, 0]                #
        self.color = (255, 220, 0)
        self.horizon = horizon
        self.window = window

    def update(self) -> float:
        self.pos[0] += self.speed
        progress = self.pos[0] / self.window.size[0]  # 0.0 -> 1.0
        self.pos[1] = self.horizon * (1 - math.sin(progress * math.pi)) + self.radius
        return progress

    def draw(self) -> None:
        screen = self.window.get_surface()
        pygame.draw.circle(screen, self.color, self.pos, self.radius)
```

Like the other classes, the sun is integrated into `main()`: the object is created in line 89 and drawn using `draw()` in line 104. Only the call to `update()` in line 100 is new. Important: The order of the `draw()` calls must be observed! The sun should be drawn after the sky, but before the meadow and the tree.

Listing 2.24: Landscape, Version 6.0, `main()`

```
85      meadow = Meadow(window, horizon)
86      sky = Sky(window, horizon)
87      tree = Tree(window, horizon)
88      house = House(window, horizon)
89      sun = Sun(window, horizon)                  #
90
91
92      running = True
93      while running:
94          # Watch for events
95          for event in pygame.event.get():
96              if event.type == pygame.QUIT:
```

```
                running = False

        # Updates
        sun.update()                            #

        # Draw
        sky.draw()
        sun.draw()                              #
        meadow.draw()
        tree.draw()
        house.draw()
        window.flip()
```

The final stage of the extension concerns the color of the sky. Depending on the position of the sun – more precisely, on the progress of the sun – the blue color of the sky should change. This is done in the new method `update()` of the class `Sky` (see source code 2.25). Here as well, the green component of the color is calculated using the sine function; a linear approach would also have been possible, but the sine function produces smoother transitions near sunrise and sunset.

Listing 2.25: Landscape, Version 7.0, `Sky.update()`

```
    def update(self, progress: float) -> None:
        brightness = max(0, min(1, math.sin(progress * math.pi)))
        blue = int(80 + brightness * 120)
        self.color = (100, blue, 235)
```

In `main()`, only the relative progress of the sun is now taken and passed to the `update()` method of `Sky`. Done :-)

Listing 2.26: Landscape, Version 7.0, `main()`

```
        # Updates
        progress = sun.update()
        sky.update(progress)
```

2.2.3 What was new?

Using graphic primitives, you can create and use your own drawings. They are usually available in both filled and unfilled variants. Colors can either be defined manually or selected from a list of predefined colors.
Objects drawn later appear in front of earlier ones.

The following Pygame elements were introduced

- Named colors:
  https://pyga.me/docs/ref/color_list.html
- `import pygame.gfxdraw`:
  https://pyga.me/docs/ref/gfxdraw.html
- `pygame.Color`:
  https://pyga.me/docs/ref/color.html
- `pygame.draw.circle()`:
  https://pyga.me/docs/ref/draw.html#pygame.draw.circle
- `pygame.draw.line()`:
  https://pyga.me/docs/ref/draw.html#pygame.draw.line
- `pygame.draw.lines()`:
  https://pyga.me/docs/ref/draw.html#pygame.draw.lines
- `pygame.draw.polygon()`:
  https://pyga.me/docs/ref/draw.html#pygame.draw.polygon
- `pygame.draw.rect()`:
  https://pyga.me/docs/ref/draw.html#pygame.draw.rect
- `pygame.gfxdraw.pixel()`:
  https://pyga.me/docs/ref/gfxdraw.html#pygame.gfxdraw.pixel
- `pygame.mouse.get_pos()`:
  https://pyga.me/docs/ref/mouse.html#pygame.mouse.get_pos
- `pygame.mouse.get_pressed()`:
  https://pyga.me/docs/ref/mouse.html#pygame.mouse.get_pressed
- `pygame.Rect`:
  https://pyga.me/docs/ref/rect.html

- `pygame.Surface.set_at()`: https://pyga.me/docs/ref/surface.html#pygame.Surface.set_at
- `pygame.Window.size`: https://pyga.me/docs/ref/window.html#pygame.Window.size

2.2.4 Homework

Please have a look at `https://pyga.me/docs/ref/draw.html` and then try to solve the following small exercises:

1. Program the following: Randomly choose a point and a radius. Using these values, draw a circle with a random, semi-transparent color. As an additional challenge, make sure that the circle may touch the edge of the window at most, but must not go beyond it.
2. Create a window with a color gradient from blue in the upper-left corner to red in the lower-left corner. Then draw two white filled circles with the same radius. One circle should be created using texttdraw.circle() and the other using `draw.aacircle()`. Compare the results.

aacircle()

3. Draw 10 random lines in a window using `draw.aaline()`. Each line should start at the edge of the window and end at the edge. Then apply `draw.flood_fill()` to the center of the window and observe the effect.

aaline()

flood_fill()

4. Try to draw the Moonlander like in figure 2.16 using only the functions in `pygame.draw`. Of course, you may also choose any other non-trivial object instead.

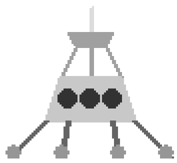

Figure 2.16: Drawing a Moonlander

2.3 Load and Blit Bitmaps

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/03%20Bitmaps

2.3.1 Introduction

config.py

In Python, it is common practice to move program settings, global variables, and similar configuration data into a file named `config.py`.

Listing 2.27: Load and blit bitmaps: config.py (1)

```
WINDOW_WIDTH = 600
WINDOW_HEIGHT = 400
FPS = 60
```

The file is then imported and usually given a shorter name, so its contents can be used across the whole project.

Listing 2.28: Load and blit bitmaps, Version 1.0

```
import config as cfg


def main():
    pygame.init()
    window = pygame.Window(
        size=(cfg.WINDOW_WIDTH, cfg.WINDOW_HEIGHT),
        title="Load and Draw of Bitmaps",
        position=(10, 50))
    screen = window.get_surface()
    clock = pygame.time.Clock()

    defender_image = pygame.image.load("images/defender01.png")  # Load bitmap
    enemy_image = pygame.image.load("images/alienbig0101.png")

    running = True
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False

        screen.fill("white")
        screen.blit(enemy_image, (10, 10))          # Draw bitmap
        screen.blit(defender_image, (10, 80))
        window.flip()
```

```
28          clock.tick(cfg.FPS)
29
30      pygame.quit()
```

In source code 2.28, two bitmaps – in this case two Portable Network Graphics (PNG) files – are loaded and displayed on the screen.

load()

blit()

Loading is done using the function `pygame.image.load()`. In line 15f., the bitmaps – also called sprites – are loaded and converted into a `Surface` object. In line 25 the two bitmaps are then printed onto the `screen` surface without any further processing using `pygame.Surface.blit()`. The first parameter of `blit()` is the `Surface` object that is to be drawn, followed by the position. Here, the horizontal (x) coordinate is specified first, and then the vertical (y) coordinate. You can *admire* the result in figure 2.17.

The coordinate system's origin

Unlike in school mathematics, the origin is not at the lower left, but at the upper left.

Figure 2.17: Load and blit bitmaps, Version 1.0

We now want to adapt the bitmaps a bit to better suit our needs. First, the documentation recommends converting the bitmap into a format that is easier for Pygame to process after loading. In addition, I want to adjust the size ratios of the two bitmaps, because the enemy appears too large compared to the defender.

Listing 2.29: Load and blit bitmaps, Version 1.1

```
15     defender_image = pygame.image.load("images/defender01.png").convert() # to display format
16     defender_image = pygame.transform.scale(defender_image, (30, 30))     # Scale surface
17
18     enemy_image = pygame.image.load("images/alienbig0101.png").convert()
19     enemy_image = pygame.transform.scale(enemy_image, (50, 45))
```

The function `pygame.Surface.load()` returned a `Surface` object. The `Surface` class now provides a method that performs the desired conversion: `pygame.Surface.convert()`. As an example, please refer to line 15.

convert()

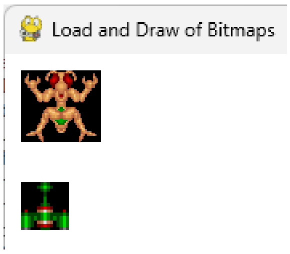

Fig. 2.18: Sizes OK

Resizing is done using `pygame.transform.scale()`. In line 16, the image is scaled to the specified (*width*, *height*) in the unit of pixels. The result shown in figure 2.18 does not quite meet my expectations.

scale()

I do like the size ratios now, but why does a black background suddenly appear? The reason is that the conversion using `convert()` caused the transparency information to be lost. Transparency controls how *see-through* a pixel is. This is achieved by storing not only the three RGB values for each pixel, but also an opacity value. This additional piece of information is called the *alpha channel*.

Alpha channel

I now have two options to make this transparency available again:

Transparency of loaded bitmaps

- `pygame.Surface.convert_alpha()`: Put very simply, the alpha channel is preserved during the conversion. If possible, this should be your method of choice.
- `pygame.Surface.set_colorkey()`: Here, you pass the color that Pygame should skip when drawing onto the target surface. This can lead to two disadvantages. First, transparency levels between fully visible and fully invisible cannot be represented. It would therefore not be possible to make a pixel *semi-transparent*. Second, parts of the figure that have the same color as the background will also appear transparent. If our alien had a black eye in the middle, it would disappear and the alien would have a hole in the center.

Listing 2.30: Load and blit bitmaps, Version 1.2

```
    defender_image = pygame.image.load("images/defender01.png").convert_alpha()  #
    defender_image = pygame.transform.scale(defender_image, (30, 30))

    enemy_image = pygame.image.load("images/alienbig0101.png").convert()
    enemy_image.set_colorkey("black")                # Set colorkey for transparency
    enemy_image = pygame.transform.scale(enemy_image, (50, 45))
```

In source code 2.30 on the preceding page, I tried out both variants, and you can see the result in figure 2.19. Now both bitmaps are visible without a black background; the white background shows through again.

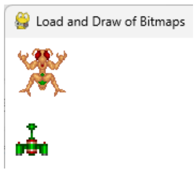

Fig. 2.19: α OK

What I still do not like is the position and the number of attackers. I want to place the defender centered at the bottom and the attackers along the top edge of the screen, arranged so that they are horizontally equidistant. There are two ways to do this: I can specify a minimum spacing and compute the number of attackers, or I can specify the maximum number of attackers and compute the spacing. Which approach I choose depends on my game logic; in most cases, the number is fixed in advance.

equidistant

Listing 2.31: Load and blit bitmaps: config.py (2)

```
WINDOW_WIDTH = 600
WINDOW_HEIGHT = 400
FPS = 60
ALIENS_NOF = 7
```

Listing 2.32: Bitmap: positioning, Version 1.4

```
    defender_image = pygame.image.load("images/defender01.png").convert_alpha()
    defender_image = pygame.transform.scale(defender_image, (30, 30))
    defender_pos_left = (cfg.WINDOW_WIDTH - 30) // 2    # Left coordinate
    defender_pos_top = cfg.WINDOW_HEIGHT - 30 - 5       # Top coordinate
    defender_pos = (defender_pos_left, defender_pos_top)# Create a 2-tuple

    alien_image = pygame.image.load("images/alienbig0101.png").convert_alpha()
    alien_image = pygame.transform.scale(alien_image, (50, 45))
    space_for_aliens = cfg.ALIENS_NOF * 50              # Space occupied by aliens
    space_availible = cfg.WINDOW_WIDTH - space_for_aliens  # Remaining available space
    space_nof = cfg.ALIENS_NOF + 1                      # Number of gaps
    space_between_aliens = space_availible // space_nof # Space per gap

    running = True
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False

        screen.fill("white")
        alien_top = 10                                  # Distance from top
        for i in range(cfg.ALIENS_NOF):                 # Compute and draw positions
            alien_left = (i + 1) * space_between_aliens + i * 50
            alien_pos = (alien_left, alien_top)
            screen.blit(alien_image, alien_pos)
        screen.blit(defender_image, defender_pos)       # Draw defender at its position
        window.flip()
        clock.tick(cfg.FPS)
```

In source code 2.32, the requirements above have been implemented. Let us take a closer look at the individual aspects.

The defender should be positioned centered at the bottom. We remember that the function `blit()` also expects the coordinates of the upper-left corner. So this position has to be calculated first. For the sake of clarity – in a normal source code I would not write the calculation in such a fine-grained way – I calculate the coordinates separately here.

The top edge is fairly easy to determine. If we set `defender_top` to the full height of the screen, `cfg.WINDOWS_HEIGHT`, we would not see the defender because it would stick out below the screen completely. So by how many pixels do we need to move the top edge upward? Exactly by the height of the spaceship, 30 px:

```
18        defender_pos_top = cfg.WINDOWS_HEIGHT - 30
```

However, I do not like how the defender looks glued to the edge this way. So I give it an additional 5 px of space, making it look more as if it were floating in space:

```
18        defender_pos_top = cfg.WINDOWS_HEIGHT - 30 - 5
```

In line 17, the distance of the left edge of the bitmap from the edge of the playfield is calculated. Using

```
17        defender_pos_left = cfg.WINDOWS_WIDTH // 2
```

we would calculate the horizontal center of the screen. However, we cannot use this value, because it would place the left edge of the defender at the horizontal center – that is, too far to the right (see figure 2.20).

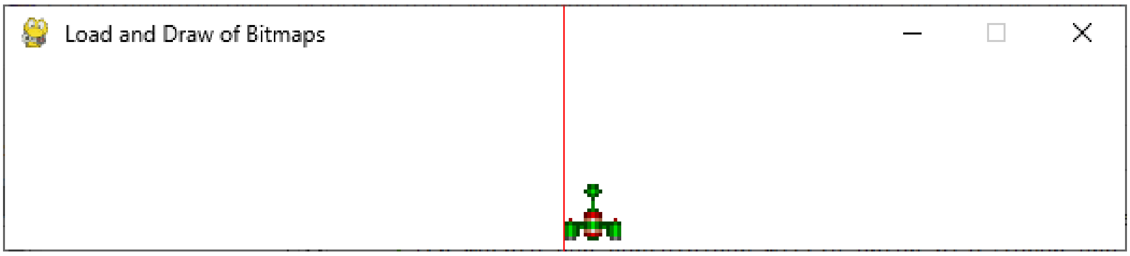

Figure 2.20: Bitmaps positioning defender

However, we can determine exactly how many pixels we have shifted too far to the right and then subtract this value: it is exactly half of the width of the defender (here 30 px):

```
17        defender_pos_left = cfg.WINDOWS_WIDTH // 2 - 30 // 2
```

With the help of a little fraction arithmetic, the expression can be simplified:

```
17        defender_pos_left = (cfg.WINDOWS_WIDTH - 30) // 2
```

Now we move on to the aliens. In the first approach, we want to display them one after another at the top without any overlap. The top edge `alien_top` can be set to a constant value with a pleasant distance of 10 px from the upper edge:

```
35        alien_top = 10
```

The left position `alien_left` has to be determined individually for each alien. Since they are placed directly next to each other at first, the left edge of one alien is exactly one alien width away from the left edge of the next one. So if I am at the 0th alien, its horizontal coordinate is directly at the left edge of the screen. For the 1st alien it is exactly 1×50 px, for the 2nd exactly 2×50 px, and so on, since the alien is 50 px wide. Written as a `for`-loop, it looks like this:

```
    for i in range(cfg.ALIENS_NOF):
        alien_left = i * 50
        alien_pos = (alien_left, alien_top)
        screen.blit(alien_image, alien_pos)
```

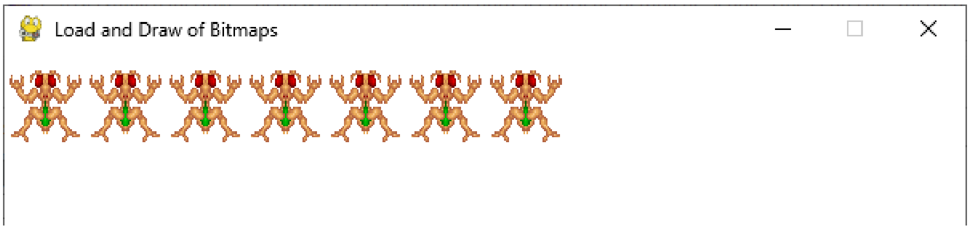

Figure 2.21: Bitmaps positioning alien, Version 1

The entire remaining space after the last alien can now be distributed before, between, and after the aliens in such a way that the spacing is the same between the aliens, between the leftmost alien and the left edge of the screen, and between the rightmost alien and the right edge of the screen. So how many gaps are there? First of all, the two outer gaps on the far left and far right – that makes 2:

```
    space_nof = 2
```

Then there are the gaps between the aliens. This is always one less than the number of aliens (count it to check!):

```
    space_nof = cfg.ALIENS_NOF - 1 + 2
```

thus:

```
    space_nof = cfg.ALIENS_NOF + 1
```

Now the available space `space_available` behind the aliens still has to be calculated. I do this by first calculating the space occupied by the aliens, `space_for_aliens`

```
    space_for_aliens = cfg.ALIENS_NOF * 50
```

and subtract this value from the screen width.

```
    space_availible = cfg.WINDOWS_WIDTH - space_for_aliens
```

So I now have the available space stored in `space_available` and the number of gaps that need to be filled stored in `space_nof`. If I now want to determine the width of the gaps, `space_between_aliens`, I simply have to divide these two values:

```
        space_between_aliens = space_availible // space_nof
```

Now we only need to adjust the calculation of `alien_left`. First, we shift the starting position by one such gap (see figure 2.22):

```
        for i in range(cfg.ALIENS_NOF):
            alien_left = space_between_aliens + i * 50
            alien_pos = (alien_left, alien_top)
            screen.blit(alien_image, alien_pos)
```

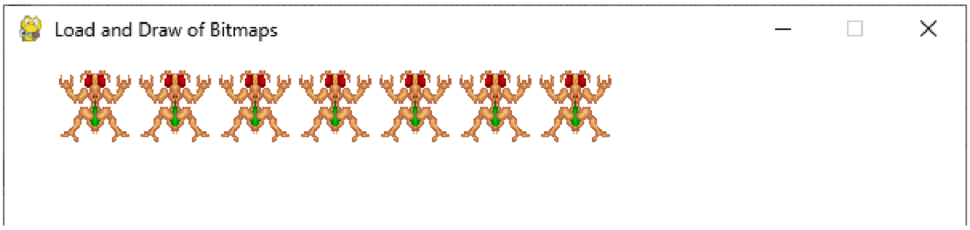

Figure 2.22: Bitmaps positioning alien, Version 2

Now the distance from one left edge to the next, which previously consisted only of the width of the alien, must be extended by the spacing `space_between_aliens`:

```
        for i in range(cfg.ALIENS_NOF):
            alien_left = (i + 1) * space_between_aliens + i * 50
            alien_pos = (alien_left, alien_top)
            screen.blit(alien_image, alien_pos)
```

And just like that, everything fits (see figure 2.23).

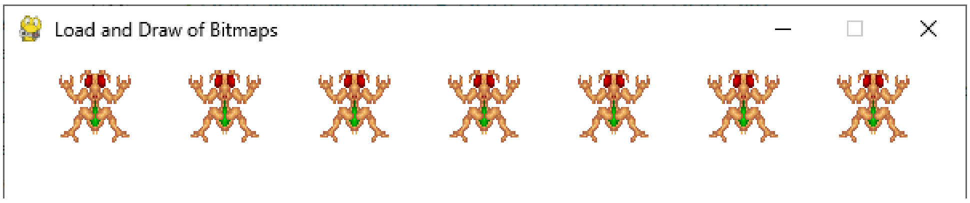

Figure 2.23: Bitmaps positioning alien, Version 3

Why am I explaining this in so much detail?

Not because it's particularly difficult, but because when you're just starting out with programming, it's easy to feel overwhelmed when trying to understand other people's source code. Where do numbers like –1 even come from? This step-by-step approach is meant to reassure you. Complex solutions are often built from many small, simple pieces put together.

2.3.2 More Input

2.3.2.1 Blitting Parts of a Bitmap

area

Very often, only parts of a bitmap need to be blitted. For this purpose, the function `Surface.blit()` provides the parameter `area`.

Fig. 2.24: Tiles to build a forest

As an example, I use a bitmap that consists of tiles of size 32 px × 32 px. From these tiles, I could build a forest and lake landscape for a game. The logic of this small application is that, using the arrow keys, I jump 32 px to the right, left, up, or down, and in this way move from tile to tile. The tile currently selected is marked with a red rectangle (line 34) and drawn as a sub-image into the lower right corner of the window – in this example, a tent or a small hut.

clamp()

Using `clamp()` I make sure that I cannot wander outside the image.

In line 35, this approach is applied as shown in figure 2.24. The first parameter of `blit()` is the bitmap – here referenced as `image` – that is to be drawn. The second parameter `(512-32, 512-32)` specifies the position within the window where the image should be drawn. How do we arrive at these values? The entire image has a width of 512 px and a height of 512 px. Each tile has a size of 32 px × 32 px, and therefore the upper-left corner of the tile must be positioned 32 px away from the edges. The third parameter – the `area` – is a 4-tuple. Its values represent *left*, *top*, *width*, and *height*. The variables `x` and `y` are determined by movement using the arrow keys, which is a preview of later sections of this book (see section 2.4 on page 59). The width and height of the tiles are fixed at 32 px.

Listing 2.33: Blit a part of a bitmap

```
4  def main():
5      pygame.init()
6      window = pygame.Window(size=(512, 512), title="Draw a Part of a Bitmap")
7      screen = window.get_surface()
8      clock = pygame.time.Clock()
9
10     image = pygame.image.load("images/forest_tiles.png")
11     x, y = 0, 0
12
13     running = True
14     while running:
15         for event in pygame.event.get():
16             if event.type == pygame.QUIT:
```

```
                running = False
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_ESCAPE:
                    running = False
                elif event.key == pygame.K_RIGHT:
                    x += 32
                elif event.key == pygame.K_LEFT:
                    x -= 32
                if event.key == pygame.K_DOWN:
                    y += 32
                if event.key == pygame.K_UP:
                    y -= 32
        x = pygame.math.clamp(x, 0, 512-32)
        y = pygame.math.clamp(y, 0, 512-32)

        screen.fill("white")
        screen.blit(image, (0, 0))
        pygame.draw.rect(screen, "red", (x, y, 32, 32), 2)   # Draw rectangle around the part
        screen.blit(image, (512-32, 512-32), (x, y, 32, 32)) # Blit a part of the image
        window.flip()
        clock.tick(60)

    pygame.quit()
```

2.3.2.2 Message Box

A message box is a simple way to communicate an information, a warning, or an error to the player. Its appearance can only be customized to a very limited extent and it usually does not fit into the visual design concept of a game. For this reason, message boxes are almost never used for in-game interactions. However, they are well suited for use during installation or configuration, or when real errors occur that require quick and clear interaction.

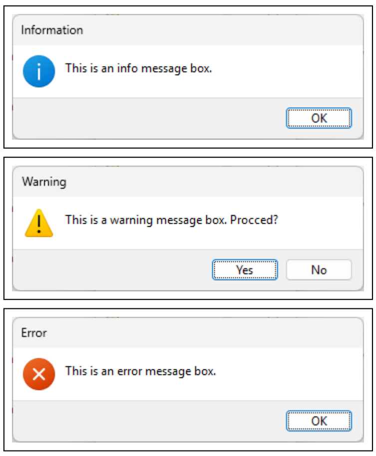

Fig. 2.25: Messageboxes

Here, I have only shown the crucial part of the program in source code 2.34 on the following page; the rest is not important. The call is made using `pygame.display.message_box()`.

message-_box()

In line 20, an information message is displayed. The first parameter is the text of the window's title bar. The second parameter is the message text, which can also be much longer and formatted. After that, the type of the message box is specified. There are three – self-explanatory – types available: `info`, `warn`, and `error`. This setting determines the icon that is shown (see figure 2.25). The call for an error message works in exactly the same way, as shown in line 30.

A bit more is demonstrated with the warning call starting at line 24. First of all, you can see that the named parameter `buttons` is passed. It contains a list of strings. Each string is the label text of one displayed button (see

Button

figure 2.25 on the preceding page). So how do we find out which button was pressed? By using the return value. Internally, the list of button labels is numbered, and each button is assigned an index. The index value of the pressed button is then returned and – as in this example with `a` – stored in a variable. Which button has which index, and how to proceed afterwards, is something you need to keep in mind. In this example, the value is simply printed to demonstrate the effect.

Index

Listing 2.34: Types of Messageboxes

```
                elif event.key == pygame.K_1:
                    pygame.display.message_box("Information",   #
                                               "This is an info message box.",
                                               "info")
                elif event.key == pygame.K_2:
                    a = pygame.display.message_box("Warning",   #
                                                   "This is a warning message box. Procced?",
                                                   "warn",
                                                   buttons=["Yes", "No"])
                    print("User selected:", a)
                elif event.key == pygame.K_3:
                    pygame.display.message_box("Error",         #
                                               "This is an error message box.",
                                               "error")
```

2.3.2.3 Creating Bitmaps

Surface()

Bitmaps do not necessarily have to be loaded from disk using `pygame.image.load()`. It is also possible to create a bitmap at runtime using `pygame.Surface()`. Let us take another look at section 2.2.2.2 on page 36. In that section, an animated landscape was created using drawing primitives. These primitives are drawn in every frame – for example 60 times per second. In the end, this is an enormous waste of computing time.

Creating bitmaps during runtime

A much more efficient approach is to draw the graphics once onto a `Surface` object and then only blit this bitmap to the correct position. Blitting `Surface` objects is much faster than drawing the shapes again and again.

Let us look at this starting with the `Meadow` class. In the constructor, a `Surface` object is created in line 11. For that, we need a width and a height. Both values are computed beforehand and stored in the local variable `widthheight`; this does not need to be a class attribute anymore, because the information is only required to create the bitmap and is not needed later.

After that, the surface is filled completely with a green color. Now we have a finished meadow graphic, and in `draw()` it only needs to be blitted.

Listing 2.35: Creating Bitmaps: Class `Meadow`

```
class Meadow:
    def __init__(self, window: pygame.window.Window, horizon: int) -> None:
        self.window = window
        self.pos = 0, horizon
        widthheight = self.window.size[0], self.window.size[1] - horizon
        self.surface = pygame.Surface(widthheight)                      # Meadow surface
        self.surface.fill((50, 180, 50))

    def draw(self) -> None:
        screen = self.window.get_surface()
        screen.blit(self.surface, self.pos)
```

The `Sky` class follows a similar approach. In the constructor, a `Surface` object is created and filled with a shade of blue, and in `draw()` the bitmap is simply blitted. Only `update()` remains computationally more expensive, since the color of the sky changes depending on the position of the sun.

Listing 2.36: Creating Bitmaps: Class `Sky`

```
class Sky:
    def __init__(self, window: pygame.window.Window, horizon: int) -> None:
        self.window = window
        self.pos = 0, 0
        self.color = (100, 150, 255)
        self.surface = pygame.Surface((self.window.size[0], horizon))   # Sky surface
        self.surface.fill(self.color)

    def update(self, progress: float) -> None:
        brightness = max(0, min(1, math.sin(progress * math.pi)))
        blue = int(80 + brightness * 120)
        self.color = (100, blue, 235)
        self.surface.fill(self.color)

    def draw(self) -> None:
        screen = self.window.get_surface()
        screen.blit(self.surface, self.pos)
```

The `Tree` class is more similar to `Meadow`. In line 40, the bitmap is created and its contents are drawn using drawing primitives. However, the `Surface` object is created with the additional parameter `pygame.SRCALPHA`. This parameter ensures that the unpainted background of the `Surface` object remains transparent. Otherwise, a black background with the size of the `Surface` object would appear around the tree; with transparency enabled, the sky, sun, and meadow can be seen through it. SRCALPHA

Listing 2.37: Creating Bitmaps: Class `Tree`

```
36 class Tree:
37     def __init__(self, window: pygame.window.Window, horizon: int) -> None:
38         self.window = window
39         self.pos = (65, horizon - 80)
40         self.surface = pygame.Surface((90, 120), pygame.SRCALPHA)       # Tree surface
41         pygame.draw.rect(self.surface, (120, 80, 40), (35, 60, 20, 60))
42         pygame.draw.circle(self.surface, (20, 140, 20), (45, 35), 35)
43 
44     def draw(self) -> None:
45         screen = self.window.get_surface()
```

```
        screen.blit(self.surface, self.pos)
```

The same approach is used in the `House` class in line 53.

Listing 2.38: Creating Bitmaps: Class `House`

```
49 class House:
50     def __init__(self, window: pygame.window.Window, horizon: int) -> None:
51         self.window = window
52         self.pos = (200, horizon - 70)
53         self.surface = pygame.Surface((150, 160), pygame.SRCALPHA)     # House surface
54         pygame.draw.rect(self.surface, (200, 100, 100), ((0,60), (150, 100)))
55         pygame.draw.polygon(
56             self.surface,
57             (150, 50, 50),
58             [(0,60), (75, 0), (150, 60)]
59         )
60         pygame.draw.rect(self.surface, (100, 60, 30), (60, 100, 30, 60))
61 
62     def draw(self) -> None:
63         screen = self.window.get_surface()
64         screen.blit(self.surface, self.pos)
```

And, for the sake of completeness, the `Sun` class as well.

Listing 2.39: Creating Bitmaps: Class `Sun`

```
class Sun:
    def __init__(self, window: pygame.window.Window, horizon: int) -> None:
        self.speed = 1
        radius = 40
        self.pos = [-radius, 0]
        self.horizon = horizon
        self.window = window
        self.surface = pygame.Surface((radius*2, radius*2), pygame.SRCALPHA)    # Sun surface
        pygame.draw.circle(self.surface,  (255, 220, 0), (radius, radius), radius)

    def update(self) -> float:
        self.pos[0] += self.speed
        progress = self.pos[0] / self.window.size[0]   # 0.0 -> 1.0
        self.pos[1] = round(self.horizon * (1 - math.sin(progress * math.pi)))
        return progress

    def draw(self) -> None:
        screen = self.window.get_surface()
        screen.blit(self.surface, (self.pos[0], self.pos[1]))
```

2.3.3 What was new?

- The position values are needed when drawing on the screen. Later, we will see that we also need these position values for other questions, such as collision detection. The position always refers to the upper-left corner of the bitmap, or in other words: *The coordinate system has its origin in the upper left, not in the lower left.*
- We often have to perform basic geometry calculations, and it is best to do them step by step.
- For such geometry calculations, the following information is needed: the position of the bitmap, its width, and its height. So far, we have treated width and height as constants, but that is not a good long-term solution.

The following Pygame elements were introduced

- `pygame.display.Info()` : https://pyga.me/docs/ref/display.html#pygame.display.Info
- `pygame.display.message_box()` : https://pyga.me/docs/ref/display.html#pygame.display.message_box
- `pygame.image` : https://pyga.me/docs/ref/image.html
- `pygame.image.load()` : https://pyga.me/docs/ref/image.html#pygame.image.load
- `pygame.Surface()`: https://pyga.me/docs/ref/surface.html
- `pygame.Surface.blit()`: https://pyga.me/docs/ref/surface.html#pygame.Surface.blit
- `pygame.Surface.convert()`: https://pyga.me/docs/ref/surface.html#pygame.Surface.convert
- `pygame.Surface.convert_alpha()`: https://pyga.me/docs/ref/surface.html#pygame.Surface.convert_alpha
- `pygame.Surface.set_colorkey()`: https://pyga.me/docs/ref/surface.html#pygame.Surface.set_colorkey

- `pygame.SRCALPHA`:
  https://pyga.me/docs/ref/surface.html
- `pygame.transform.scale()`:
  https://pyga.me/docs/ref/transform.html#pygame.transform.scale

2.3.4 Homework

1. Look for freely available sources of game graphics (sprites). If you know that you want to work with this more intensively, also look for sources that are behind a paywall.
2. Blit your own graphics to sensible positions within your window.
3. Try to build a realistic background for a simple game using graphics. If needed, make use of the option to blit sub-images from a larger bitmap.

2.4 Moving Bitmaps

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/04%20Moving

2.4.1 Class `Rect`/`FRect`

In the summary of the previous chapter, we noted that when displaying bitmaps we need the *upper-left corner* as the position value, and we need the *height* and *width*, for example for distance calculations. These values can be conveniently encoded in a rectangle. For this purpose, Pygame provides the classes `pygame.rect.Rect` (integers only) and `pygame.rect.FRect` (floating-point numbers). In figure 2.26, you can find what I consider to be the most important attributes of this class.

Rect

FRect

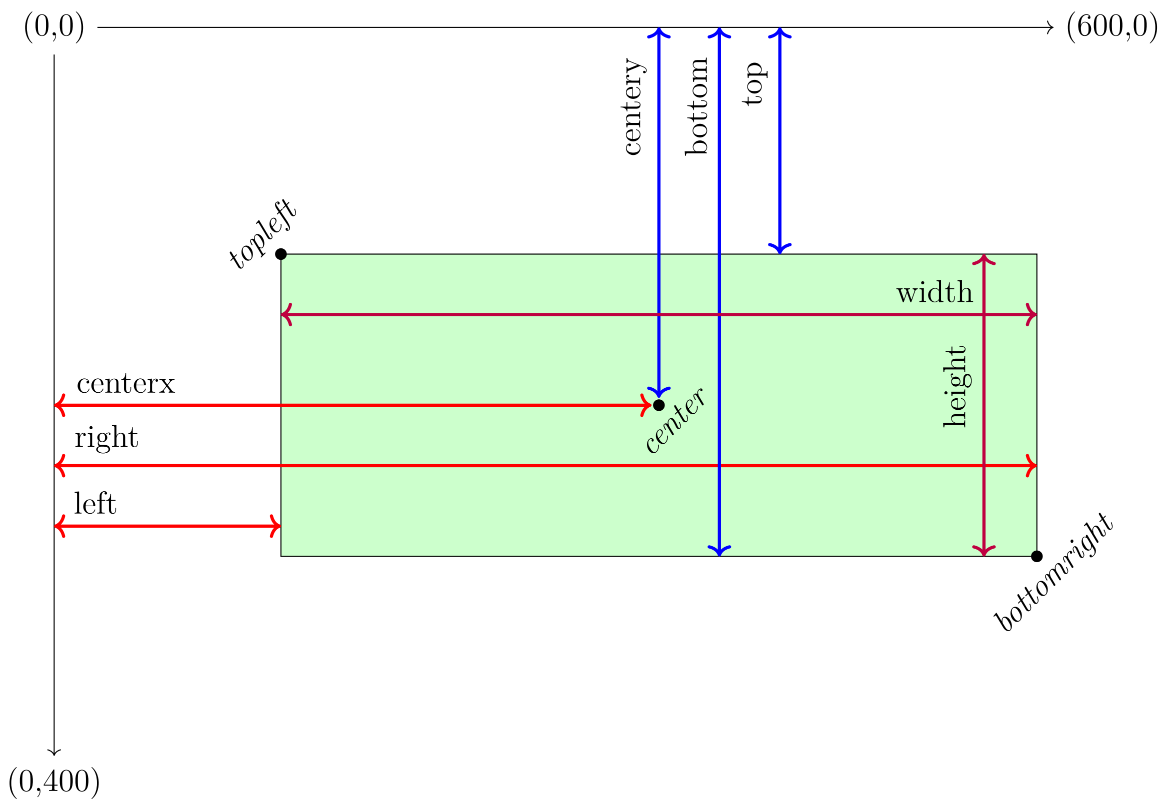

Figure 2.26: Elements of a `Rect-`/`FRect-`Object

In the figure, line segments are shown in normal font, while points are shown in *italic font*. The segments are one-dimensional, and the points are two-dimensional (x, y). The coordinate x represents the horizontal distance from the origin of the coordinate system or the upper-left corner of the rectangle, and y represents the vertical distance. The meaning of the individual labels should be self-explanatory.

The nice advantage is that all these values are computed from each other. For example, if I set `topleft = (10, 10)` and `width, height = 30, 40`, all other values are calculated automatically. I no longer need to compute the right edge manually using `left + width`; instead, I can directly use `right`.

It is also often useful to work with the center position `center` or the corresponding coordinates `centerx` and `centery`. If I change the center to `center = (100, 10)`, all other values are updated accordingly and do not need to be recalculated by me – very convenient.

Let us take a look at a reduced version of the last source code. In source code 2.41, the `Rect` class is already being used; i.e., in line 3 the window dimensions are stored in a `Rect` object.

Listing 2.40: Moving Bitmaps: `config.py`

```
1  import pygame
2
3  WINDOW = pygame.Rect((0, 0), (600, 100))          # Rect object
4  FPS = 60
```

As a result, the screen information can be accessed conveniently and without performing manual calculations in line 9, line 18, and line 19.

Listing 2.41: Moving Bitmaps, Version 1.0

```
6  def main():
7      pygame.init()
8      window = pygame.Window(
9          size=cfg.WINDOW.size,                         # Accessing a Rect attribute
10         title="Movement",
11         position=(10, 50))
12     screen = window.get_surface()
13     clock = pygame.time.Clock()
14
15     defender_image = pygame.image.load("images/defender01.png").convert_alpha()
16     defender_image = pygame.transform.scale(defender_image, (30, 30))
17     defender_rect = defender_image.get_rect()          # Rect-object
18     defender_rect.centerx = cfg.WINDOW.centerx         # Not only using left
19     defender_rect.bottom = cfg.WINDOW.height - 5       # Not only using top
20
21     running = True
22     while running:
23         # Events
24         for event in pygame.event.get():
25             if event.type == pygame.QUIT:
26                 running = False
27
28         # Update
29
```

```
30         # Draw
31         screen.fill("white")
32         screen.blit(defender_image, defender_rect)      # blit can also take a Rect
33         window.flip()
34         clock.tick(cfg.FPS)
35 
36     pygame.quit()
```

For `Surface` objects, we can conveniently create a `Rect` object using `pygame.Surface.get_rect()` (line 17). Positioning can now be handled much more easily via the attributes. For example, the center no longer needs to be part of a calculation; instead, I can directly set the horizontal center to half the window width (line 18). Likewise, the vertical coordinate no longer has to be considered from the top edge; instead, I can specify the distance of the bottom edge from the screen edge in a much more intuitive way (line 19). And as a final bonus, the `Rect` object can even be passed directly as a parameter to the `blit()` function (line 32).

get_rect()

blit()

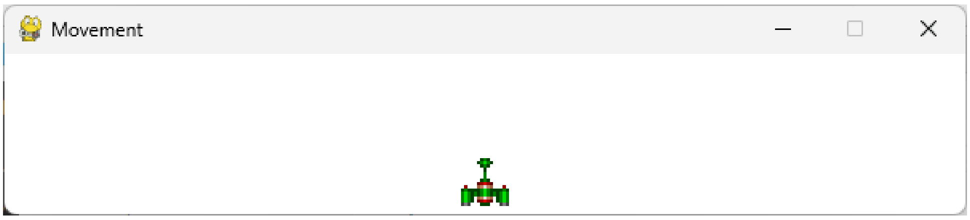

Figure 2.27: Moving Bitmaps, Version 1.0

The result is unspectacular (see figure 2.27) and has nothing to do with movement yet.

2.4.2 Introduction

Movement in games is animated by changing positions. If the spaceship is supposed to move to the right, the horizontal coordinate of the ship therefore has to increase. Which horizontal coordinate you use for this – `left`, `right`, or `centerx` – can be chosen depending on your game logic. In our example, this does not matter, so I will use `left`.

```
defender_rect.left += 1
```

This small addition alone now causes our spaceship to move to the right.

The +1 encodes two pieces of information:

- **Direction:** Here, the sign is `+`. This increases the value of `left` in each loop iteration; as a result, the left edge of the graphic moves to the right. If you wanted to move to the left, the sign would have to be `-`. In that case, the horizontal coordinate would become smaller and approach 0. Completely analogously, the sign also controls the direction in the vertical axis. A `+` moves the graphic downward, and a `-` moves it upward. Try it out!

- **Speed:** The `1` specifies by how much the value of `left` changes. The larger this value is, the larger the jumps between frames; the movement appears faster.

Listing 2.42: Moving Bitmaps, Version 1.2

```
17  defender_speed = 2          #
18  defender_direction_h = +1   #
19
20  running = True
21  while running:
22      # Events
23      for event in pygame.event.get():
24          if event.type == pygame.QUIT:
25              running = False
26
27      # Update
28      defender_rect.left += defender_direction_h * defender_speed  # more flexible
```

These two pieces of information are now used in source code 2.42 to make movement much more flexible. In line 17, the speed is now represented by the variable `defender_speed`. This would allow us to change the speed dynamically during the game, for example when accelerating by firing rocket thrusters.

The direction is also stored in a variable in line 18: `defender_direction`. At the moment it is positive, but we will soon see that we can also use it for changing direction.

Both values can now be used in line 28 to calculate the new horizontal position.

If you run the program, the defender will leave the screen after a while and disappear beyond the right edge, never to be seen again. Let us now use our rectangle for a first simple collision check. I want the spaceship to *bounce* off the edges and reverse its direction.

Listing 2.43: Move Bitmaps, Version 1.3

```
27      # Update
28      defender_rect.left += defender_direction_h * defender_speed
29      if defender_rect.right >= cfg.WINDOW.right:     # Right edge reached
30          defender_direction_h *= -1                  # Change direction
31      elif defender_rect.left <= cfg.WINDOW.left:     # Left edge reached
32          defender_direction_h *= -1
```

Direction change

I hope you can recognize the idea behind the code. After calculating the new horizontal position, line 29 checks whether the new right edge of the bitmap has reached or exceeded the right edge of the screen. If this is the case, the sign of the direction variable is simply reversed! The same logic works analogously when the left edge of the screen is reached.

Try combining this with vertical movement as well.

There is still one problem: In line 28, the new position is assigned to the `Rect` object even though it may already extend beyond the edge. With a speed of `1` or `2`, this may not be very noticeable, but if we set the speed to the width of the spaceship, the problem becomes obvious (temporarily set `cfg.FPS = 5` so that you can see it clearly). The spaceship ends up leaving the screen halfway.

So we should check the new position first and only then assign it to the `Rect` object `defender_rect`. In this context, let us introduce a very useful method of the `Rect` class: `pygame.rect.Rect.move()`. move()

Listing 2.44: Move Bitmaps, Version 1.4

```
27         # Update
28         newpos = defender_rect.move(defender_direction_h * defender_speed, 0)  # New position
29         if newpos.right >= cfg.WINDOW.right:
30             defender_direction_h *= -1
31             newpos.right = cfg.WINDOW.right      # Align to right edge
32         elif newpos.left <= cfg.WINDOW.left:
33             defender_direction_h *= -1
34             newpos.left = cfg.WINDOW.left        # Align to left edge
35         defender_rect = newpos                   # Accept new position
```

The new function appears for the first time in line 28. It takes two parameters. The first one specifies the horizontal displacement, and the second one specifies the vertical displacement. Since we do not want to change the vertical position, this parameter is constant 0 in our example. The function returns a new `Rect` object containing the updated position values. We store this temporarily in `newpos`.

The subsequent collision checks are then performed using the `newpos` rectangle. If a collision occurs, the direction values are changed as before. Likewise, the left edge of the bitmap is aligned with the left edge of the screen, and the right edge is aligned with the right edge. After that, `newpos` becomes the new rectangle for the defender (line 35).

Figure 2.28: The Defender moves and bounces

2.4.3 More Input

2.4.3.1 Normalizing Speeds (*delta time*)

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/04%20Moving/dt

At the moment, the movement does not depend only on `defender_speed`, but also on the frame rate `cfg.FPS`. To illustrate this dependency, I modified the previous source code for a small experiment (see source code 2.45 and source code 2.46 on the facing page).

Listing 2.45: Movement without normalization: `config.py`

```
1 import pygame
2
3 WINDOW = pygame.Rect((0, 0), (120, 650))
4 FPS = 60 # 10 30 60 120 240 300 600
5 LIMIT = 500
```

In line 4, you can see the different frame rates with which the experiment was carried out. In the line above, the window dimensions are set so that the window is tall and narrow, and in the line below the absolute number of milliseconds is specified during which the spaceship moves upward.

get_ticks()

Line 20 stores the time at which the spaceship's ascent began. For this purpose, the function `pygame.time.get_ticks()` returns the number of milliseconds since the call to `pygame.init()`; for example, 5 ms.

Inside the main loop, the spaceship now moves upward by a certain number of pixels per frame. The new position is calculated by adding the product of direction and speed to the `top` coordinate of the old position (line 31) – so there is nothing new at this point.

After a fixed time interval (`cfg.LIMIT`, here 500 ms), the direction stored in `defender_direction` is set to 0, causing the movement to stop. To do this, line 23 checks whether the current number of milliseconds since the start of the program is greater than `start_time` plus `cfg.LIMIT`.

In numerical terms: during the first loop iteration (frame 1), the condition would be, for example,

Is 17 ms greater than 5 ms + 500 ms?

The answer is *No*, so the spaceship continues to move upward. At frame 61, the condition would be

Is 508 ms greater than 5 ms + 500 ms?

Now the answer is *Yes*, and the direction variable is therefore set to 0, stopping the movement.

Listing 2.46: Movement without normalization

```
def main():
    pygame.init()
    window = pygame.Window(size=cfg.WINDOW.size, title="Movement", position=(10, 50))
    screen = window.get_surface()
    clock = pygame.time.Clock()

    defender_image = pygame.image.load("images/defender01.png").convert_alpha()
    defender_image = pygame.transform.scale(defender_image, (30, 30))
    defender_rect = defender_image.get_rect()
    defender_rect.centerx = cfg.WINDOW.centerx
    defender_rect.bottom = cfg.WINDOW.bottom - 5
    defender_speed = 2
    defender_direction_v = -1

    start_time = pygame.time.get_ticks()                        # Movement starts
    running = True
    while running:
        if pygame.time.get_ticks() > start_time + cfg.LIMIT:    # Ready?
            defender_speed = 0
        # Events
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False

        # Update
        defender_rect.top += defender_direction_v * defender_speed  # New height
        if defender_rect.bottom >= cfg.WINDOW.bottom:
            defender_direction_v *= -1
        elif defender_rect.top <= 0:
            defender_direction_v *= -1

        # Draw
        screen.fill("white")
        screen.blit(defender_image, defender_rect)
        window.flip()
        clock.tick(cfg.FPS)
    print(f"top={defender_rect.top}")

    pygame.quit()
```

In figure 2.29 on the next page, you can see screenshots of the distances the spaceship has traveled after half a second. In all experiments, the speed `defender_speed` remained the same – namely 2. Only the frame rate was increased.

How do these different heights come about? After all, only `defender_speed` is supposed to define the speed. The relationship should become clear in table 2.3 on the following page. The first column shows the speed of an object; in our example, this is the variable `defender_speed`. This value specifies how many pixels per frame the object is moved; this value does not change. The second column shows the frame rate, that is, the number of frames per second. In our example, this value is defined in `cfg.FPS`. The duration of the movement is shown in the third column. We have a duration of 500 ms, that is 0.5 s. In our example, this value is stored in `cfg.LIMIT` and is also the same for all experiments.

The last column shows the calculated distance in pixels that the moving object has traveled. Now the relationship becomes clear: because we repeat the main program loop a different number of times depending on the frame rate, different distances are covered within the same amount of time.

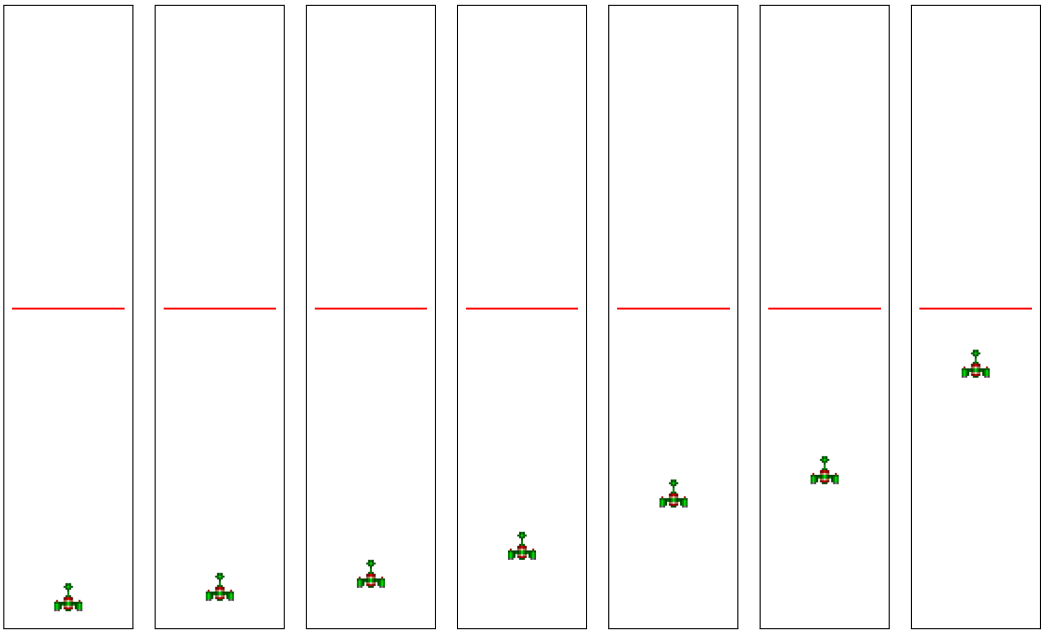

Figure 2.29: Non-normalized movement with identical speed but different frame rates (from left to right: 10, 30, 60, 120, 240, 300, 600)

Table 2.3: Distance without normalized movement

speed ($\frac{px}{f}$) *	FPS ($\frac{f}{s}$) *	time (s) =	distance (px)
2 *	10 *	0.5 =	10
2 *	30 *	0.5 =	30
2 *	60 *	0.5 =	60
2 *	120 *	0.5 =	120
2 *	240 *	0.5 =	240
2 *	300 *	0.5 =	300

What we therefore need is a mechanism that removes the influence of the frame rate again. This factor has to be constructed in such a way that, when multiplied by the frame rate, it always yields 1 as a result. In that case, the frame rate would effectively act like a multiplication by 1 in the overall product and would therefore no longer have any influence.

The obvious approach is to take the inverse of the frame rate, that is $\frac{1}{fps}$. This correction value is called *delta time (dt)*. The calculation would then look, for example, as shown

deltatime

in table 2.4. The second and third columns cancel each other out, so that the distance remains the same – independent of the chosen frame rate.

That is exactly what we tried to achieve.

Table 2.4: Distance with normalized movement

speed ($\frac{px}{s}$) *	FPS ($\frac{f}{s}$) *	dt ($\frac{s}{f}$) *	time (s) =	distance (px)
2 *	10 *	$\frac{1}{10}$ *	0.5 =	1
2 *	30 *	$\frac{1}{30}$ *	0.5 =	1
2 *	60 *	$\frac{1}{60}$ *	0.5 =	1
2 *	120 *	$\frac{1}{120}$ *	0.5 =	1
2 *	240 *	$\frac{1}{240}$ *	0.5 =	1
2 *	300 *	$\frac{1}{300}$ *	0.5 =	1

In this context, it becomes apparent that the distance is surprisingly short: only 1 px per second. Please note that the unit of the first column has changed as well. The speed no longer specifies the number of pixels per frame, but the number of pixels per second! We therefore need to choose a different speed value; based on our window size, I decided on 600 px/s. After one second, our spaceship will have reached the top.

In table 2.5, I calculated the expected final position (`.top`) after half a second. In the left half of the table, the traveled distance is calculated. Surprisingly, it always amounts to 300 px. From the window height (`WINDOW.height`), we have to subtract this distance. In addition, we subtract the height of our spaceship (30 px) and the small offset of 5 px, since we did not want to start the spaceship directly at the bottom edge. We therefore expect our spaceship to reach the final position calculated in table 2.5 after half a second.

Table 2.5: Pixel coordinates with normalized speed

speed *	FPS *	dt *	time =	distance →	`WINDOW.height` -	height -	offset =	`.top`
600 *	10 *	$\frac{1}{10}$ *	0.5 =	300 →	650-300 -	30 -	5 =	315
600 *	30 *	$\frac{1}{30}$ *	0.5 =	300 →	650-300 -	30 -	5 =	315
600 *	60 *	$\frac{1}{60}$ *	0.5 =	300 →	650-300 -	30 -	5 =	315
600 *	120 *	$\frac{1}{120}$ *	0.5 =	300 →	650-300 -	30 -	5 =	315
600 *	240 *	$\frac{1}{240}$ *	0.5 =	300 →	650-300 -	30 -	5 =	315
600 *	300 *	$\frac{1}{300}$ *	0.5 =	300 →	650-300 -	30 -	5 =	315

Although table 2.5 may look complicated, the implementation is surprisingly simple. First, the adjustment in `config.py`. In line 6, the correction factor is defined – as discussed above – as the inverse of the frame rate.

Listing 2.47: Movement with normalization and $dt = 1/fps$: `config.py`

```
import pygame

WINDOW = pygame.Rect((0, 0), (120, 650))
FPS = 60 # 10 30 60 120 240 300 600
LIMIT = 500
DELTATIME = 1.0 / FPS  #
```

The speed is adjusted from 2 to 600 in line 17, and in line 31 the correction factor `DELTATIME` is included as a factor in the calculation. That's it; in figure 2.33 on page 74 we can admire the *perfect* result on one of my slower computers.

Listing 2.48: Movement with normalization and $dt = 1/fps$

```
6  def main():
7      pygame.init()
8      window = pygame.Window(size=cfg.WINDOW.size, title="Movement", position=(10, 50))
9      screen = window.get_surface()
10     clock = pygame.time.Clock()
11
12     defender_image = pygame.image.load("images/defender01.png").convert_alpha()
13     defender_image = pygame.transform.scale(defender_image, (30, 30))
14     defender_rect = defender_image.get_rect()
15     defender_rect.centerx = cfg.WINDOW.centerx
16     defender_rect.bottom = cfg.WINDOW.bottom - 5
17     defender_speed = 600                             # Not px/f but px/s
18     defender_direction_v = -1
19
20     start_time = pygame.time.get_ticks()
21     running = True
22     while running:
23         if pygame.time.get_ticks() > start_time + cfg.LIMIT:
24             defender_speed = 0
25         # Events
26         for event in pygame.event.get():
27             if event.type == pygame.QUIT:
28                 running = False
29
30         # Update
31         defender_rect.top += defender_direction_v * defender_speed * cfg.DELTATIME  #
32         if defender_rect.bottom >= cfg.WINDOW.bottom:
33             defender_direction_v *= -1
34         elif defender_rect.top <= 0:
35             defender_direction_v *= -1
36
37         # Draw
38         screen.fill("white")
39         pygame.draw.line(screen, "red", (0, 315), (cfg.WINDOW.width, 315), 2)
40         screen.blit(defender_image, defender_rect)
41         window.flip()
42         clock.tick(cfg.FPS)
43     print(f"top={defender_rect.top}")
```

2.4.3.2 Optimizing Normalized Speed

Two issues cause the error shown

in figure 2.33 on page 74:

- **Rounding errors:** In theory, multiplying the frame rate by the delta time should always yield 1.0. Unfortunately, this is not the case. When computing delta time, a value close to the exact value is stored due to the way a floating-point number is represented; for example, instead of the exact value $0.0\overline{3}$ for $\frac{1.0}{30.0}$, the stored value is 0.033333333333333330. Over time, this rounding error accumulates to perceptible amounts.

- **Incorrect understanding of** ***fps*****:** The frame rate does not define that the main program loop is executed *exactly* 60 times per second, for example, but that it is executed *at most* 60 times per second. If the game logic or rendering takes more time than $\frac{1}{60}$ s, at least one frame will be skipped. This can also happen if the computer loses performance due to other operations (for example, cloud synchronization).

We cannot solve the first problem without a significant loss of performance, so we will not consider it any further. The second problem, however, can be addressed. Instead of a fixed delta time, we need a value that is based on the actual duration of a frame. The method `pygame.clock.tick()` in line 47 provides a good estimate of the frame time. Luckily, this feature is already built in and can therefore be used directly (see source code 2.49). The result in figure 2.34 on page 74 is better, but still not satisfying :-(. In figure 2.30 on the following page, you can see that the red line more or less dances around the green one, and no clear trend is visible.

tick()

Listing 2.49: Normalized Movement with `pygame.clock.tick()`

```
46         window.flip()
47         Settings.DELTATIME = clock.tick(Settings.FPS) / 1000.0   #
48     print(f"top={defender_rect.top}")
```

The cause is a problem that we should have fixed immediately. In the assignment in line 31 in source code 2.48 on the preceding page, the right-hand side is a floating-point value, while the left-hand side is an integer. As a result, the decimal places are truncated in every loop iteration. For example, if the spaceship were supposed to move by 5.8 px in each frame, the following values would occur:

Float for logic, Int for rendering

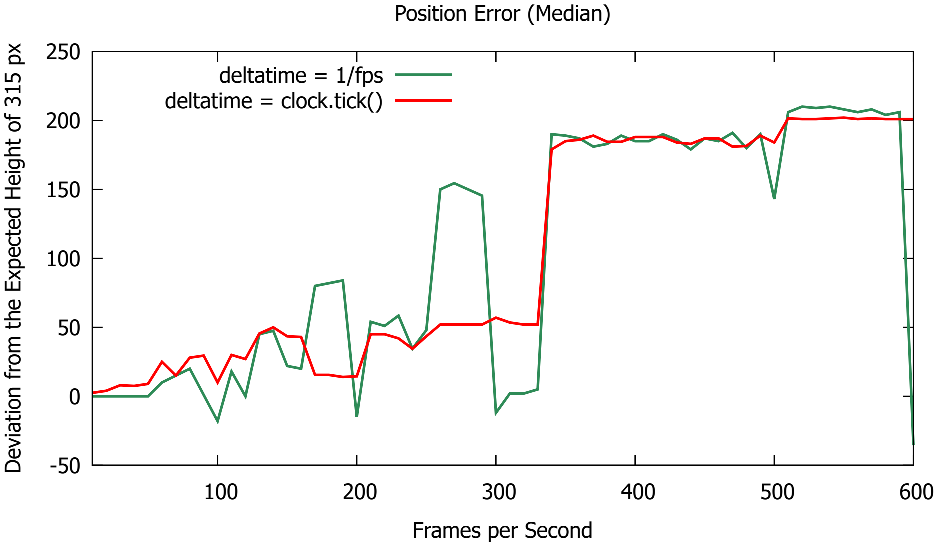

Figure 2.30: Position Error of $1/fps$ and `pygame.clock.tick()`

Table 2.6: Error Propagation

Frame	1	2	3	4	5	6	7	8	9
Actual Value	5.0	10.0	15.0	20.0	25.0	30.0	35.0	40.0	45.0
Correct Value	5.8	11.6	14.4	23.2	29.0	34.8	40.6	46.4	52.2
Error	0.8	1.3	2.4	3.2	4.0	4.8	5.6	6.4	7.2

FRect

Recently, Pygame introduced a variant of `Rect`, namely `FRect`. In this class, all values are stored as `floats`, so fractional parts are no longer truncated. Alternatively, we would have to store the position values independently of the `Rect` object in an additional float variable in order to preserve the fractional parts, for example in a `pygame.math.Vector2` object.

Listing 2.50: Normalized Movement with Positions in Float

```
11 def main():
12     pygame.init()
13     window = pygame.Window(size=Settings.WINDOW.size, title="Movement", position=(10, 50))
14     screen = window.get_surface()
15     clock = pygame.time.Clock()
16
17     defender_image = pygame.image.load("images/defender01.png").convert_alpha()
18     defender_image = pygame.transform.scale(defender_image, (30, 30))
19     defender_rect = pygame.FRect(defender_image.get_rect())  # float
20     defender_rect.centerx = Settings.WINDOW.centerx
21     defender_rect.bottom = Settings.WINDOW.bottom - 5
22     defender_speed = 600
23     defender_direction_v = -1
24
25     start_time = pygame.time.get_ticks()
26     running = True
27     while running:
28         if pygame.time.get_ticks() > start_time + Settings.LIMIT:
29             defender_speed = 0
30         # Events
31         for event in pygame.event.get():
32             if event.type == pygame.QUIT:
33                 running = False
34
35         # Update
36         defender_rect.top += defender_direction_v * defender_speed * Settings.DELTATIME
37         if defender_rect.bottom >= Settings.WINDOW.bottom:
38             defender_direction_v *= -1
39         elif defender_rect.top <= 0:
40             defender_direction_v *= -1
41
42         # Draw
43         screen.fill("white")
44         pygame.draw.line(screen, "red", (0, 315), (Settings.WINDOW.width, 315), 2)
45         screen.blit(defender_image, defender_rect)
46         window.flip()
47         Settings.DELTATIME = clock.tick(Settings.FPS) / 1000.0
48     print(f"top={defender_rect.top}")
```

In figure 2.35 on page 75, we can see that the result has already improved significantly. The deviation from the optimal value 315 px has also been reduced dramatically. The difference is visualized in figure 2.31 on the next page.

However, there is yet another source of error: `pygame.clock.tick()` does not provide enough decimal precision. Over long runtimes, the missing fractional parts accumulate and again lead to noticeable errors. There are better Python functions for measuring elapsed time.

line 22 of source code 2.51 on page 73, `time.time()` is used to return the number of seconds since January 1, 1970 as a floating-point number. The fractional part represents fractions of a second. This measurement is more precise than the one provided by `pygame.clock.tick()` and, depending on the time-measurement capabilities of the computer architecture and the operating system, can provide more decimal places—up to the nanosecond range.

time()

In line 45, the current time is measured after one frame has elapsed, and in the following line the elapsed time is computed. This value represents the actual *delta time* of the

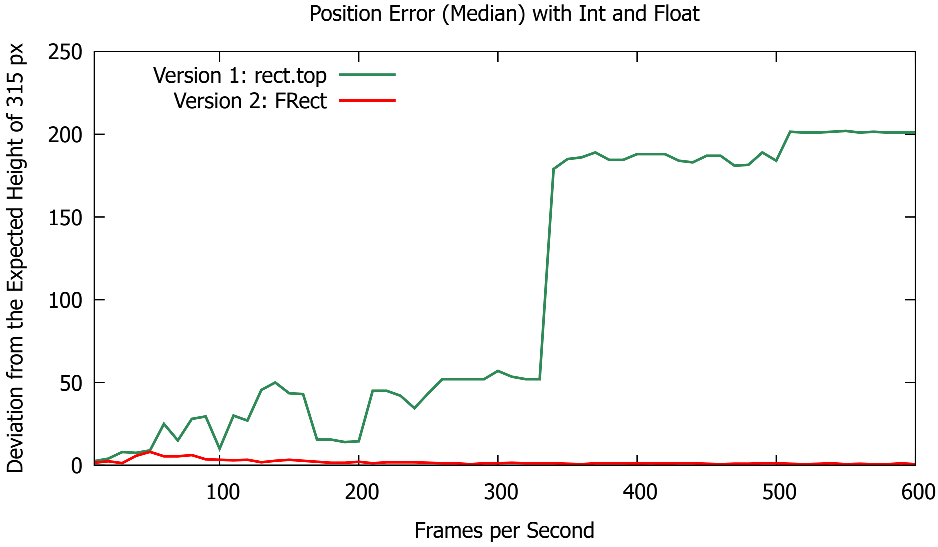

Figure 2.31: Position Error of Rect and FRect

frame, now with higher precision. Afterwards, in line 47, the new start time of the next frame is stored so that the elapsed time can be computed again after the next frame.

figure 2.36 on page 75 shows that the target positions are reached almost perfectly for all frame rates. However, comparing the position errors in figure 2.32 on the facing page does not allow for a clear evaluation. I suspect that experiments with significantly longer runtimes would make a difference visible. We must – and can – live with the remaining error.

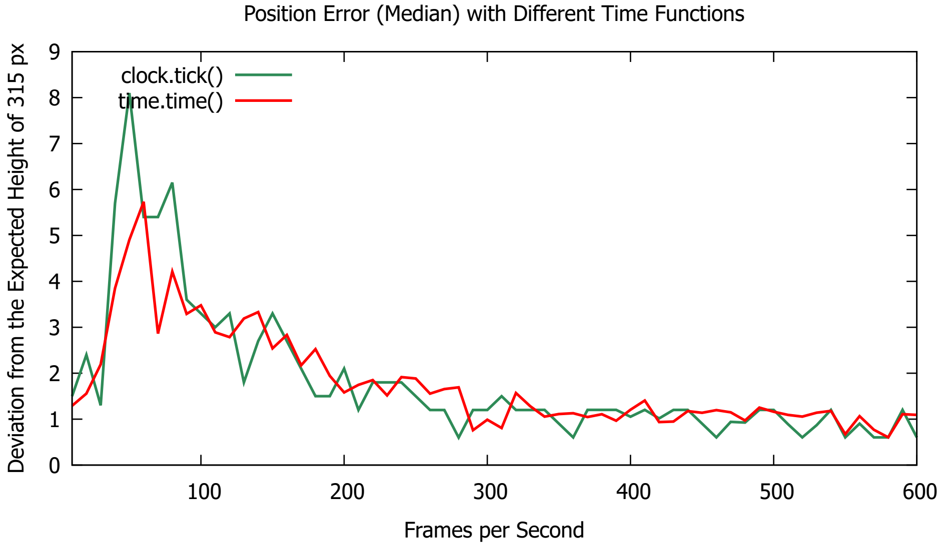

Figure 2.32: Position Error with Different Time Functions

Listing 2.51: Movement with normalization and `time.time()`

```python
def main():
    pygame.init()
    window = pygame.Window(size=cfg.WINDOW.size, title="Movement", position=(10, 50))
    screen = window.get_surface()
    clock = pygame.time.Clock()

    defender_image = pygame.image.load("images/defender01.png").convert_alpha()
    defender_image = pygame.transform.scale(defender_image, (30, 30))
    defender_rect = pygame.FRect(defender_image.get_rect())
    defender_rect.centerx = cfg.WINDOW.centerx
    defender_rect.bottom = cfg.WINDOW.height - 5
    defender_speed = 600
    defender_direction_v = -1

    start_time = pygame.time.get_ticks()
    time_previous = time()  # remember start time
    running = True
    while running:
        if pygame.time.get_ticks() > start_time + cfg.LIMIT:
            defender_speed = 0
        # Events
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False

        # Update
        defender_rect.top += defender_direction_v * defender_speed * cfg.DELTATIME
        if defender_rect.bottom >= cfg.WINDOW.height:
            defender_direction_v *= -1
        elif defender_rect.top <= 0:
            defender_direction_v *= -1

        # Draw
        screen.fill("white")
        pygame.draw.line(screen, "red", (0, 315), (cfg.WINDOW.width, 315), 2)
```

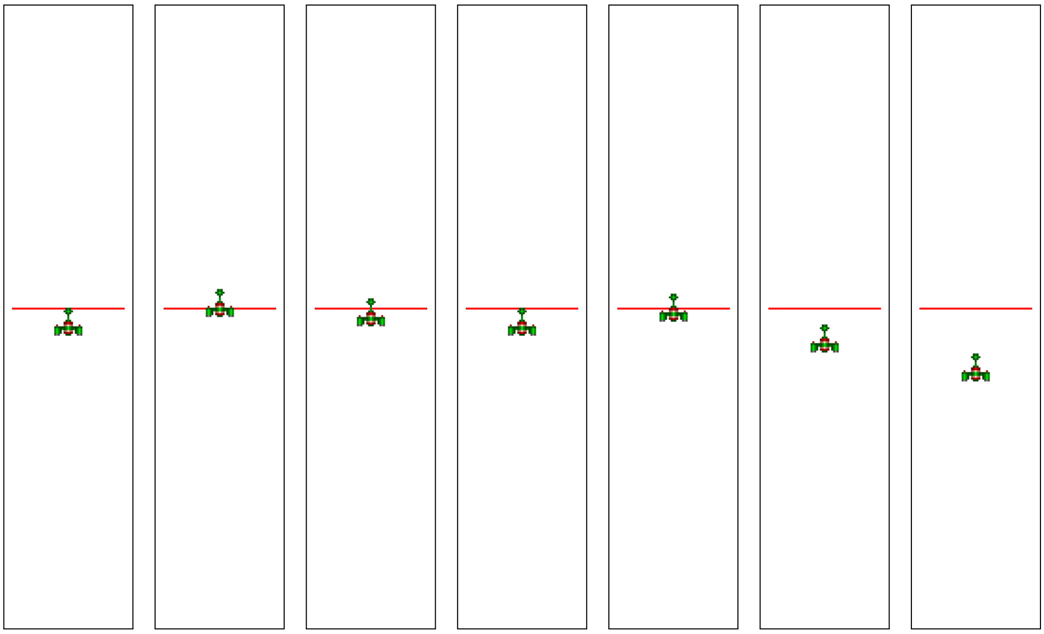

Figure 2.33: Movement with normalization and $dt = 1/fps$ using constant speed but different fps (from left to right: 10, 30, 60, 120, 240, 300, 600)

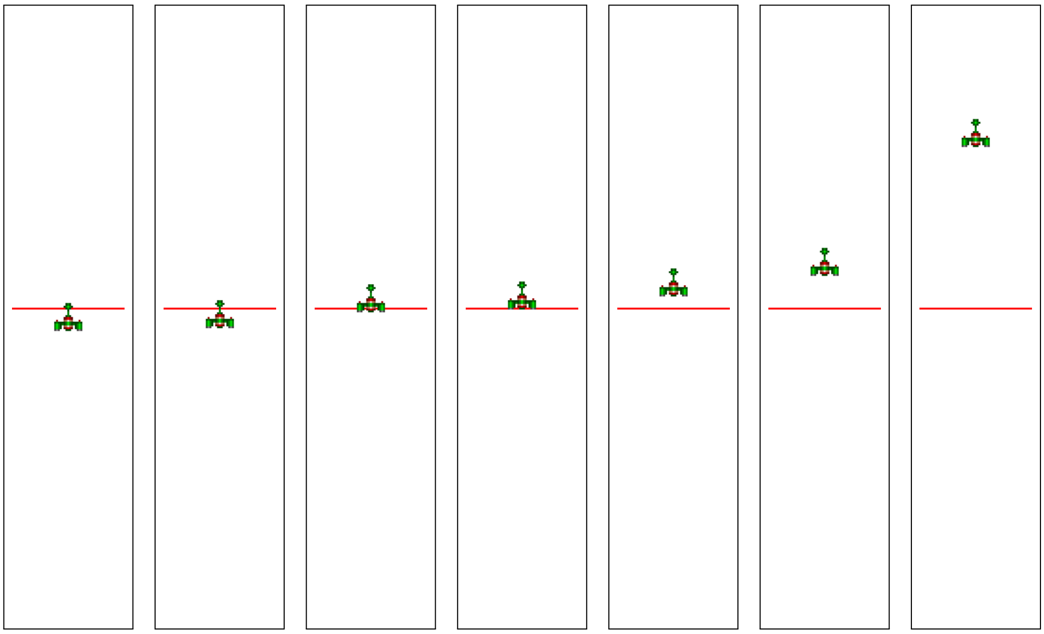

Figure 2.34: Movement with normalization and `pygame.clock.tick()` using constant speed but different fps (from left to right: 10, 30, 60, 120, 240, 300, 600)

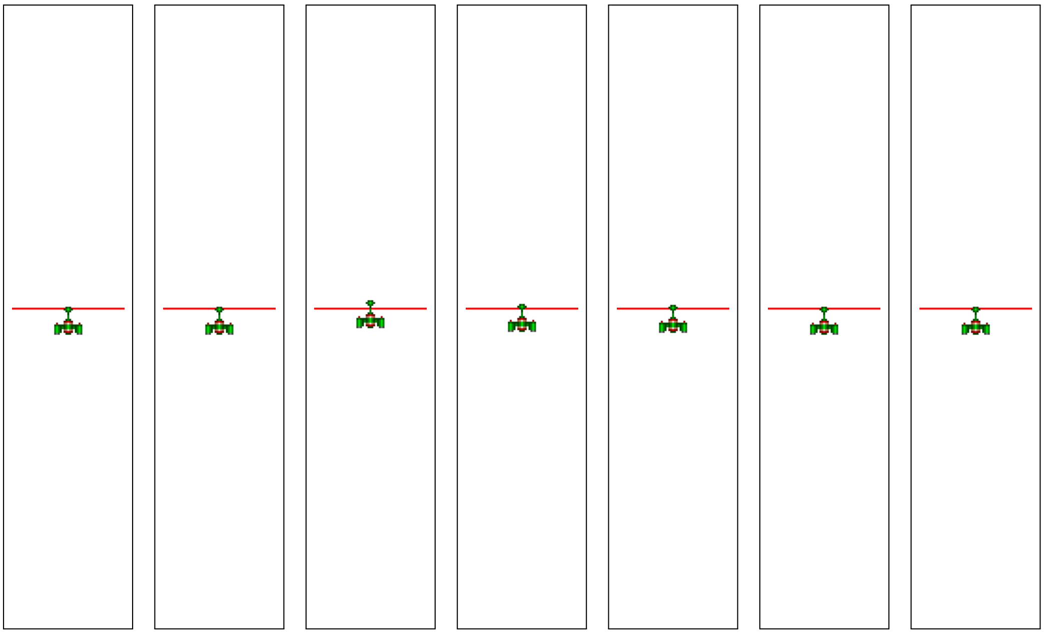

Figure 2.35: Movement with normalization and `pygame.clock.tick()` (float) using constant speed but different fps (from left to right: 10, 30, 60, 120, 240, 300, 600)

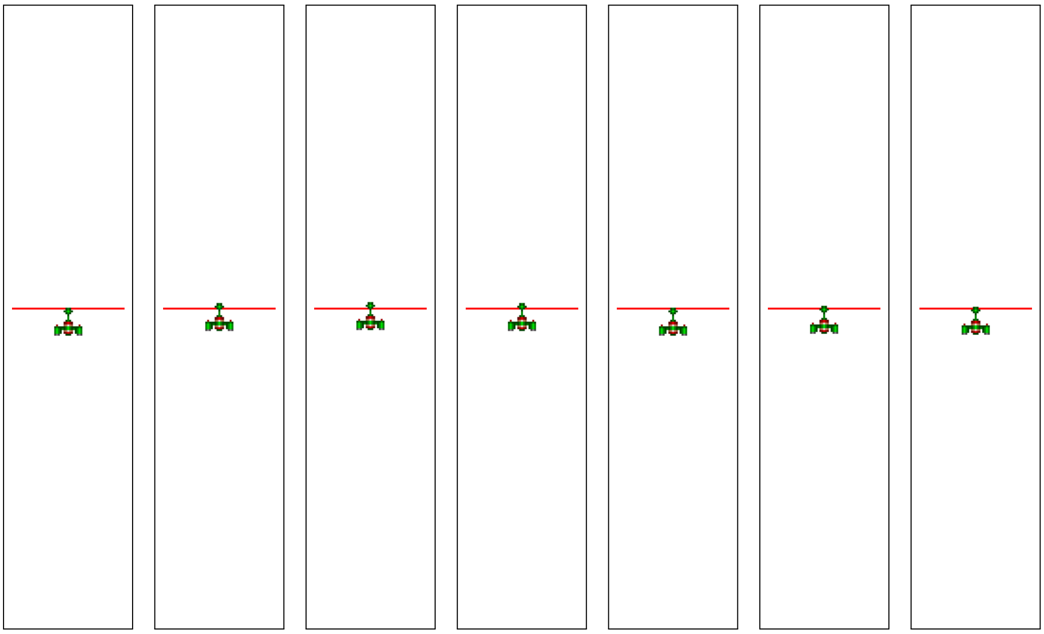

Figure 2.36: Movement with normalization and `time.time()` using constant speed but different fps (from left to right: 10, 30, 60, 120, 240, 300, 600), Version 3

```
        screen.blit(defender_image, defender_rect)
        window.flip()
        clock.tick(cfg.FPS)
        time_current = time()  # remember stop time
        cfg.DELTATIME = time_current - time_previous  # Time consumption
        time_previous = time_current  # New start time
    pygame.quit()
```

2.4.4 What was new?

- The position of an object is stored in a `Rect` or `FRect` object. In each frame, the position is checked and modified if necessary. When the screen is updated, this creates the impression of movement. The result of a movement is usually first stored temporarily in a variable and checked before it is applied as the new position.
- The direction of movement is encoded by the sign, and the speed by the value of the speed variable. Horizontal and vertical movement are handled separately.
- To become independent of the actual frame rate, a correction factor (delta time) must be used when calculating the new position. This value can either be computed manually or obtained from a call to `pygame.time.Clock.tick()`.

The following Pygame elements were introduced

- `pygame.rect.FRect`: https://pyga.me/docs/ref/rect.html
- `pygame.rect.FRect.move()`: https://pyga.me/docs/ref/rect.html#pygame.Rect.move
- `pygame.rect.Rect`: https://pyga.me/docs/ref/rect.html
- `pygame.rect.Rect.move()`: https://pyga.me/docs/ref/rect.html#pygame.Rect.move
- `pygame.Surface.get_rect()`: https://pyga.me/docs/ref/surface.html#pygame.Surface.get_rect
- `pygame.time.get_ticks()`: https://pyga.me/docs/ref/surface.html#pygame.time.get_ticks

- `pygame.math.Vector2`:
  https://pyga.me/docs/ref/math.html#pygame.math.Vector2
- `pygame.math.Vector3`:
  https://pyga.me/docs/ref/math.html#pygame.math.Vector3

2.4.5 Homework

1. Use `centerx`, `centery`, or `center` instead of `top` and `left`. Does it work? Do you notice any remarkable differences?
2. Create an application in which two identical objects travel the same horizontal distance of 800 px. One object uses position values stored as `int`, the other uses position values stored as `float`. Both should move at a speed of 50 px/s. What can you observe?
3. Create an application where the speed is not constant. Four objects should travel the same horizontal distance of 800 px in parallel, just like before. The behavior should be as follows:
   a) Object 1: It continuously accelerates and reaches its maximum speed at the right end.
   b) Object 2: It accelerates over time, reaches its maximum speed at the midpoint of the distance, and then slows down again. At the right end, its speed is 0 px/s. The increase and decrease in speed are linear.
   c) Object 3: It accelerates over time, reaches its maximum speed at the midpoint of the distance, and then slows down again. At the right end, its speed is 0 px/s. The increase and decrease in speed follow a sine curve on $[0, \pi]$.
   d) Object 4: Like object 3, but a variable controls how many intervals of $[0, \pi]$ are completed before reaching the right end. For example, the object speeds up and slows down again 5 times.
4. Something for the ambitious among you: All four objects reach the right edge at the same time.

2.5 Class Sprite

2.5.1 Introduction

So ... where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/05%20Sprite/invader

In section 2.4.3.1 on page 64, it became apparent that many variables start with `defender_`. In other words, they are attributes of a single entity and almost demand to be expressed as a class.

This class should contain all information related to updating and rendering the bitmap. Some elements, such as `defender_image` and `defender_rect`, seem to play a role in virtually every bitmap processing task. Furthermore, every bitmap will require some form of state update and screen output. In fact,

Pygame already provides a class that offers exactly such a framework

`pygame.sprite.Sprite`.

Let us therefore define the `Defender` class as a subclass of `Sprite` (line 7).

Listing 2.52: Sprites (1), Version 1.0

```
7  class Defender(pygame.sprite.Sprite):              # Child class of Sprite
8
9      def __init__(self) -> None:                    # Constructor
10         super().__init__()
11         self.image = pygame.image.load("images/defender01.png").convert_alpha()
12         self.image = pygame.transform.scale(self.image, (30, 30))
13         self.rect = pygame.FRect(self.image.get_rect())
14         self.rect.centerx = cfg.WINDOW.centerx
15         self.rect.bottom = cfg.WINDOW.bottom - 5
16         self.speed = 300
17
18     def update(self) -> None:                      # State update
19         newpos = self.rect.move(self.speed * cfg.DELTATIME, 0)
20         if newpos.right >= cfg.WINDOW.right:
21             self.change_direction()
22             newpos.right = cfg.WINDOW.right
23         elif newpos.left <= cfg.WINDOW.left:
24             self.change_direction()
25             newpos.left = cfg.WINDOW.left
26         self.rect = newpos
27
28     def draw(self, screen: pygame.Surface) -> None: # Drawing
29         screen.blit(self.image, self.rect)
30
31     def change_direction(self) -> None:            # OO style
32         self.speed *= -1
```

The lines in the constructor (line 9ff.) correspond to those of the previous version. Only the prefix `defender_` is replaced by `self.`, which turns the variables into attributes of the class. You should have no difficulty understanding these changes.

Every subclass of `Sprite` must provide two attributes: `rect` and `image`. These two attributes are accessed by the already predefined solutions for collision detection, rendering, and related tasks. We will see their usefulness later on. self.rect self.image

In line 18ff., the boundary collisions and state changes are implemented. One detail that stands out is the computation of the new position using `move()`.

New is the call to the method `change_direction()`. This method (line 31) is more *object-oriented* than the previous version. In object-oriented programming, algorithms are not implemented directly; instead, messages are sent to objects, which then handle the details internally – in a way that is not visible from the outside. In this case, this means that instead of performing the direction change directly at the relevant point, I send a message to the object telling it that the direction needs to be changed.

With the method `draw()` in line 28, the screen output is encapsulated.

Listing 2.53: Sprites (2), Version 1.0

```
35 def main():
36     pygame.init()
37     window = pygame.Window(size=cfg.WINDOW.size, title="Sprite", position=(10, 50))
38     screen = window.get_surface()
39     clock = pygame.time.Clock()
40     defender = Defender()                          # Create object
41 
42     time_previous = time()
43     running = True
44     while running:
45         # Events
46         for event in pygame.event.get():
47             if event.type == pygame.QUIT:
48                 running = False
49 
50         # Update
51         defender.update()                          # Call
52 
53         # Draw
54         screen.fill("white")
55         defender.draw(screen)                      # Call
56         window.flip()
57 
58         clock.tick(cfg.FPS)
59         time_current = time()
60         cfg.DELTATIME = time_current - time_previous
61         time_previous = time_current
62     pygame.quit()
```

Using the `Defender` class has now become straightforward. In line 40, an object of the class is created. In line 51, `update()` is called, and in line 55, `draw()` is executed.

One advantage of the new architecture is the improved clarity and readability of the main program. By following naming conventions (descriptive class and method names), the overall control flow becomes clearer and is no longer obscured by implementation details.

I now want to make use of the capabilities of the `Sprite` class so that boundary collision checks no longer have to be implemented manually.

move_ip()

Let us get started: Since we want to organize collision detection differently, we first simplify `update()` again. We now only compute the new position. In doing so, the method `pygame.FRect.move_ip()` is introduced in line 19. It works like `move()`, but in this case the modification is applied directly to the rectangle; `ip` stands for *in place*. With `move()`, the original rectangle remains unchanged.

Listing 2.54: Sprites (1), Version 1.1

```
18      def update(self) -> None:
19          self.rect.move_ip(self.speed * cfg.DELTATIME, 0)  # Move in-place
```

To make the boundaries visible and to better recognize collisions, the edges are now replaced by two stone walls on the left and right. These bitmaps are also implemented as subclasses of `pygame.sprite.Sprite`. Since the state of the two walls never changes, the implementation of `update()` can be omitted.

Listing 2.55: Sprites (2), Version 1.1

```
class Border(pygame.sprite.Sprite):

    def __init__(self, leftright: str) -> None:
        super().__init__()
        self.image = pygame.image.load("images/brick01.png").convert_alpha()
        self.image = pygame.transform.scale(self.image, (35, cfg.WINDOW.height))
        self.rect = self.image.get_rect()
        if leftright == "right":
            self.rect.left = cfg.WINDOW.width - self.rect.width

    def draw(self, screen: pygame.Surface) -> None:
        screen.blit(self.image, self.rect)
```

I now create the two boundaries:

Listing 2.56: Sprites (3), Version 1.1

```
    border_left = Border("left")
    border_right = Border("right")
```

So far, everything has been easy.

Listing 2.57: Sprites (4), Version 1.1

```
        # Update
        if pygame.sprite.collide_rect(defender, border_left):
            defender.change_direction()
        elif pygame.sprite.collide_rect(defender, border_right):
            defender.change_direction()
        defender.update()
```

collide_rect()

What is happening here? The method `pygame.sprite.collide_rect()` checks whether the rectangles of two `Sprite` objects collide. This means that I no longer have to manually check the left and right boundaries myself.

Here, the collision of a single object with both boundaries – or more generally, with many `Sprite` objects – is tested.

Spritegroups

In practice, sprites rarely exist on their own; they are usually organized into groups. This concept is also built into Pygame and leads to further simplifications.

Listing 2.58: Sprites (1), Version 1.2

```
36 def main():
37     pygame.init()
38     window = pygame.Window(size=cfg.WINDOW.size, title="Sprite", position=(10, 50))
39     screen = window.get_surface()
40     clock = pygame.time.Clock()
41 
42     defender = pygame.sprite.GroupSingle(Defender())
43     all_border = pygame.sprite.Group()
44     all_border.add(Border("left"))
45     all_border.add(Border("right"))
46 
47     time_previous = time()
48     running = True
49     while running:
50         # Events
51         for event in pygame.event.get():
52             if event.type == pygame.QUIT:
53                 running = False
54 
55         # Update
56         if pygame.sprite.spritecollide(defender.sprite, all_border, False):  # !
57             defender.sprite.change_direction()                               #
58         defender.update()
59 
60         # Draw
61         screen.fill((255, 255, 255))
62         defender.draw(screen)
63         all_border.draw(screen)                                              # In one go
64         window.flip()
65 
66         clock.tick(cfg.FPS)
67         time_current = time()
68         cfg.DELTATIME = time_current - time_previous
69         time_previous = time_current
70     pygame.quit()
```

The defender is no longer addressed directly, but is instead packed into a luxury box. I will come back to this later. The two `Border` objects are no longer stored in two separate object variables either; instead, they are placed into a luxury box as well: a `pygame.sprite.Group`. Here, I could also store additional boundaries or walls. From the point of view of the game logic, they could then all be processed together in one go. This becomes clear in this mini example in two places.

Group

The first location is line 56, where a different variant of collision detection is used:

spritecol-lide()

pygame.sprite.spritecollide(). The first parameter is a *single* Sprite object – in our case, the defender. The second parameter is a sprite group containing all Border objects. Thus, the defender is checked for collisions with all members of the group. This only works if all sprites provide a Rect or FRect object named rect as an attribute. The third parameter – False in this example – controls whether the colliding sprite should be removed from the group. This feature is quite useful in games, for example when rocks that are shot by a spaceship should be deleted.

The second location is line 63. Here, draw() is no longer called for each object individually, but once for the entire group. When using this service, the draw() method can be removed from your own classes (here Border and Defender), which simplifies things even further.

GroupSin-gle

It therefore seems like a good idea to pack sprites into such luxury boxes. But what about the defender? To take advantage of sprite groups, you can also create groups that contain only a single element. To allow these groups to work more efficiently – after all, it is known that they contain only one element – Pygame provides the special case pygame.sprite.GroupSingle. Since there is often a need to access the single Sprite object of the *group*, this group provides the additional attribute sprite (see line 56f.).

2.5.2 More Input

2.5.2.1 OO Issues

So . . . where are the files?

- https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/05%20Sprite/invader/v04
- https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/05%20Sprite/invader/v05

I want to pursue my object-oriented approach even further and also turn the main program into a Game class. What is important to me is to establish a sense of structural discipline right from the beginning. The longer you stay in software development, the more you will grow fond of terms like *order* and *structure*. They help you avoid losing the thread, even in more complex games. A particularly helpful concept here is the Single Responsibility Principle (SRP).

Single Responsibility Principle (SRP)

Each class or function should have exactly one responsibility. It should focus on that single task and do it well. A solution to a specific problem should be encapsulated in one class or one method, and changes to that responsibility should affect only that part of the code.

Listing 2.59: Game-Klasse

```python

    def __init__(self) -> None:
        pygame.init()
        self.window = pygame.Window(size=cfg.WINDOW.size, title="Sprite", position=(10, 50))
        self.screen = self.window.get_surface()
        self.clock = pygame.time.Clock()

        self.defender = pygame.sprite.GroupSingle(Defender())
        self.all_border = pygame.sprite.Group()
        self.all_border.add(Border("left"))
        self.all_border.add(Border("right"))
        self.running = False

    def run(self) -> None:
        time_previous = time()
        self.running = True
        while self.running:
            self.watch_for_events()
            self.update()
            self.draw()
            self.clock.tick(cfg.FPS)
            time_current = time()
            cfg.DELTATIME = time_current - time_previous
            time_previous = time_current
        pygame.quit()

    def watch_for_events(self) -> None:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                self.running = False

    def update(self) -> None:
        if pygame.sprite.spritecollide(self.defender.sprite, self.all_border, False):
            self.defender.sprite.change_direction()  # I don't like this!
        self.defender.update()

    def draw(self) -> None:
        self.screen.fill((255, 255, 255))
        self.defender.draw(self.screen)
        self.all_border.draw(self.screen)
        self.window.flip()
```

An example of this last point is the design of the `Game` class. Here, the source code is no longer simply placed in `__main__`, but is instead encapsulated, structured, and thus made flexibly reusable. A clear example of the SRP can be seen in the methods `watch_for_events()`, `update()`, and `draw()`. It is simply not the responsibility of `run()` to organize everything. From the perspective of the main loop, it is irrelevant which events are queried or how they are processed. All that matters is that events are handled once per frame. Likewise, `run()` should not be concerned with the order in which sprites are drawn to the screen. That task belongs to the `draw()` method. The `run()` method merely ensures that sprites first update their state and that rendering happens afterwards.

One aspect remains that I would still like to address here: the call to `change_direction()` in line 70 does not appeal to me. It violates object-oriented design rules, specifically the Liskov Substitution Principle (LSP).

Liskov Substitution Principle (LSP)

A principle of object-oriented programming stating that objects of a derived class must be usable anywhere an object of the base class is expected, without altering the correct behavior of the program. Formulated by Barbara Liskov in 1987. The LSP ensures that inheritance does not introduce unexpected side effects and that class hierarchies remain consistent.

The sprite group is a collection of `Sprite` objects. However, the class `pygame.sprite.Sprite` does not define a method called `change_direction()`. Calling such a method here is therefore not entirely clean. Python does not have a problem with this, but that should not be the benchmark.

update()

A better approach is to adapt the `update()` method instead. If you take a closer look at the signature of `pygame.sprite.Sprite.update()`, you will see that it is designed to accept freely definable parameters. I have developed the habit of using a parameter named `action` to trigger specific behavior in subclasses. With this approach, `change_direction()` is called internally from `update()` (see line 23) rather than being invoked from the outside.

Listing 2.60: `Defender.update()`

```
20        if "action" in kwargs.keys():
21            if kwargs["action"] == "newpos":         # Compute new position
22                self.rect.move_ip(self.speed * cfg.DELTATIME, 0)
23            elif kwargs["action"] == "direction":   # Change direction
24                self.change_direction()
```

The call then takes place indirectly in source code 2.61 at line 75 by using the argument passed to the method.

Note: This also complies with the object-oriented design principle Don't ask – tell.

Listing 2.61: `Game.update()`

```
74        if pygame.sprite.spritecollide(self.defender.sprite, self.all_border, False):
75            self.defender.update(action="direction")  # Better (don't aks - tell!)
76        self.defender.update(action="newpos")
```

2.5.2.2 Add Sprite Objects to a Group Right Away

So ... where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/05%20Sprite/invadergroup/v01

It is often very convenient to assign a sprite object to a group already at the time it is created. To do this, the signature of `__init__()` only needs to be adjusted accordingly.

The parameter

`*groups`

together with the corresponding call to the constructor of the superclass

`super().__init__(*groups)`

ensures that the sprite object is immediately added to the given group or groups. Below is the complete source code. In line 61, the group is then simply passed to the constructor as the last argument.

Listing 2.62: Add Sprite Objects to a Group Right Away

```
1  from random import randint
2  from time import time
3  from typing import Any
4
5  import pygame
6
7
8  class Ship(pygame.sprite.Sprite):
9      def __init__(self, position:tuple[int,int], *groups: pygame.sprite.AbstractGroup[Any])
           -> None:
10         super().__init__(*groups)
11         self.image = pygame.image.load("images/defender01.png").convert_alpha()
12         self.image = pygame.transform.scale(self.image, (30, 30))
13         self.rect = pygame.FRect(self.image.get_rect())
14         self.rect.left = position[0]
15         self.rect.bottom = position[1]
16         self.speed = -300
17
18     def update(self, *args: Any, **kwargs: Any) -> None:
19         self.rect.move_ip(0, self.speed * Game.DELTATIME)
20         self.rect.clamp_ip(Game.WINDOW)
21         return super().update(*args, **kwargs)
22
23
24
25 class Game():
26     FPS = 60
27     DELTATIME = 1.0 / FPS
28     SPWAN = 15
29     WINDOW = pygame.Rect(0,0,300,600)
30
31     def __init__(self) -> None:
32         pygame.init()
33         self.window = pygame.Window(size=Game.WINDOW.size, title="Spritegroup")
```

```
        self.screen = self.window.get_surface()
        self.clock = pygame.time.Clock()
        self.ships = pygame.sprite.Group()
        self.running = True
        self.counter = 0


    def run(self) -> None:
        time_previous = time()
        while self.running:
            self.watch_for_events()
            self.update()
            self.draw()
            self.clock.tick(Game.FPS)
            time_current = time()
            Game.DELTATIME = time_current - time_previous
            time_previous = time_current

    def watch_for_events(self) -> None:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                self.running = False

    def update(self) -> None:
        self.counter += 1
        if self.counter > Game.SPWAN:
            self.counter = 0
            Ship((randint(0, 300-30), 600), self.ships) #
        self.ships.update()

    def draw(self) -> None:
        self.screen.fill((255, 255, 255))
        self.ships.draw(self.screen)
        self.window.flip()
```

2.5.2.3 Delete Sprites from Groups

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/05%20Sprite/invadergroup/v02

If you follow the program logic in source code 2.62 on the previous page, you will notice that a spaceship is created at the bottom edge four times per second and then flies upward. Once it reaches the top, it simply stops.

The latter behavior is usually rather pointless. A more natural approach is to remove all spaceships that have crossed an upper boundary. Think, for example, of projectiles that leave the playfield.

kill()

This can be achieved by calling `pygame.sprite.Sprite.kill()`, which instructs Pygame to remove the `Sprite` object from *all* sprite groups. If no further references to the `Sprite` object exist, it will then be deleted by Python's garbage collector.

You can see an example of this in source code 2.63. Once the spaceship has completely passed the midpoint of the window, it is removed.

Listing 2.63: Kill a Sprite

```
    def update(self, *args: Any, **kwargs: Any) -> None:
        self.rect.move_ip(0, self.speed * Game.DELTATIME)
        if self.rect.bottom < Game.WINDOW.centery:
            self.kill()
        return super().update(*args, **kwargs)
```

2.5.3 What was new?

From a behavioral point of view: *nothing at all*. The existing application has merely been embedded into a flexible framework.

The following Pygame elements were introduced

- `pygame.Rect.move()`: https://pyga.me/docs/ref/rect.html#pygame.Rect.move
- `pygame.Rect.move_ip()`: https://pyga.me/docs/ref/rect.html#pygame.Rect.move_ip
- `pygame.sprite.Group`: https://pyga.me/docs/ref/sprite.html#pygame.sprite.Group
- `pygame.sprite.GroupSingle`: https://pyga.me/docs/ref/sprite.html#pygame.sprite.GroupSingle
- `pygame.sprite.GroupSingle.sprite`: https://pyga.me/docs/ref/sprite.html#pygame.sprite.GroupSingle
- `pygame.sprite.Sprite`: https://pyga.me/docs/ref/sprite.html#pygame.sprite.Sprite
- `pygame.sprite.Sprite-kill()`: https://pyga.me/docs/ref/sprite.html#pygame.sprite.Sprite.kill
- `pygame.sprite.collide_rect()`: https://pyga.me/docs/ref/sprite.html#pygame.sprite.collide_rect
- `pygame.sprite.spritecollide()`: https://pyga.me/docs/ref/sprite.html#pygame.sprite.spritecollide

2.5.4 Homework

1. Modify the source code from section 2.3.2.3 on page 54 so that `Meadow`, `Sky`, `Tree`, `House`, and `Sun` are subclasses of `pygame.sprite.Sprite`.
2. Manage all `Sprite` classes that implement an `update()` method in one `pygame.sprite.Group` object, and store the others in a separate group.

2.6 Handling Keyboard Input

2.6.1 Introduction

So ... where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/06%20Keyboard/keyboard/v01

I do not intend to cover the keyboard exhaustively here, but merely to illustrate the basic principle. The movement of the spaceship is controlled by keyboard events. Every time I press a arrow key – left, right, up, down – the spaceship moves in the corresponding direction. If I release the arrow key, the spaceship stops. The game can now also be exited using the Escape key (Boss key).

As a first step, a dictionary of possible directions is created in `config.py`.

Vector2D

These directions are managed as `Vector2` objects , since they are easier to use for mathematical operations.

Listing 2.64: Control Direction by Keys (1), `config.py`

```python
import pygame

WINDOW = pygame.Rect((0, 0), (300, 300))
FPS = 60
DELTATIME = 1.0 / FPS
DIRECTIONS = {
    "stop": pygame.math.Vector2(0, 0),
    "right": pygame.math.Vector2(1, 0),
    "left": pygame.math.Vector2(-1, 0),
    "up": pygame.math.Vector2(0, -1),
    "down": pygame.math.Vector2(0, 1),
}
```

Next, we prepare the `Defender` class or modify it slightly (source code 2.65 on the following page). The sprite is no longer placed at the bottom but centered (line 15), and the spaceship should now also be able to move vertically. For this, we either need two separate variables or a `Vector2` object. I choose a `Vector2` object, where the first component represents the horizontal direction vector and the second the vertical direction. Each direction vector is set according to the semantics introduced earlier.

clamp_ip()
clamp()

I would like to draw special attention to line 23. The methods `clamp_ip()` as well as `clamp()` provide a very convenient shortcut in programming. Both methods check whether the inner rectangle has crossed the boundary of the outer rectangle on any side and, if necessary, move it back to the edge of the outer rectangle.

Here is an equivalent check without using `clamp()`, shown only for the left edge of the outer rectangle:

```
        if inner_rect.right < outer_rect.left:
                inner_rect.right = outer_rect.left + 1
```

The method `clamp()` performs this kind of check for all sides of the inner rectangle. Using `clamp()` or `clamp_ip()`, you can therefore ensure that a sprite never leaves a defined play area.

The difference between the two methods is that `clamp()` does not modify the values of the inner rectangle but instead returns a new, adjusted rectangle, whereas `clamp_ip()` modifies the values of the inner rectangle directly.

Listing 2.65: Control Direction by Keys (2), Class `Defender`

```
class Defender(pygame.sprite.Sprite):

    def __init__(self) -> None:
        super().__init__()
        self.image = pygame.image.load("images/defender01.png").convert_alpha()
        self.image = pygame.transform.scale(self.image, (30, 30))
        self.rect = pygame.FRect(self.image.get_rect())
        self.rect.center = cfg.WINDOW.center            #
        self.speed = 100
        self.direction = cfg.DIRECTIONS["stop"]

    def update(self, *args: Any, **kwargs: Any) -> None:
        if "action" in kwargs.keys():
            if kwargs["action"] == "move":
                self.rect.move_ip(self.direction.elementwise() * self.speed *
                    cfg.DELTATIME)
                self.rect.clamp_ip(cfg.WINDOW)      # Keep inside window
        elif "direction" in kwargs.keys():
            self.direction = cfg.DIRECTIONS[kwargs["direction"]]
```

KEY-DOWN KEYUP

key

Let us now turn to the actual handling of keyboard input: Pressing a key can trigger the event types `pygame.KEYDOWN` or `pygame.KEYUP`. In our example (line 57), we are interested in which key is *pressed*, so we use `KEYDOWN`. After that, we can determine which key was pressed via `pygame.event.key`. For this purpose, Pygame provides a set of predefined constants in `pygame.key` (see table 2.7 on page 101 and table 2.8 on page 104).

Listing 2.66: Control Direction by Keys (3)), `Game.watch_for_events()`

```
54         for event in pygame.event.get():
55             if event.type == pygame.QUIT:
56                 self.running = False
57             elif event.type == pygame.KEYDOWN:          # Key pressed
58                 if event.key == pygame.K_ESCAPE:        #
59                     self.running = False
60                 elif event.key == pygame.K_RIGHT:       # Arrows
61                     self.defender.update(direction="right")
62                 elif event.key == pygame.K_LEFT:
63                     self.defender.update(direction="left")
64                 elif event.key == pygame.K_UP:
65                     self.defender.update(direction="up")
```

```
66                elif event.key == pygame.K_DOWN:
67                    self.defender.update(direction="down")
68            elif event.type == pygame.KEYUP:           # Key released
69                if event.key in (pygame.K_RIGHT, pygame.K_LEFT,
70                                 pygame.K_UP, pygame.K_DOWN):
71                    self.defender.update(direction="stop")
```

K_ESCAPE

Let us start with the boss key. In line 58, the constant `K_ESCAPE` is used to check whether the pressed key is the Esc. As with clicking the window's close button, the flag of the main program loop is then simply set to `False`. Try it out!

K_LEFT
K_RIGHT
K_UP
K_DOWN

After that, the four arrow keys are handled starting at line 60 ff. Using `K_LEFT`, `K_RIGHT`, `K_UP`, and `K_DOWN`, the corresponding arrow key is checked and the appropriate message is sent to the defender.

If one of the arrow keys is released (`pygame.KEYUP` in line 68), the spaceship is stopped.

This should be sufficient for now. The keyboard is only one possible way to control a game. Mouse input, game controllers, and joysticks are also supported in Pygame.

2.6.2 More Input

2.6.2.1 Example: Shift and Related Keys

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/06%20Keyboard/keyboard/v02

With the help of ⇧, Ctrl, or other modifier keys, additional meanings can be assigned to regular keys. But how can we detect that such keys are being pressed? In the source code of the previous example, only a reaction to *one* key press per condition is implemented.

I will now extend the example so that pressing a movement key together with the left Shift key makes the spaceship move faster, while pressing it together with the right Shift key makes it move slower.

To do this, we first adapt the `update()` method of the `Defender`. As you can see, the signals to speed up or slow down can now be handled as well. It is also possible to reset the speed to its normal value. The concrete values are more or less arbitrary, but they are chosen so that the difference in speed is easy to perceive.

Listing 2.67: Control Direction by Keys (4)), `Defender.update()`

```
        if "action" in kwargs.keys():
            if kwargs["action"] == "move":
                    self.rect.move_ip(self.direction.elementwise() * self.speed *
                        cfg.DELTATIME)
                    self.rect.clamp_ip(cfg.WINDOW)
        elif "direction" in kwargs.keys():
            self.direction = cfg.DIRECTIONS[kwargs["direction"]]
        elif "speed" in kwargs.keys():
            if kwargs["speed"] == "faster":
                self.speed = 300
            elif kwargs["speed"] == "slower":
                self.speed = 10
            elif kwargs["speed"] == "normal":
                self.speed = 100
```

event.mod

KMOD_LSHIFT
KMOD_RSHIFT

Now we adapt `watch_for_events()`. To be able to detect simultaneous key presses, a slightly different mechanism must be used. Internally, a `modifier` bit is set. This bitmask must be checked using the binary AND operation. In our example, this is done in line 68 using `KMOD_LSHIFT` and in line 70 using `KMOD_RSHIFT`.

Listing 2.68: Control Direction by Keys (5)), `Game.watch_for_events()`

```
61         for event in pygame.event.get():
62             if event.type == pygame.QUIT:
63                 self.running = False
64             elif event.type == pygame.KEYDOWN:
65                 if event.mod == pygame.KMOD_NONE:                    # No modifier keys
66                     self.defender.update(speed="normal")
67                 else:
68                     if event.mod & pygame.KMOD_LSHIFT:               # Left shift pressed
69                         self.defender.update(speed="faster")
70                     if event.mod & pygame.KMOD_RSHIFT:               # Right shift pressed
71                         self.defender.update(speed="slower")
72                 if event.key == pygame.K_ESCAPE:
73                     self.running = False
74                 elif event.key == pygame.K_RIGHT:
75                     self.defender.update(direction="right")
76                 elif event.key == pygame.K_LEFT:
77                     self.defender.update(direction="left")
78                 elif event.key == pygame.K_UP:
79                     self.defender.update(direction="up")
80                 elif event.key == pygame.K_DOWN:
81                     self.defender.update(direction="down")
82             elif event.type == pygame.KEYUP:
83                 self.defender.update(speed="normal")
84                 if event.key in (pygame.K_RIGHT, pygame.K_LEFT,
85                                  pygame.K_UP, pygame.K_DOWN):
86                     self.defender.update(direction="stop")
```

Querying modifier keys

You cannot check the modifier keys directly. Instead, you have to use a binary AND operation to determine whether the corresponding bit is set in a mask or not.

In this way, the state of multiple modifier keys can be encoded and queried within a single integer. For example, if you want to check whether the keys ⇧ and Alt are pressed at the same time, you can do so as follows:

```
        if (event.mod & pygame.KMOD_SHIFT) and (event.mod & pygame.KMOD_ALT):
                ...
```

Or

```
        if (event.mod & (pygame.KMOD_SHIFT event.mod | pygame.KMOD_ALT)):
            ...
```

As a complement, it should also be noted that `KMOD_NONE` can be used to check whether no modifier key is pressed at all (see line 65).

KMOD-_NONE

2.6.2.2 In Which Window was the Key Pressed?

So ... where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/06%20Keyboard/keyboard/v03

When working with multiple windows, it is certainly important to determine in which window a keyboard input was made. There are many ways to achieve this, but the two approaches presented here are probably the most straightforward.

As a basis, we use the example from section 2.1.2.1 on page 21. Two windows are displayed next to each other.

The variables `window_first` and `window_second` are handle to the two windows. Internally, these handles are essentially memory addresses (Random Access Memory (RAM)) through which the windows can be accessed via their properties and methods. Naturally, this address is unique for each window.

When a keyboard event is triggered, the event also carries a handle to the window in which it occurred. This handle can be accessed via `event.window`, which is exactly what happens in line 26 and line 29. By comparing these handles, it is therefore possible to determine unambiguously in which window the event was triggered.

event.window

Alternatively, one can use the `id` attribute. This value is also unique for each window and reflects the order in which the windows were created.

Window.id

Listing 2.69: In which window was the key pressed?

```python
def main():
    pygame.init()
    window_first = pygame.Window(size=(300, 50),
        title="Main Window",
        position=(500, 50))
    window_second = pygame.Window(size=(300, 50),
        title="Side Window",
        position=(820, 50))
    screen_first = window_first.get_surface()
    screen_second = window_second.get_surface()
    clock = pygame.time.Clock()


    running = True
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
            elif event.type == pygame.WINDOWCLOSE:
                running = False
                event.window.destroy()
            elif event.type == pygame.KEYDOWN:
                if event.window == window_first:    # Check which window the event belongs to
                    window_id = window_first.id
                    event.window.title = "Main Window (Key Pressed: '" +
                        pygame.key.name(event.key) + "')"
                elif event.window == window_second: #
                    window_id = window_second.id
                    event.window.title = "Side Window (Key Pressed: '" +
                        pygame.key.name(event.key) + "')"
                else:
                    window_id = None

                print(f"ID {window_id}: {event.window.title}")
        if running:
            screen_first.fill((0, 255, 0))
            window_first.flip()
            screen_second.fill((255, 0, 0))
            window_second.flip()
            clock.tick(60)

    pygame.quit()
```

Running the program produces the following console output when the corresponding keys are pressed:

```
ID 1: Main Window (Key Pressed: 'a')
ID 1: Main Window (Key Pressed: 's')
ID 1: Main Window (Key Pressed: '5')
ID 1: Main Window (Key Pressed: 'j')
ID 2: Side Window (Key Pressed: 'm')
ID 2: Side Window (Key Pressed: 'space')
ID 2: Side Window (Key Pressed: '3')
ID 2: Side Window (Key Pressed: '4')
```

2.6.2.3 Example: Visualizing the Keyboard

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/06%20Keyboard/typewriter

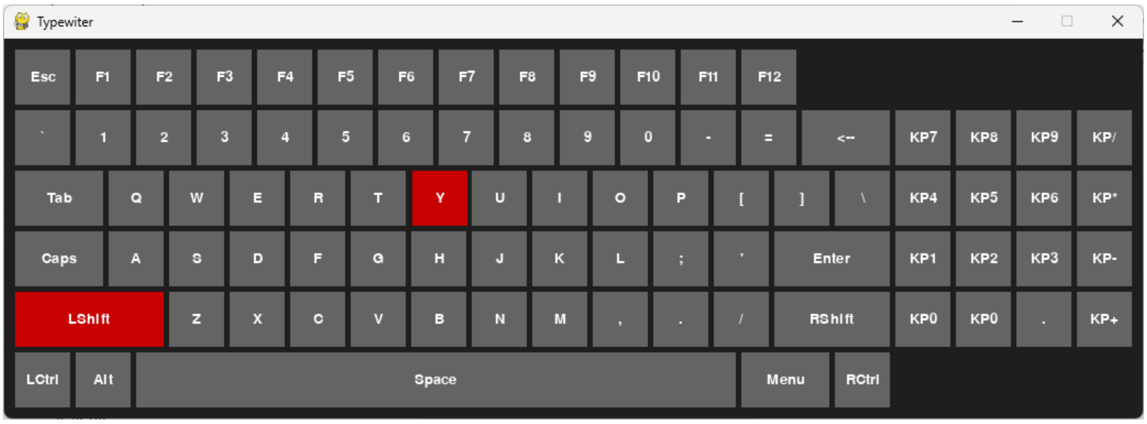

Figure 2.37: Typewriter

Just as a small exercise, let us write a simple program that visualizes a key press (see figure 2.37). For this purpose, I want to define the keyboard layout in `config.py` so that it can be easily adapted to different variants – here a simple U.S. keyboard layout.

Listing 2.70: Typewriter, `config.py`

```python
import pygame

WINDOW = pygame.Rect((0, 0), (1035, 345))
FPS = 30

ROWS = [
    ['Esc'] + [f'F{i}' for i in range(1, 13)],
    ['`','1','2','3','4','5','6','7','8','9','0','-','=','<--', 'KP7','KP8','KP9','KP/'],
    ['Tab','Q','W','E','R','T','Y','U','I','O','P','[',']','\\', 'KP4','KP5','KP6','KP*'],
    ['Caps','A','S','D','F','G','H','J','K','L',';','\'','Enter', 'KP1','KP2','KP3','KP-'],
    ['LShift','Z','X','C','V','B','N','M',',','.','/','RShift', 'KP0','KP0','.','KP+'],
    ['LCtrl','Win','Alt','Space','AltGr','Menu','RCtrl']
]

KEY = {'width': 50, 'height': 50, 'spacing': 5}
```

The constructor of `KeySprite` receives its label (i.e., its meaning), its value according to table 2.7 on page 101, and its position. Based on this data, the sprite, the text label, its position, and its size are computed and processed.

A *normal* key has exactly the width and height defined in `config.py` by the dictionary

KEY. Therefore, in line 24 the width is multiplied by the factor 1, and no additional spacing is needed.

Other keys have different widths; for example, the Space key has a width of 10 keys. This is determined using a case distinction and stored in `factor_width`.

From this factor, we can also determine how many gaps between keys must be included in the width calculation. For a normal key, no additional gap is consumed. For two keys, there is exactly one gap between them; for three keys, there are two gaps, and so on. So the number of gaps is the number of normal key widths minus 1. However, since we also use factor values such as 1.5, we must always compute the next higher integer. The result is the calculation in line 26.

In the next line, the total width of the key can then be computed based on the two factors.

The attribute `pressed` is a flag that stores whether the key is currently pressed or not. After that, the rectangle is created using the given position, and the label is rendered. Text output using fonts will be explained in more detail in a later section.

In `update()`, the key is filled either gray or red. From a performance point of view, this is not ideal; strictly speaking, we would only need to redraw the key when the state of `pressed` changes, but let us keep it simple here for now.

Listing 2.71: Typewriter, Class `KeySprite`

```
 8 class KeySprite(pygame.sprite.Sprite):
 9     def __init__(self, label: str, key: int, left: int, top: int) -> None:
10         super().__init__()
11         self.font = pygame.font.SysFont(None, 20)
12         self.label = label
13         self.key = key
14         if label == 'Enter':
15             factor_width = 2.0
16         elif label == 'Tab' or label == 'Caps' or label == '<--' or label == 'Menu':
17             factor_width = 1.5
18         elif label == 'LShift':
19             factor_width = 2.5
20         elif label == 'RShift':
21             factor_width = 2.0
22         elif label == 'Space':
23             factor_width = 10.0
24         else:                                        # Normal key
25             factor_width = 1.0
26         factor_spacing = ceil(factor_width - 1)      # Used space
27         width = int(factor_width * cfg.KEY['width'] + (factor_spacing * cfg.KEY['spacing']))
28         self.image = pygame.Surface((width, cfg.KEY['height']))
29         self.pressed = False
30         self.rect = self.image.get_rect(topleft=(left, top))
31         self.txt_surf = self.font.render(label, True, (255, 255, 255))
32         self.txt_rect = self.txt_surf.get_rect()
33         self.txt_rect.center = self.image.get_rect().center
34         self.update()
35 
36     def update(self) -> None:
37         if self.pressed:
38             self.image.fill((200, 0, 0))
39             self.image.blit(self.txt_surf, self.txt_rect)
40         else:
```

```
        self.image.fill((100, 100, 100))
        self.image.blit(self.txt_surf, self.txt_rect)
```

The constructor and the `run()` method of `Game` should be self-explanatory. In the dictionary `keyboard`, all keys are stored using the identifiers defined in table 2.7 on page 101 as dictionary keys.

Listing 2.72: Typewriter, `Game.init()` and `Game.run()`

```
class Game(object):

    def __init__(self) -> None:
        pygame.init()
        self.window = pygame.Window(size=cfg.WINDOW.size, title="Typewiter")
        self.screen = self.window.get_surface()
        self.clock = pygame.time.Clock()
        self.running = False
        self.all_sprites = pygame.sprite.Group()
        self.keyboard = self.generate_sprites()

    def run(self) -> None:
        self.running = True
        while self.running:
            self.watch_for_events()
            if self.running:
                self.update()
                self.draw()
                self.clock.tick(cfg.FPS)
        pygame.quit()
```

The creation of this dictionary takes place in the method `generate_sprites()`. First, I define the starting position of the first key. It is placed in the upper-left corner and starts with twice the vertical key spacing – a purely arbitrary choice, but one that results in a visually pleasing layout.

Next, the keyboard layout defined in `config.py` is traversed using a nested loop. In line 125, the corresponding `Pygame` key value is determined from the key label. This value is used as the dictionary key in line 130, allowing it to be easily compared with and processed from the parameters of keyboard events later on.

For each key, a corresponding sprite is created. The new horizontal position `left` is calculated by adding the key spacing to the actual width of the key. Once all keys in a row have been processed, `left` is reset to its initial value and the vertical position `top` is shifted downward accordingly.

Finally, the generated dictionary is returned. Alternatively, it could have been stored directly as a class attribute; however, the approach chosen here keeps the method self-contained.

Listing 2.73: Typewriter, `Game.generate_sprites()`

```
120     def generate_sprites(self) -> dict[int, KeySprite]:
121         keyboard = {}
122         left = top = 2 * cfg.KEY['spacing']
123         for row in cfg.ROWS:
```

```
            for label in row:
                key = self.label2key(label)            # Get pygame key constant
                if key is not None:
                    keysprite = KeySprite(label, key, left, top)
                    self.all_sprites.add(keysprite)
                    left += keysprite.rect.width + cfg.KEY['spacing']
                    keyboard[keysprite.key] = keysprite # Map key constant to sprite
            left = 2 * cfg.KEY['spacing']
            top += cfg.KEY['height'] + cfg.KEY['spacing']
        return keyboard
```

The tricky part is the method `label2key()`. Its task is to determine the corresponding `Pygame` key code, as listed in table 2.7 on page 101, based on a given key label. For many keys, such as T, we can directly use `pygame.key.key_code()` in line 116.

key_code()

For all other keys, we construct our own mapping tables. The underlying logic is more about Python than about Pygame itself and is therefore left to the interested reader to explore.

Listing 2.74: Typewriter, `Game.label2key()`

```
85      def label2key(self, label: str) -> int | None:
86          specials = {
87              'Space': pygame.K_SPACE,
88              'Enter': pygame.K_RETURN,
89              '<--': pygame.K_BACKSPACE,
90              'Tab': pygame.K_TAB,
91              'LShift': pygame.K_LSHIFT,
92              'LCtrl': pygame.K_LCTRL,
93              'RShift': pygame.K_RSHIFT,
94              'RCtrl': pygame.K_RCTRL,
95              'Alt': pygame.K_LALT,
96              'Caps': pygame.K_CAPSLOCK,
97              'Esc': pygame.K_ESCAPE,
98          }
99          for i in range(1, 13):
100             specials[f'F{i}'] = getattr(pygame, f'K_F{i}')
101
102         numpad = {
103             'KP+': pygame.K_KP_PLUS,
104             'KP-': pygame.K_KP_MINUS,
105             'KP*': pygame.K_KP_MULTIPLY,
106             'KP/': pygame.K_KP_DIVIDE,
107         }
108         for i in [0,1,2,3,4,5,6,7,8,9]:
109             numpad[f'KP{i}'] = getattr(pygame, f'K_KP{i}')
110
111         if label in specials:
112             return specials[label]
113         elif label in numpad:
114             return numpad[label]
115         try:
116             return pygame.key.key_code(label.lower())   # Alphanumeric keys
117         except Exception:
118             return None
```

The implementations of the methods `update()` and `draw()` are straightforward and require no further explanation.

Listing 2.75: Typewriter, `Game.update()` and `Game.draw()`

```python
    def update(self) -> None:
        self.all_sprites.update()

    def draw(self) -> None:
        self.screen.fill((30, 30, 30))
        self.all_sprites.draw(self.screen)
        self.window.flip()
```

2.6.2.4 Not by Event, but by Function

Another way to find out which key was pressed or released

- `pygame.key.get_pressed()`
- `pygame.key.get_just_pressed()`
- `pygame.key.get_just_released()`

These functions each return a dictionary containing all available keys, where every entry is associated with a Boolean flag:

- the value is `True` if the key is currently pressed (or was released, depending on the function),
- otherwise it is `False`.

The infix *just* therefore refers to the time span within two frames.

So, if you want to check whether K is currently pressed, you can simply write something like:

```python
        if pygame.key.get_pressed()[pygame.K_k]:
            ...
```

Often you will see constructions like this:

```python
        while pygame.key.get_pressed()[pygame.K_LEFT]:
            ...
```

With this approach, actions inside the loop can react immediately to a key being held down – for example, *walking to the right*.

However, you then get *trapped* inside the loop. If the game is supposed to react to something else as well, or if other game objects need to be updated, this does not work.

In that case, it is better to query the keyboard once per frame:

```
        keys = pygame.key.get_pressed()
        if keys[pygame.K_LEFT]:
            ...
```

This can be quite useful, but I have never really grown comfortable with this style of logic. Handling events as described above (see section 2.6.1 on page 89) does not trap me in loops and also feels "cleaner" to me, even though I cannot fully justify that feeling.

One important note from the documentation: with this function, you cannot determine the order in which keys were used. This is only possible via event handling.

What was new?

The keyboard sends event messages that can be intercepted and evaluated. First, a distinction is made as to what kind of keyboard action occurred (`event.type`), and then which key was involved (`event.key`). Using `event.mod`, it is possible to query bitwise which modifier keys on the keyboard were used.

The following Pygame elements were introduced

- `pygame.rect.FRect.clamp()`: https://pyga.me/docs/ref/rect.html#pygame.FRect.clamp
- `pygame.rect.FRect.clamp_ip()`: https://pyga.me/docs/ref/rect.html#pygame.FRect.clamp_ip
- `pygame.rect.Rect.clamp()`: https://pyga.me/docs/ref/rect.html#pygame.Rect.clamp
- `pygame.rect.Rect.clamp_ip()`: https://pyga.me/docs/ref/rect.html#pygame.Rect.clamp_ip
- `pygame.key`: https://pyga.me/docs/ref/key.html
- `pygame.key.get_pressed()`: https://pyga.me/docs/ref/key.html#pygame.key.get_pressed
- `pygame.key.get_just_pressed()`: https://pyga.me/docs/ref/key.html#pygame.key.get_just_pressed
- `pygame.key.get_just_released()`: https://pyga.me/docs/ref/key.html#pygame.key.get_just_released

`pygame.key.key_code()`:
https://pyga.me/docs/ref/key.html#pygame.key.key_code

`pygame.KEYDOWN`, `pygame.KEYUP`:
https://pyga.me/docs/ref/event.html

Table 2.7: Predefined Keyboard Constants

Constant	Meaning	Description
K_BACKSPACE	\b	backspace
K_TAB	\t	tabulator
K_CLEAR		clear
K_RETURN	\r	return, enter
K_PAUSE		pause
K_ESCAPE	^[	escape
K_SPACE		space
K_EXCLAIM	!	exclaim
K_QUOTEDBL	"	double quote
K_HASH	#	hash
K_DOLLAR	$	dollar
K_AMPERSAND	&	ampersand
K_QUOTE	'	quote
K_LEFTPAREN	(	left parenthesis
K_RIGHTPAREN	)	right parenthesis
K_ASTERISK	*	asterisk
K_PLUS	+	plus
K_COMMA	,	comma
K_MINUS	-	minus
K_PERIOD	.	period
K_SLASH	/	slash
K_0	0	0
K_1	1	1
K_2	2	2
K_3	3	3
K_4	4	4
K_5	5	5
K_6	6	6
K_7	7	7
K_8	8	8
K_9	9	9
K_COLON	:	colon

continued on next page

Table 2.7: Predefined Keyboard Constants (continued)

Constant	Meaning	Description
K_SEMICOLON	;	semicolon
K_LESS	<	less-than
K_EQUALS	=	equals
K_GREATER	>	greater-than
K_QUESTION	?	question mark
K_AT	@	at
K_LEFTBRACKET	[	left bracket
K_BACKSLASH	\	backslash
K_RIGHTBRACKET	]	right bracket
K_CARET	^	caret
K_UNDERSCORE	_	underscore
K_BACKQUOTE	`	grave
K_a	a	a
K_b	b	b
K_c	c	c
K_d	d	d
K_e	e	e
K_f	f	f
K_g	g	g
K_h	h	h
K_i	i	i
K_j	j	j
K_k	k	k
K_l	l	l
K_m	m	m
K_n	n	n
K_o	o	o
K_p	p	p
K_q	q	q
K_r	r	r
K_s	s	s
K_t	t	t
K_u	u	u
K_v	v	v
K_w	w	w
K_x	x	x
K_y	y	y
K_z	z	z
K_DELETE		delete

continued on next page

Table 2.7: Predefined Keyboard Constants (continued)

Constant	Meaning	Description
K_KP0		keypad 0
K_KP1		keypad 1
K_KP2		keypad 2
K_KP3		keypad 3
K_KP4		keypad 4
K_KP5		keypad 5
K_KP6		keypad 6
K_KP7		keypad 7
K_KP8		keypad 8
K_KP9		keypad 9
K_KP_PERIOD	.	keypad period
K_KP_DIVIDE	/	keypad divide
K_KP_MULTIPLY	*	Nummernfeld multiply
K_KP_MINUS	-	keypad minus
K_KP_PLUS	+	keypad plus
K_KP_ENTER	\r	keypad return, enter
K_KP_EQUALS	=	keypad equals
K_UP		up arrow
K_DOWN		down arrow
K_RIGHT		right arrow
K_LEFT		left arrow
K_INSERT		insert
K_HOME		home
K_END		end
K_PAGEUP		page up
K_PAGEDOWN		page down
K_F1		F1
K_F2		F2
K_F3		F3
K_F4		F4
K_F5		F5
K_F6		F6
K_F7		F7
K_F8		F8
K_F9		F9
K_F10		F10
K_F11		F11
K_F12		F12
K_F13		F13

continued on next page

Table 2.7: Predefined Keyboard Constants (continued)

Constant	Meaning	Description
K_F14		F14
K_F15		F15
K_NUMLOCK		numlock
K_CAPSLOCK		capslock
K_SCROLLOCK		scrollock
K_RSHIFT		right shift
K_LSHIFT		left shift
K_RCTRL		right control
K_LCTRL		left control
K_RALT		right alt
K_LALT		left alt
K_RMETA		right meta
K_LMETA		left meta
K_LSUPER		left Windows key
K_RSUPER		right windows key
K_MODE		mode shift/AltGr
K_HELP		help
K_PRINT		print screen
K_SYSREQ		sysreq
K_BREAK		break
K_MENU		menu
K_POWER		power
K_EURO	€	Euro
K_AC_BACK		Android back button

Table 2.8: Predefined Keyboard Modifier

Constant	Description
KMOD_NONE	no modifier keys pressed
KMOD_LSHIFT	left shift
KMOD_RSHIFT	right shift
KMOD_SHIFT	left or right shift or both
KMOD_LCTRL	left control
KMOD_RCTRL	right control
KMOD_CTRL	left or right control or both
KMOD_LALT	left alt
KMOD_RALT	right alt
KMOD_ALT	left or right alt or both

continued on next page

Table 2.8: Predefined Keyboard Modifier (continued)

Constant	Description
`KMOD_LMETA`	left meta
`KMOD_RMETA`	right meta
`KMOD_META`	left or right meta or both
`KMOD_CAPS`	caps lock
`KMOD_NUM`	num lock
`KMOD_MODE`	AltGr

2.6.3 Homework

1. Place a ball bitmap with a non-uniform shape in the center of the window. The ball starts with a radius of 100 px. Pressing + increases the size of the ball, and pressing - decreases it. If the ball touches the border of the window, its size must not be reduced any further. The minimum radius is 10 px. scale()
2. Use the arrow keys to move the ball inside the window. The ball must not leave the window.
3. Using ⇧+← rotates the ball by $-90°$, and ⇧+→ rotates it by $90°$. The function you need is `pygame.transform.rotate()`. rotate()
4. Change the rotation from $-90°$ to $-10°$ and from $90°$ to $10°$. Do you notice anything strange? Find out how to avoid this behavior. It is indeed a little tricky.

2.7 Text output using fonts

2.7.1 Introduction

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/07%20Fonts

Figure 2.38: Simple text output using fonts

In many games, information is not shown only in a symbolic way on the playfield (for example, three little figures to represent three lives), but also as written text. One way to achieve this is by displaying text using installed fonts.

render

The basic idea is simple: first, a `Font` object is created (see line 14). Then this font is used to generate a Surface object that contains the text – the text is rendered onto a `Surface` object (see line 28). Once rendered, the text behaves like any other bitmap and can be blitted onto the screen.

Listing 2.76: Simple text output using fonts

```
7  def main():
8      pygame.init()
9      window = pygame.Window(size=cfg.WINDOW.size, title="Text with Fonts")
10     screen = window.get_surface()
11     clock = pygame.time.Clock()
12     all_sprites = pygame.sprite.Group()
13
14     font = pygame.font.Font(pygame.font.get_default_font(), 24)     # Using default font
15     text = "This is an example of printing text using a font"
16
17     running = True
18     while running:
19         for event in pygame.event.get():
20             if event.type == pygame.QUIT:
21                 running = False
22             elif event.type == pygame.KEYDOWN:
23                 if event.key == pygame.K_ESCAPE:
24                     running = False
25         all_sprites.update()
26         screen.fill("white")
27         # Render and center the text
```

```
28         text_surface = font.render(text, True, (0, 0, 0))           # Render as usual
29         text_rect = text_surface.get_rect(center=(cfg.WINDOW.width // 2, cfg.WINDOW.height
               // 2))
30         screen.blit(text_surface, text_rect)
31         all_sprites.draw(screen)
32         window.flip()
33         clock.tick(cfg.FPS)
34 
35     pygame.quit()
```

But how does the contructor of `Font` know, which font I want? In line 14 we use the predefined default font. The method `pygame.font.get_default_font()` returns a unique string – the internal name of the font. You can find in section 2.7.2.2 on page 110 a program, which shows you all installed fonts and there internal names.

get_default_font()

If you want to use a specific font, you can get all required information with `pygame.font.match_font()`:

match_font()

```
14     font = pygame.font.Font(pygame.font.match_font("arial"), 24)
```

Don't wast computing time

It is a waste of computing time to render the text every frame. Try to cache the rendered `Surface`-Object and use some kind of dirty-flag to indicate, if the text was updated.

2.7.2 More Input

2.7.2.1 A More Sophisticated Approach

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/07%20Fonts

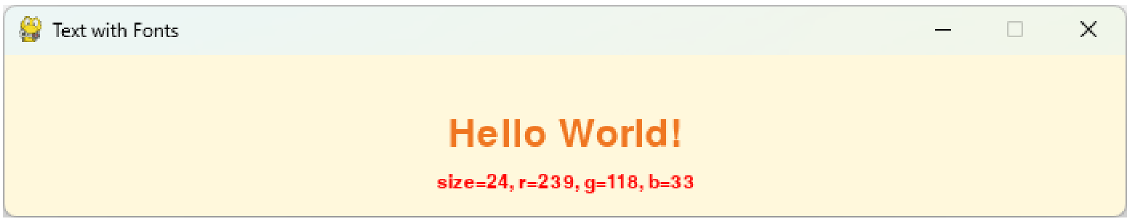

Figure 2.39: Text output using fonts

For a small example, I have wrapped this functionality into a class. You can easily extend it, modify it, or adapt it to your own needs.

Listing 2.77: Text output using fonts (1), `config.py`

```
import pygame

WINDOW = pygame.Rect((0, 0), (700, 100))
FPS = 60
```

Font

And now the class `TextSprite`: do not let the OO approach confuse you. In fact, everything is quite simple. We need a `pygame.font.Font` object . This object, in turn, requires two pieces of information: which installed font it should use, and the font size in *pt*.

get_default_font()

One way to obtain an installed font is the method `pygame.font.get_default_font()`. Its call in line 34 returns the default font configured by the operating system. The font size (`fontsize`) can then be chosen freely according to our needs.

Listing 2.78: Text output using fonts (2), `TextSprite`

```
8  class TextSprite(pygame.sprite.Sprite):
9      def __init__(self, fontsize: int, fontcolor: list[int],
10                  center: Tuple[int, int], text: str = "Hello World!") -> None:
11         super().__init__()
12         self.image = None
13         self.rect = None
14         self.fontsize = fontsize
15         self.fontcolor = fontcolor
16         self.fontsize_update(0)     # 0!
17         self.text = text
18         self.center = center
19         self.dirty = True
20         self.render()               #
21 
22     def render(self) -> None:
23         self.image = self.font.render(self.text, True, self.fontcolor)  # Bitmap
24         self.rect = self.image.get_rect()
25         self.rect.center = self.center
26         self.dirty = False
27 
28     def text_update(self, text: str) -> None:
29         self.text = text
30         self.dirty = True
31 
32     def fontsize_update(self, step: int = 1) -> None:
33         self.fontsize += step
34         self.font = pygame.font.Font(pygame.font.get_default_font(), self.fontsize)  #
35         self.dirty = True
36 
37     def fontcolor_update(self, delta: Tuple[int, int, int]) -> None:
38         for i in range(3):
39             self.fontcolor[i] = (self.fontcolor[i] + delta[i]) % 256
40         self.dirty = True
41 
42     def update(self) -> None:
43         if self.dirty:
44             self.render()
```

Let us now take a closer look at the constructor. The attributes `image` and `rect` are initially created as dummy values; strictly speaking, this would not even be necessary. After storing the passed information about font size and font color in attributes, the `Font` object can be created. This is done by calling `fontsize_update()` in line 16. Passing the value 0 makes it clear that the size itself is not meant to be changed here, but that the object creation should take place.

Next, the actual text that is to be rendered as a label is stored, as well as the position where the center of the text should be placed. At this point, all required information is available, and by calling `render()` in line 20, the `Surface` object is created using `pygame.font.render()` (line 23). Afterwards, the rectangle of the bitmap is determined and its center is moved to the desired position.

render()

Finally, there are the two methods `fontsize_update()` and `fontcolor_update()`. Both allow the font size and font color to be changed at runtime. Their semantics should be self-explanatory.

How can such a class be used in practice? Here is a simple example. A greeting is displayed in the center using the object `hello` (line 53). Below it, the object `info` displays which font size and font color were used to render the greeting (line 53).

Listing 2.79: Text output using fonts (3), Main program

```
47 def main():
48     pygame.init()
49     window = pygame.Window(size=cfg.WINDOW.size, title="Text with Fonts", position=(10, 50))
50     screen = window.get_surface()
51     clock = pygame.time.Clock()
52 
53     hello = TextSprite(24, [255, 255, 255], (cfg.WINDOW.center))  #
54     info = TextSprite(12, [255, 0, 0], (cfg.WINDOW.centerx, cfg.WINDOW.bottom - 20))  #
           Fontinfo
55     all_sprites = pygame.sprite.Group()
56     all_sprites.add(hello, info)
57 
58     running = True
59     while running:
60         for event in pygame.event.get():
61             if event.type == pygame.QUIT:
62                 running = False
63             elif event.type == pygame.KEYDOWN:
64                 if event.key == pygame.K_ESCAPE:
65                     running = False
66                 elif event.key == pygame.K_KP_PLUS or event.key == pygame.K_PLUS:   # Bigger
67                     hello.fontsize_update(+1)
68                 elif event.key == pygame.K_KP_MINUS or event.key == pygame.K_MINUS: # Smaller
69                     hello.fontsize_update(-1)
70                 elif event.key == pygame.K_r:
71                     if event.mod & pygame.KMOD_SHIFT:
72                         hello.fontcolor_update((-1, 0, 0))  # Less red
73                     else:
74                         hello.fontcolor_update((+1, 0, 0))  # More red
75                 elif event.key == pygame.K_g:
76                     if event.mod & pygame.KMOD_SHIFT:
77                         hello.fontcolor_update((0, -1, 0))  # Less green
78                     else:
79                         hello.fontcolor_update((0, +1, 0))  # More green
80                 elif event.key == pygame.K_b:
```

```python
81                     if event.mod & pygame.KMOD_SHIFT:
82                         hello.fontcolor_update((0, 0, -1))  # Less blue
83                     else:
84                         hello.fontcolor_update((0, 0, +1))  # More blue
85
86             info.text_update(f"size={hello.fontsize}, r={hello.fontcolor[0]},
                   g={hello.fontcolor[1]}, b={hello.fontcolor[2]}")
87             all_sprites.update()
88             screen.fill("cornsilk1")
89             all_sprites.draw(screen)
90             window.flip()
91             clock.tick(cfg.FPS)
```

The greeting can be resized using `+` and `-` (line 66 ff.). The keys `r`, `g`, and `b` are used to manipulate the corresponding color channel. An uppercase letter increases the value (for example in line 72), while the lowercase letter decreases it (for example in line 74).

In figure 2.39 on page 107 you can see one possible visual result.

2.7.2.2 List of all Installed Fonts

So ... where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/07%20Fonts/fontslist

As another example, I would like to show you a small program that lists all installed fonts. This may be useful for getting ideas for visual design. The first part of the code should not cause any problems in terms of understanding.

Figure 2.40: List of all installed fonts

Listing 2.80: List of all installed fonts (1), `config.py`

```
import pygame

WINDOW = pygame.Rect((0, 0), (900, 300))
FPS = 15
```

The class `TextSprite` was customized a little bit, but has still the same logic.

Listing 2.81: List of all installed fonts (2), `TextSprite`

```
class TextSprite(pygame.sprite.Sprite):
    def __init__(self, fontname: str, fontsize: int = 24, fontcolor: list[int] = [255, 255,
        255], text: str = "") -> None:
        super().__init__()
        self.image = None
        self.fontname = fontname
        self.fontsize = fontsize
        self.fontcolor = fontcolor
        self.fontsize_update(0)
        if text == "":
            self.text = f"{self.fontname}: abcdefghijklmnopqrstxyzßöäü0123456789"
        else:
            self.text = text
        self.render()

    def render(self) -> None:
        self.image = self.font.render(self.text, True, self.fontcolor)
        self.rect = self.image.get_rect()

    def fontsize_update(self, step: int = 1) -> None:
        self.fontsize += step
        self.font = pygame.font.Font(pygame.font.match_font(self.fontname), self.fontsize)  #

    def fontcolor_update(self, delta: list[int]) -> None:
        for i in range(3):
            self.fontcolor[i] = (self.fontcolor[i] + delta[i]) % 256

    def update(self) -> None:
        self.render()
```

The class `BigImage` is responsible for managing all `FontSprite` images as one large bitmap. Later on, only a subsection of this bitmap is blitted onto the screen. This subsection depends on the current position within the list and is controlled by the attribute `offset`, which is updated in the method `update()` (line 46).

First, it is checked whether the upper or lower end of the bitmap has been reached. If this is the case, `top` or `bottom` is set accordingly, so that the entire screen is always filled. Otherwise, the `offset` rectangle is shifted up or down, and the corresponding subsection is determined using `pygame.Surface.subsurface()`. subsurface()

Listing 2.82: List of all installed fonts (3), `BigImage`

```
36 class BigImage(pygame.sprite.Sprite):
37     def __init__(self):
38         super().__init__()
39         self.offset = pygame.Rect(cfg.WINDOW)
40
41     def create_image(self, width: int, height: int) -> None:
```

```python
        self.image_total = pygame.Surface((width, height))
        self.image_total.fill("white")
        self.update(0)

    def update(self, delta: int) -> None:  # Compute offset
        if self.offset.top + delta >= 0:
            if self.offset.bottom + delta <= self.image_total.get_rect().height:
                self.offset.move_ip(0, delta)
            else:
                self.offset.bottom = self.image_total.get_rect().height
        else:
            self.offset.top = 0
        self.image = self.image_total.subsurface(self.offset)
        self.rect = self.image.get_rect()
```

get_fonts()

And now the main program. In the first part, a list of all installed font names is obtained via `pygame.font.get_fonts()` (line 64). Each of these names is passed to the constructor of `TextSprite`.

match_font()

Using the method `pygame.font.match_font()` (line 26), the actual font file is then searched for on the system. This method takes advantage of the fact that the font file name can usually be derived from the font name and the file extension `ttf`.

Listing 2.83: List of all installed fonts (4), `main()`

```python
58 def main():
59     pygame.init()
60     window = pygame.Window(size=cfg.WINDOW.size, title="List of all Installed Fonts",
           position=(10, 50))
61     screen = window.get_surface()
62     clock = pygame.time.Clock()
63 
64     fonts = pygame.font.get_fonts()  # All installed Fonts
65 
66     list_of_fontsprites = pygame.sprite.Group()
67     height = 0
68     width = 0
69     for name in sorted(fonts):
70         try:
71             t = TextSprite(name, 24, "black")
72             t.rect.top = height
73             height += t.rect.height
74             width = t.rect.width if t.rect.width > width else width
75             list_of_fontsprites.add(t)
76         except OSError as err:
77             print(f"OS error {err}")
78         except pygame.error as perr:
79             print(f"Pygame error: {perr} with font {name}")
80 
81     bigimage = pygame.sprite.GroupSingle(BigImage())
82     bigimage.sprite.create_image(width, height)
83     list_of_fontsprites.draw(bigimage.sprite.image_total)  #
84 
85     running = True
86     while running:
87         for event in pygame.event.get():
88             if event.type == pygame.QUIT:
89                 running = False
90             elif event.type == pygame.KEYDOWN:
91                 if event.key == pygame.K_ESCAPE:
92                     running = False
93                 if event.key == pygame.K_UP:
```

```
94                  bigimage.update(-cfg.WINDOW.height // 2)
95              if event.key == pygame.K_DOWN:
96                  bigimage.update(cfg.WINDOW.height // 2)
97
98          bigimage.draw(screen)
99          window.flip()
100         clock.tick(cfg.FPS)
101
102     pygame.quit()
```

In the `for`-loop, `TextSprite` objects are now created for all fonts, and their height and width are determined. All of these individual bitmaps are then blitted onto the large bitmap (line 83).

The main loop is now only responsible for scrolling (each time by one third of the screen height) and for terminating the program.

2.7.2.3 Using Locally Installed Fonts

So ... where are the files?

- https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/07%20Fonts/localfont
- https://github.com/adamsralf/pygame_book/blob/main/src/00%20Introduction/07%20Fonts/localfont/fonts/Alex%20Winterbottom%20License.txt

Figure 2.41: Example of using a locally installed font

In almost all aspects, using local fonts is identical to using system fonts. So what does this distinction actually mean? System fonts are fonts that have been registered and installed in the operating system via an installation process. Local fonts, on the other hand, are font files – such as `ttf` files – that are stored in a subdirectory of the game itself. In our example, this is the file `rothenbg.ttf` located in the `fonts` subdirectory.

There are a few practical reasons why local fonts are often preferred in games. By shipping a font file together with the game, you ensure that the visual appearance of the text is identical on all systems, independent of which fonts are installed on the player's operating system.

In the example above (source code 2.78 on page 108, line 34), the constructor of `Font` was informed which system font to use via the function `get_default_font()`. In source code 2.81 on page 111, this was done by specifying a name known to the system – that is, a string under which the font is registered in the operating system.

In the example source code 2.84, however, a file name including a relative path is passed directly to the constructor of `Font`. In this case, `Font` looks for the corresponding file and uses it to build its `Font` object (see line 14).

From the programmer's point of view, the difference between system fonts and local fonts is very small. In practice, it only affects how the constructor of `Font` is called. Once the `Font` object has been created, rendering and using text works in exactly the same way for both system and local fonts.

When using local fonts, it is important to pay attention to licensing. Not every font may be freely redistributed. Before including a font file in your project, you should always check whether its license allows redistribution as part of a game or application. Our example was taken from https://www.fontsquirrel.com/fonts/list/tag/historical and was kindly made available as freeware by Alex Winterbottom.

Listing 2.84: Using locally installed fonts

```
7  def main():
8      pygame.init()
9      window = pygame.Window(size=cfg.WINDOW.size, title="Text with Local Fonts")
10     screen = window.get_surface()
11     clock = pygame.time.Clock()
12     all_sprites = pygame.sprite.Group()
13
14     font = pygame.font.Font("./fonts/rothenbg.ttf", 24)     # Adjust the path as necessary
15     text = "This is an example of printing text using a locally installed font"
16
17     running = True
18     while running:
19         for event in pygame.event.get():
20             if event.type == pygame.QUIT:
21                 running = False
22             elif event.type == pygame.KEYDOWN:
23                 if event.key == pygame.K_ESCAPE:
24                     running = False
25         all_sprites.update()
26         screen.fill("white")
27         # Render and center the text
28         text_surface = font.render(text, True, (0, 0, 0))      # Render as usual
29         text_rect = text_surface.get_rect(center=(cfg.WINDOW.width // 2, cfg.WINDOW.height
               // 2))
30         screen.blit(text_surface, text_rect)
31         all_sprites.draw(screen)
32         window.flip()
33         clock.tick(cfg.FPS)
34
35     pygame.quit()
```

In figure 2.41 on the preceding page you can admire the result ;-)

2.7.2.4 Text output

So ... where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/07%20Fonts/textbitmaps

Text output is often not done via fonts, but via a SpriteLib. Such a library contains character sprites, symbols, or digits, usually in a special design that matches the style of the game. In figure 2.42 on the next page you can see a spritelib that provides sprites for a World War II dogfight game. Among other things, it contains the sprites for the digits 0–9 and the letters of the Latin alphabet.

One advantage of this approach is that you do not have to rely on a specific game font being available on the target system. If you render text using a font such as *Calibri*, that font must be installed on the player's computer.

A disadvantage is that bitmaps usually scale poorly, so you often do not have many different font sizes available.

The idea is to *punch out* the individual letters from the spritelib and store them in a suitable data structure. Whenever text needs to be displayed, the string is split into its characters, and the corresponding letter sprites are blitted from the data structure onto a target bitmap – for example onto the screen.

I will demonstrate this with a simple example. Our starting point is a spritelib that contains a character set in five different colors (see figure 2.43 on the following page).

The first part of source code 2.85 should look familiar and is only extended by a few convenience features. The file paths are now determined via the functions `filepath()` and `imagepath()`.

Listing 2.85: Textbitmaps (1), `config.py`

```python
import os

import pygame

WINDOW = pygame.Rect((0, 0), (700, 650))
PATH: dict[str, str] = {}
PATH["file"] = os.path.dirname(os.path.abspath(__file__))
PATH["image"] = os.path.join(PATH["file"], "images")
FPS = 60

def filepath(name: str) -> str:
    return os.path.join(PATH["file"], name)

def imagepath(name: str) -> str:
    return os.path.join(PATH["image"], name)
```

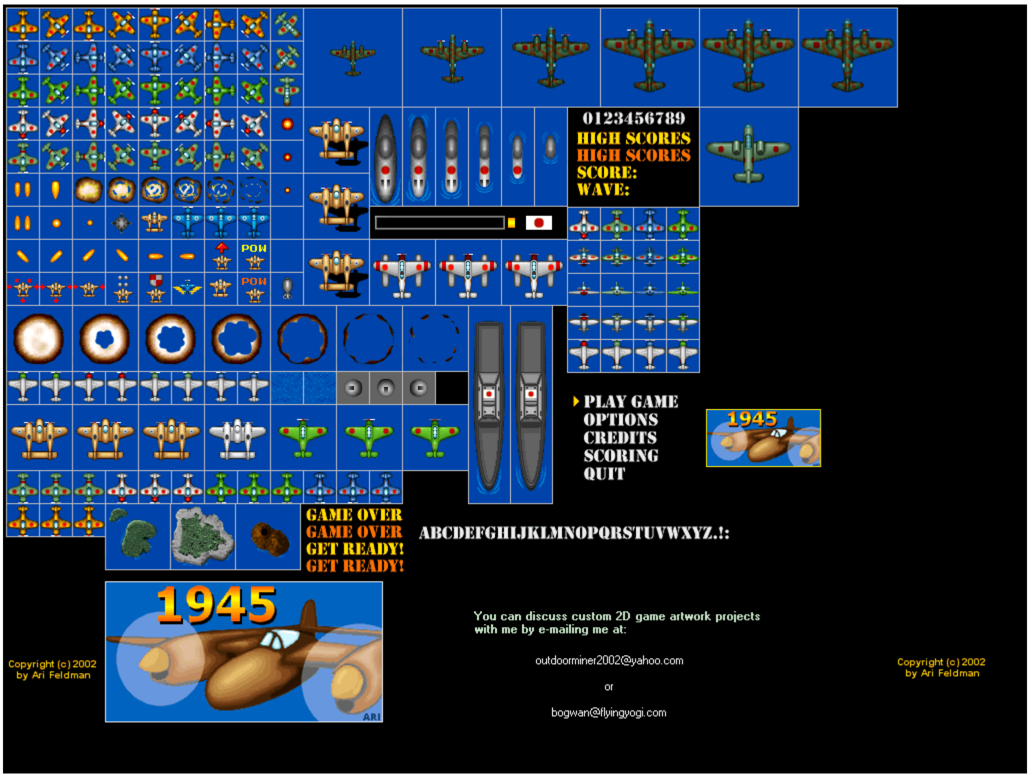

Figure 2.42: Example of a spritelib

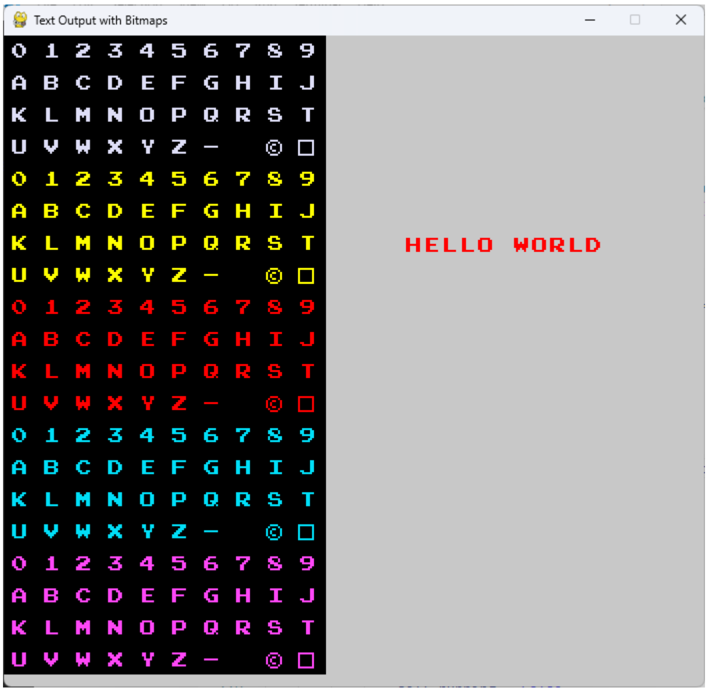

Figure 2.43: Text output using bitmaps

The class `Spritelib` is mainly used as a container. It loads the spritelib containing the letters and symbols and stores several parameters that are needed to extract individual letters or symbols precisely from the bitmap:

- `nof`: Stores the number of rows and columns. In our case, the symbol set is arranged in the bitmap in 4 rows and 10 columns. Since we are only interested in one color at a time, this information is sufficient.
- `letter`: Each sprite has a fixed width and height. In our case, this is particularly convenient because all sprites have the same dimensions. Take a look at the three squares around the letters `N`, `W`, and `X` in figure 2.44. All sprites have a width and a height of 18 px.
- `offset`: The first sprite in the top-left corner has a distance from the left edge and from the top edge of the bitmap. This can be seen clearly for the sprite of the digit `0` in figure 2.44. There is a square around the bitmap and a gap between this square and the upper and left edges (marked by the green line). In our example, both offsets have a value of 6 px.
- `distance`: Each sprite has a fixed distance to the next sprite to the right and to the one below. Fortunately, the sprites in our spritelib are arranged equidistantly, which simplifies things a lot. Using the sprite for `X` in figure 2.44 as an example, you can see these distances. In our case, they are 14 px each.

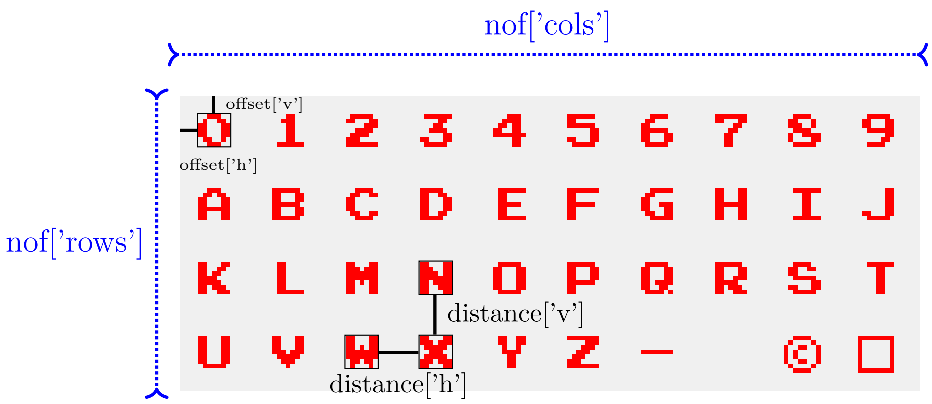

Figure 2.44: Meaning of the variables in `Spritelib`

Listing 2.86: Textbitmaps (2), `Spritelib`

```python
class Spritelib(pygame.sprite.Sprite):

    def __init__(self, filename: str) -> None:
        super().__init__()
        self.image = pygame.image.load(cfg.imagepath(filename)).convert()
        self.rect = self.image.get_rect()
        self.nof = {"rows": 4, "cols": 10}
```

```python
        self.letter = {"width": 18, "height": 18}
        self.offset = {"h": 6, "v": 6}
        self.distance = {"h": 14, "v": 14}

    def draw(self, screen: pygame.Surface) -> None:
        screen.blit(self.image, self.rect)
```

Dictionary

Let us now move on to the actually interesting class: `Letters`. This class cuts out all sprites of a single color from the spritelib and makes them available as `Surface` objects in a dictionary. This involves quite a bit of calculation, but do not let that intimidate you; in the end, it is nothing more than basic arithmetic.

Let us start with the constructor. The constructor has two parameters. The first parameter, `spritelib`, is a reference to the `Spritelib` object, which has loaded the original bitmap and provides several spacing and layout parameters. The second parameter, `colornumber`, allows us to extract the complete set of symbols for exactly one color later on: `0` stands for the white sprites, `1` for the yellow ones, and so on.

Listing 2.87: Textbitmaps (3), Constructor of `Letters`

```python
class Letters(object):

    def __init__(self, spritelib: Spritelib, colornumber: int) -> None:
        super().__init__()
        self.spritelib = spritelib
        self.letters: dict[str, pygame.Surface] = {}
        self.create_letter_bitmap(colornumber)
```

In the method `create_letter_bitmap()`, the individual sprites are now cut out and stored in a dictionary. The indices of this dictionary are defined in line 71. Here, the order must of course match the order in which the sprites are cut out. The variable `index` ensures exactly this: with each loop iteration, the next `lettername` is used as the key for the dictionary.

In line 76, the position – i. e. the pixel coordinates – of the first sprite is calculated. Try to follow the arithmetic yourself using the information given in figure 2.44 on the previous page! Don't worry: it is not difficult, just a bit lengthy.

Starting at line 77, a nested `for`-loop begins. The outer loop iterates over all rows of the spritelib, and the inner loop over the columns. The goal of this construction is to create one `Rect` object for each sprite, in which the position and size of the sprite are stored. In line 78, the top coordinate is calculated, and in line 80 the left coordinate. If you have understood line 76, these two calculations should no longer pose any problems.

Height and width in line 80 are straightforward, since all sprites always have the same dimensions. After that, the `Rect` object is created and used to cut out the bitmap with the help of `subsurface()`. This extracted bitmap is then stored in the dictionary under its symbol name.

Listing 2.88: Textbitmaps (4): `create_letter_bitmap()` von `Letters`

```python
    def create_letter_bitmap(self, colornumber: int):
        lettername = (
            "0",
            "1",  # The rows between 34 and 65 are skipped!!
            "2",
            "z",
            "-",
            " ",
            "copy",
            "square",
        )  #
        index = 0
        startpos = (
            self.spritelib.offset["h"],
            self.spritelib.offset["v"] + colornumber * self.spritelib.nof["rows"] *
                (self.spritelib.letter["height"] + self.spritelib.distance["v"]),
        )  #
        for row in range(self.spritelib.nof["rows"]):           # Rows
            for col in range(self.spritelib.nof["cols"]):       # Columns
                left = startpos[0] + col * (self.spritelib.letter["width"] +
                    self.spritelib.distance["h"])  #
                top = startpos[1] + row * (self.spritelib.letter["height"] +
                    self.spritelib.distance["v"])  #
                width  = self.spritelib.letter["width"]         # Size
                height = self.spritelib.letter["height"]
                r = pygame.Rect(left, top, width, height)
                self.letters[lettername[index]] = self.spritelib.image.subsurface(r)  #
                index += 1
```

The method `get_text()` finally returns the matching sequence of bitmap sprites for a given text. To do this, it uses the method `get_letter()`, which is necessary so that the program does not crash when an undefined letter or symbol is used. For example, if you type an ü, a square placeholder will be displayed.

Listing 2.89: Textbitmaps (5), `get_letter()` and `get_text()` von `Letters`

```python
    def get_letter(self, letter: str) -> pygame.Surface:
        if letter in self.letters:
            return self.letters[letter]
        else:
            return self.letters["square"]

    def get_text(self, text: str) -> pygame.Surface:
        l = len(text) * self.spritelib.letter["width"]
        h = self.spritelib.letter["height"]
        bitmap = pygame.Surface((l, h))
        bitmap.set_colorkey((0, 0, 0))
        for a in range(len(text)):
            bitmap.blit(self.get_letter(text[a]), (a * self.spritelib.letter["width"], 0))
        return bitmap
```

The actual main program is encapsulated in the class `TextBitmaps`. Since the source code does not introduce anything fundamentally new, it should be largely self-explanatory. However, I would like to take a closer look at two specific lines:

- line 123: Here, slicing of arrays is used. The value `-1` causes the end index of the slice to start at the last element and then move one step to the left. The result is a new string that is shortened by its last character.

unicode

- line 125: The attribute `unicode` provides, where applicable, the value of the pressed key in unicode format. This means that meaningful letters, digits, and similar characters are added directly to the string.

Listing 2.90: Textbitmaps (6), `TextBitmaps`

```
103 class TextBitmaps(object):
104 
105     def __init__(self) -> None:
106         pygame.init()
107         self.window = pygame.Window(size=cfg.WINDOW.size, title="Text Output with Bitmaps",
                position=(10, 50))
108         self.screen = self.window.get_surface()
109         self.clock = pygame.time.Clock()
110 
111         self.filename = "chars.png"
112         self.running = False
113         self.input = ""
114 
115     def watch_for_events(self) -> None:
116         for event in pygame.event.get():
117             if event.type == pygame.QUIT:
118                 self.running = False
119             elif event.type == pygame.KEYDOWN:
120                 if event.key == pygame.K_ESCAPE:
121                     self.running = False
122                 elif event.key == pygame.K_BACKSPACE:
123                     self.input = self.input[:-1] # Remove last character
124                 else:
125                     self.input += event.unicode  # Keyboard input as Unicode character
126 
127     def run(self) -> None:
128         spritelib = Spritelib(self.filename)
129         letters = Letters(spritelib, 2)
130         self.running = True
131         while self.running:
132             self.watch_for_events()
133             self.screen.fill((200, 200, 200))
134             self.screen.blit(letters.get_text(self.input), (400, 200))
135             spritelib.draw(self.screen)
136             self.window.flip()
137             self.clock.tick(cfg.FPS)
138 
139         pygame.quit()
```

2.7.3 What was new?

- To produce text output, you can use either system-installed fonts or local font files. In the first step, a suitable font object is created. In the second step, this object is used to render a given text into a bitmap — a `Surface` object. This bitmap can then be blitted to the desired position just like any other `Surface` object.
- But Text output is not only created using fonts, but also by means of spritelibs that contain character bitmaps. These bitmaps are cut out and then assembled into new composite bitmaps.

The following Pygame elements were introduced

- `pygame.event.Event.unicode`:
  https://pyga.me/docs/ref/event.html
- `pygame.font.Font`:
  https://pyga.me/docs/ref/font.html
- `pygame.font.get_default_font()`:
  https://pyga.me/docs/ref/font.html#pygame.font.get_default_font
- `pygame.font.get_fonts()`:
  https://pyga.me/docs/ref/font.html#pygame.font.get_fonts
- `pygame.font.match_font()`:
  https://pyga.me/docs/ref/font.html#pygame.font.match_font
- `pygame.font.Font.render()`:
  https://pyga.me/docs/ref/font.html#pygame.font.Font.render

  `pygame.Surface.subsurface()`:
  https://pyga.me/docs/ref/surface.html#pygame.Surface.subsurface
- `pygame.Surface.subsurface()`:
  https://pyga.me/docs/ref/surface.html#pygame.Surface.subsurface

2.7.4 Homework

Create a program that simulates a analog clock. The program contains the following status variables: number of lives, score, high score, the number of seconds since the program started, and a game title. Please position the information inside the window as follows:

- Title line: on the left the current date, in the center the game title, and on the right the current time
- Status line at the very bottom: on the left the number of lives, in the center score/high score, and on the right the number of seconds since the game started

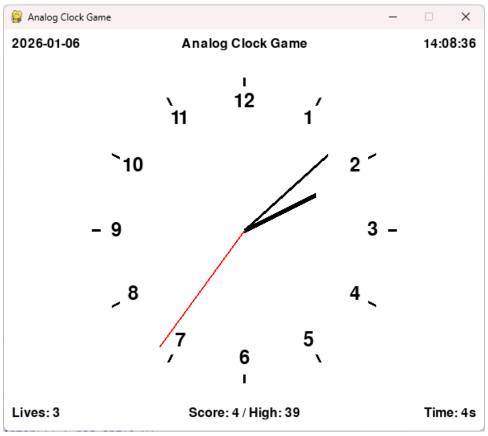

Fig. 2.45: Clock

The values should update dynamically. The score increases by 1 every second. This score is written to the file `highscore.txt` if it is greater than the current high score. Date, time, and elapsed time are obtained from suitable functions of the `time` module. Draw an analog clock in the center with second hand and dial like in figure 2.45.

2.8 Collision Detection

2.8.1 Introduction

Collision detection is used very often in game programming: characters must not walk through obstacles, projectiles hit targets, balls bounce off walls, and so on. For this reason, Pygame provides a whole variety of collision detection methods:

- **Rectangle overlap**: When we looked at the `Sprite` class, we already saw that the attribute `rect` is required. It contains the position and size of the surrounding rectangle. If two sprites meet, it is checked whether their rectangles overlap. This is a very *cheap* detection method, because only a few comparisons are needed to decide whether two rectangles touch or overlap. This method does not consider the actual shape of the sprite, only its bounding rectangle. Here is an example implementation:

```python
def rectangle_collision(rect1, rect2):
    return rect1.left < rect2.right and
           rect2.left < rect1.right and
           rect1.top < rect2.bottom and
           rect2.top < rect1.bottom
```

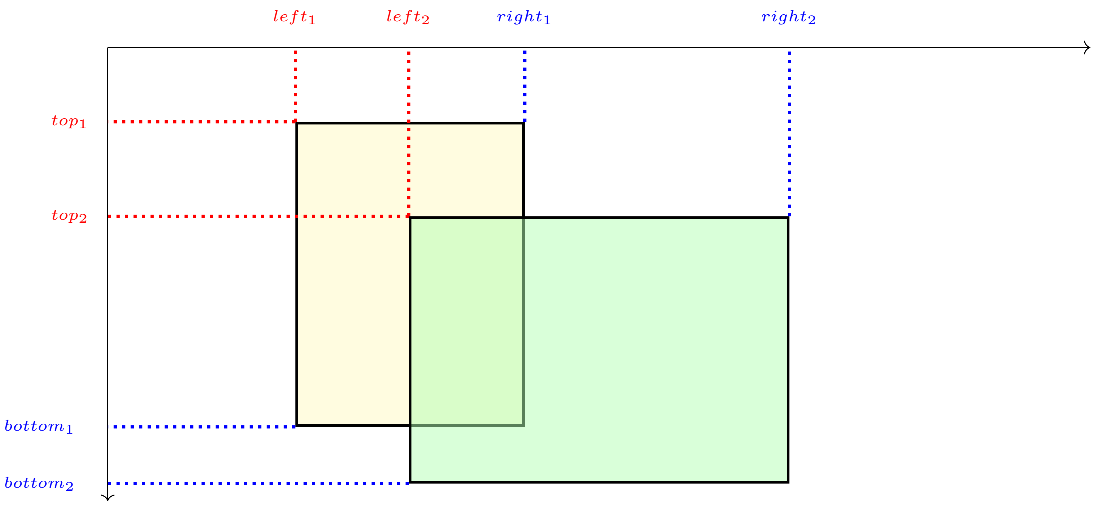

Figure 2.46: Collision detection with rectangles

- **Circle overlap**: For rather round sprites, it is recommended not to check rectangles, but to use an bounding circle for collision detection instead. This collision test is also quite fast, because only the distance between the centers has to be compared: $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} < r_1 + r_2$. For performance reason the check is usually computed as: $(x_2 - x_1)^2 + (y_2 - y_1)^2 < (r_1 + r_2)^2$

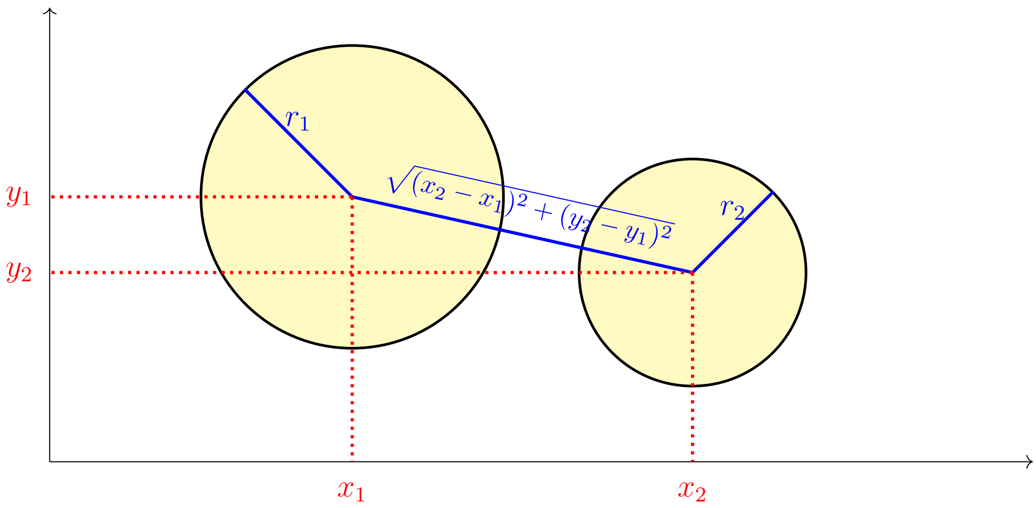

Figure 2.47: Collision detection with circles

- **Pixel overlap**: In pixel-perfect collision detection, every pixel of both sprites is checked to see whether they occupy the same position. If *yes*, the sprites overlap; if *no*, they do not. This is the most expensive collision test, but also the most accurate one.

  To reduce the computational effort, the intersection rectangle of the two sprites is determined first. As with rectangle collision detection, it is first checked whether the two rectangles overlap at all. If they do not, the test can stop immediately. If they do, the intersection of the two rectangles is itself a rectangle (see figure 2.48).

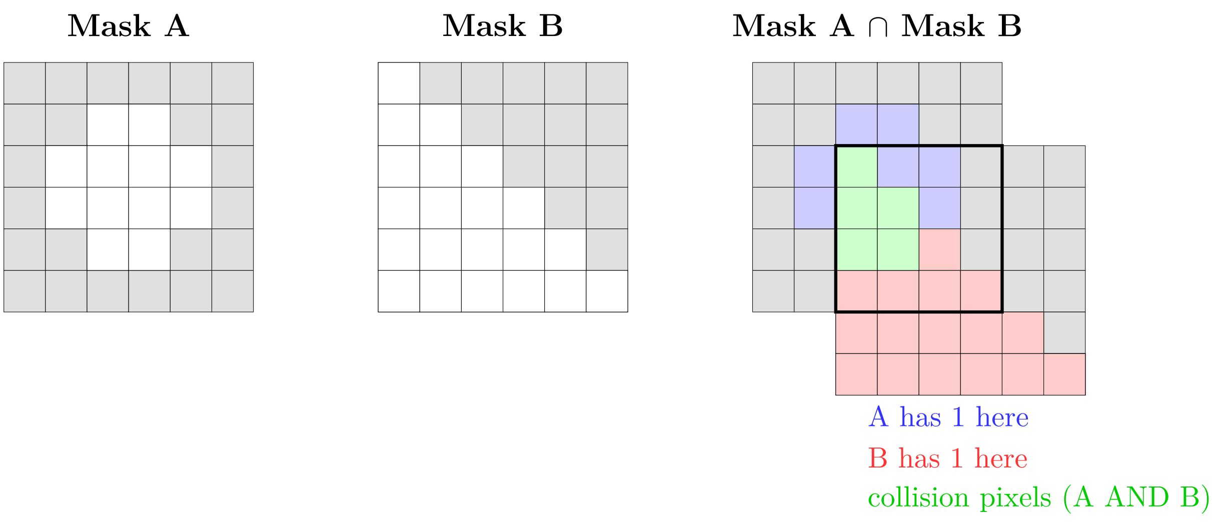

Figure 2.48: Collision detection using masks

If two pixels have the same position, they must lie inside this intersection rectangle.

Therefore, the pixel test can be limited to this usually much smaller area (see figure 2.49).

Another problem with pixel-perfect collision detection is distinguishing background from foreground. How should the collision test know whether a blue pixel belongs to the object or to the background? There are several approaches to this problem. The simplest one is to create a black-and-white image for each sprite (a mask); the white pixels are relevant, while the black pixels can be ignored. The pixel collision test is then performed only on these masks.

mask

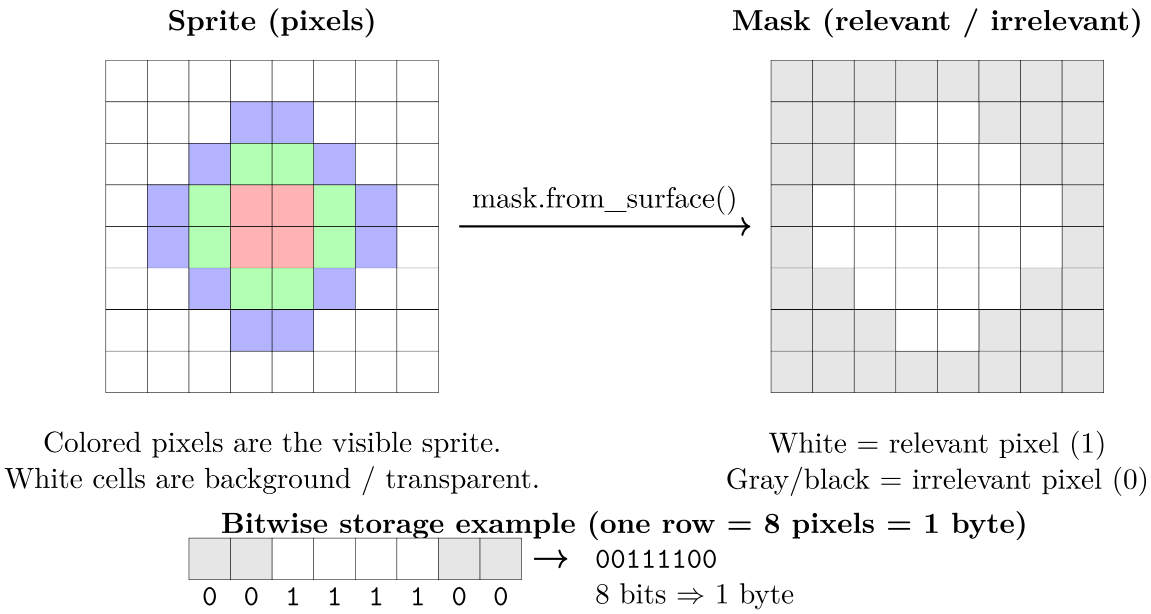

Figure 2.49: From sprite to mask

Let us look at the collision detection behaviour in more detail. In figure 2.50 on the following page we see four sprites: a wall, a spaceship, a monster, and a projectile. None of the sprites are touching each other.

In figure 2.51 on the next page you can clearly see the effect of collision detection using the bounding rectangles (bounding boxes). For the wall, everything is perfect: the projectile hits the wall, and the color indicates that the program has detected the collision.

However, we can also see the disadvantage when looking at the spaceship. A collision is detected even though the two sprites do not actually touch. The reason is that the spaceship's bounding rectangle also includes the empty corners, so the rectangles overlap and a collision is reported. The same effect can also be observed with the monster.

The situation is different when we use collision detection based on bounding circles (figure 2.52 on the following page). Now the collision with the wall is no longer detected

Figure 2.50: Four sprites

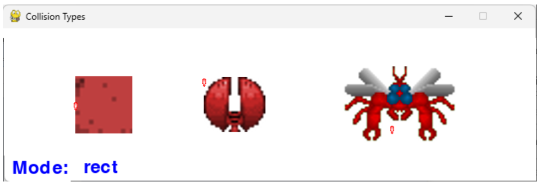

Figure 2.51: Collision detection using rectangles (montage)

correctly, because the corners of the wall do not belong to the inner circle. For the spaceship, however, this method produces exactly the desired result, since the empty corners are not part of the bounding circle. If we move a little further to the right, the spaceship would also turn red, because a collision would then be detected. The monster still produces an incorrect result.

Finally, there is pixel-perfect collision detection (figure 2.53). The collision with the wall is detected correctly. Even more interesting are the results for the spaceship and the monster. Both correctly report no collision, because the projectile is inside the rectangle and the inner circle, but only on transparent pixels. Feel free to try it yourself: move the projectile slightly to the left or right, and you will immediately see the pixel-perfect collision detection in action through the color change.

Figure 2.52: Collision detection using circles (montage)

Figure 2.53: Collision detection using masks (montage)

2.8.2 More Input

2.8.2.1 Three Types of Collision Detection (of a Bullet)

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/08%20Collision/v01

Let us now take a closer look at the corresponding source code. However, I will skip another discussion of the `config.py`.

Listing 2.91: Collision types (1), `config.py`

```
from os import path

import pygame

WINDOW = pygame.Rect((0, 0), (700, 200))
FPS = 60
TITLE = "Collision Types"
PATH: dict[str, str] = {}
PATH["file"] = path.dirname(path.abspath(__file__))
PATH["image"] = path.join(PATH["file"], "images")
MODE = "rect"

@staticmethod
def filepath(name: str) -> str:
    return path.join(PATH["file"], name)

@staticmethod
def imagepath(name: str) -> str:
    return path.join(PATH["image"], name)
```

Things become more interesting with the `Obstacle` class. This class is used for the wall, the spaceship, and the monster. For rectangle-based collision detection, the surrounding rectangle is required. As usual, it is obtained in line 14 using `pygame.Surface.get_rect()` and stored in the attribute `rect`.

get_rect()

For sprites with implicit transparency or explicit transparency set via `set_colorkey()`, the mask can be created very easily using `pygame.mask.from_surface()` (line 15). In order for the predefined collision detection functions to work, this mask must be stored in the `Sprite` object using the attribute `mask`.

self.rect

from_surface()

self.mask

In line 16, the bounding radius is calculated. This is implemented in a somewhat unclean way. Strictly speaking, one should determine the minimum of width and height and divide it by two.

```
15 self.radius = min(self.rect.width, self.rect.height) // 2
```

As with the mask, the radius must also be stored in an attribute so that the predefined collision methods can work: `radius`.

self.radius

The flag `hit` is only used to ensure that the correct image is displayed depending on the detected collision. As you have probably already noticed, two images are loaded for these sprites: one for the *not hit* state and one for the *hit* state.

Listing 2.92: Collision types (2), `Obstacle`

```
 7 class Obstacle(pygame.sprite.Sprite):
 8
 9     def __init__(self, filename1: str, filename2: str) -> None:
10         super().__init__()
11         self.image_normal = pygame.image.load(cfg.imagepath(filename1)).convert_alpha()
12         self.image_hit = pygame.image.load(cfg.imagepath(filename2)).convert_alpha()
13         self.image = self.image_normal
14         self.rect: pygame.Rect = self.image.get_rect()      # Bounding rectangle
15         self.mask = pygame.mask.from_surface(self.image)    # Pixel mask
16         self.radius = self.rect.width // 2                  # Bounding circle
```

```
        self.rect.centery = cfg.WINDOW.centery
        self.hit = False

    def update(self, *args: Any, **kwargs: Any) -> None:
        if "hit" in kwargs.keys():
            self.hit = kwargs["hit"]
        self.image = self.image_hit if (self.hit) else self.image_normal
```

The `Bullet` class is similar in many ways to the `Obstacle` class. Since we also want to use this class for all three types of collision detection, we need the same three attributes here as well: `rect`, `radius`, and `mask`.

In addition, the class contains a few lines of code to allow the bullet to move; this should be self-explanatory. Note: for the sake of simplicity, no boundary check has been implemented. There is no real need for it here.

Listing 2.93: Collision types (3), `Bullet`

```
class Bullet(pygame.sprite.Sprite):

    def __init__(self, picturefile: str) -> None:
        super().__init__()
        self.image = pygame.image.load(cfg.imagepath(picturefile)).convert_alpha()
        self.rect = self.image.get_rect()
        self.radius = self.rect.centery
        self.mask = pygame.mask.from_surface(self.image)
        self.rect.center = (10, 10)
        self.directions = {"stop": (0, 0), "down": (0, 1), "up": (0, -1),
                           "left": (-1, 0), "right": (1, 0)}
        self.set_direction("stop")

    def update(self, *args: Any, **kwargs: Any) -> None:
        if "action" in kwargs.keys():
            if kwargs["action"] == "move":
                self.rect.move_ip(self.speed)
        elif "direction" in kwargs.keys():
            self.set_direction(kwargs["direction"])

    def set_direction(self, direction: str) -> None:
        self.speed = self.directions[direction]
```

And now the `Game` class. In the constructor, the usual things happen. There is nothing particularly noteworthy here.

Listing 2.94: Collision types (4), Constructor of `Game`

```
class Game(object):

    def __init__(self) -> None:
        pygame.init()
        self.window = pygame.Window(size=cfg.WINDOW.size, title=cfg.TITLE)
        self.screen = self.window.get_surface()
        self.clock = pygame.time.Clock()

        self.font = pygame.font.Font(pygame.font.get_default_font(), 24)
        self.bullet = pygame.sprite.GroupSingle(Bullet("shoot.png"))
        self.all_obstacles = pygame.sprite.Group()
        self.all_obstacles.add(Obstacle("brick1.png", "brick2.png"))
        self.all_obstacles.add(Obstacle("ship1.png", "ship2.png"))
```

```python
        self.all_obstacles.add(Obstacle("alienbig1.png", "alienbig2.png"))
        self.running = False
```

The methods `run()` and `watch_for_events()` also follow well-established patterns.

Listing 2.95: Collision types (5), `run()` and `watch_for_events()` of `Game`

```python
    def run(self) -> None:
        self.resize()
        self.running = True
        while self.running:
            self.watch_for_events()
            self.update()
            self.draw()
            self.clock.tick(cfg.FPS)
        pygame.quit()

    def watch_for_events(self) -> None:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                self.running = False
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_ESCAPE:
                    self.running = False
                elif event.key == pygame.K_DOWN:
                    self.bullet.sprite.update(direction="down")
                elif event.key == pygame.K_UP:
                    self.bullet.sprite.update(direction="up")
                elif event.key == pygame.K_LEFT:
                    self.bullet.sprite.update(direction="left")
                elif event.key == pygame.K_RIGHT:
                    self.bullet.sprite.update(direction="right")
                elif event.key == pygame.K_r:
                    cfg.MODE = "rect"
                elif event.key == pygame.K_c:
                    cfg.MODE = "circle"
                elif event.key == pygame.K_m:
                    cfg.MODE = "mask"
            elif event.type == pygame.KEYUP:
                self.bullet.sprite.update(direction="stop")
```

The same applies to the methods `update()` and `draw()`.

Listing 2.96: Collision types (6), `update()` and `draw()` of `Game`

```python
    def update(self) -> None:
        self.check_for_collision()
        self.bullet.update(action="move")
        self.all_obstacles.update()

    def draw(self) -> None:
        self.screen.fill("white")
        self.all_obstacles.draw(self.screen)
        self.bullet.draw(self.screen)
        text_surface_modus = self.font.render(f"Mode: {cfg.MODE}", True, "blue")
        self.screen.blit(text_surface_modus, dest=(10, cfg.WINDOW.bottom - 30))
        self.window.flip()
```

The method `resize()` is not related to collision detection itself. Its only purpose is to ensure that the `Obstacle` objects are distributed evenly across the width of the window.

The first for-loop calculates the total width of all Obstacle objects. This information is needed to compute the spacing in line 117. To do this, the total obstacle width is subtracted from the window width. The remaining number of pixels can then be distributed across the gaps.

And how many gaps do we have? There are two gaps between the three Obstacle objects, one gap to the left border, and one to the right border – a total of four gaps. The resulting spacing is stored in padding.

In the second for-loop, the left position of each Obstacle object can then be calculated and set accordingly.

Listing 2.97: Collision types (7), resize() of Game

```
113     def resize(self) -> None:
114         total_width = 0
115         for s in self.all_obstacles:
116             total_width += s.rect.width
117         padding = (cfg.WINDOW.width - total_width) // 4  # Spacing between obstacles
118         for i in range(len(self.all_obstacles)):
119             if i == 0:
120                 self.all_obstacles.sprites()[i].rect.left = padding
121             else:
122                 self.all_obstacles.sprites()[i].rect.left = self.all_obstacles.sprites()[i -
                        1].rect.right + padding
```

Drum roll

And now the actual collision detection. Depending on which collision method we have selected, the corresponding collision function is called inside the for-loop: pygame.sprite.collide_circle(), pygame.sprite.collide_mask(), or pygame.sprite.collide_rect().

The semantics are actually quite simple. Each of these methods is given two Sprite objects and returns True if a collision is detected; otherwise it returns False. As already mentioned above, it is important to ensure that the method being used can find the information it requires in the sprite:

Required information

- pygame.sprite.collide_circle() needs self.radius
- pygame.sprite.collide_mask() needs self.mask
- pygame.sprite.collide_rect() needs self.rect

Listing 2.98: Collision types (8), `check_for_collision()` of `Game`

```
    def check_for_collision(self) -> None:
        if cfg.MODE == "circle":
            for s in self.all_obstacles:
                s.update(hit=pygame.sprite.collide_circle(self.bullet.sprite, s))
        elif cfg.MODE == "mask":
            for s in self.all_obstacles:
                s.update(hit=pygame.sprite.collide_mask(self.bullet.sprite, s))
        else:
            for s in self.all_obstacles:
                s.update(hit=pygame.sprite.collide_rect(self.bullet.sprite, s))
```

2.8.2.2 Checking all Sprites in a List

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/08%20Collision/v03

Rectangle-based collision detection between a single sprite and a list of sprites – that is, checking whether one sprite collides with any sprite in a `SpriteGroup` – is used so often that a dedicated method exists for this purpose: `pygame.sprite.spritecollide()`.

spritecollide()

The first parameter is a single `Sprite` object – in this case, our fireball. The second parameter is the list of sprites in which a collision should be checked. The third parameter controls whether the colliding objects should be removed from the list. This is very useful, for example, when an obstacle should disappear upon contact.

Below a minimal example (printed only partially). In line 54 the key part happens: one sprite – `player` – is checked for collisions with many sprites – `blocks`. Every sprite that the player collides with is removed from all groups using `kill()`, and is therefore most likely deleted completely.

Listing 2.99: `spritecollide()`

```
31 def main():
32     pygame.init()
33     screen = pygame.display.set_mode((640, 480))
34     clock = pygame.time.Clock()
35
36     player = Player((320, 240))
37     blocks = pygame.sprite.Group()
38     for x in range(100, 541, 80):
39         for y in (120, 200, 280):
40             blocks.add(Block((x, y)))
41
42     all_sprites = pygame.sprite.Group(player, *blocks.sprites())
43
44     font = pygame.font.SysFont(None, 24)
45
46     while True:
47         for event in pygame.event.get():
```

```python
            if event.type == pygame.QUIT:
                pygame.quit()

        keys = pygame.key.get_pressed()
        player.update(keys)

        pygame.sprite.spritecollide(player, blocks, True)   # spritecollide(sprite, group, dokill)

        screen.fill((30, 30, 30))
        all_sprites.draw(screen)
        text = font.render(f"Blocks remaining: {len(blocks)}", True, (200, 200, 200))
        screen.blit(text, (10, 10))

        pygame.display.flip()
        clock.tick(60)
```

2.8.2.3 Using Function Pointer/Collision Callback

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/08%20Collision/v02

function pointer

collision callback

The method `pygame.sprite.spritecollide()` has a fourth parameter as well. This parameter can be used to pass a function pointer or *collision callback* to a different collision detection method. This function must accept two `Sprite` objects as parameters.

This means you can either use your own custom collision function or one of the three predefined methods: `collide_circle()`, `collide_mask()`, or `collide_rect()`. If nothing is specified here – as in our source code – `collide_rect()` is used automatically.

Listing 2.100: Dynamic collision callback

```python
    def check_for_collision(self) -> None:
        match cfg.MODE:
            case "circle":
                func = pygame.sprite.collide_circle
            case "mask":
                func = pygame.sprite.collide_mask
            case _:
                func = pygame.sprite.collide_rect
        hits = pygame.sprite.spritecollide(self.bullet.sprite, self.all_obstacles, False, func)
        for s in self.all_obstacles:
            s.update(hit=s in hits)
```

2.8.3 What was new?

There are three standard ways to test the collision of two sprites:

- checking whether their rectangles intersect,
- whether their bounding circles intersect, or
- whether the pixels of the objects overlap.

In order to perform these collision tests, a sprite must provide the required information: `rect`, `radius`, or `mask`.

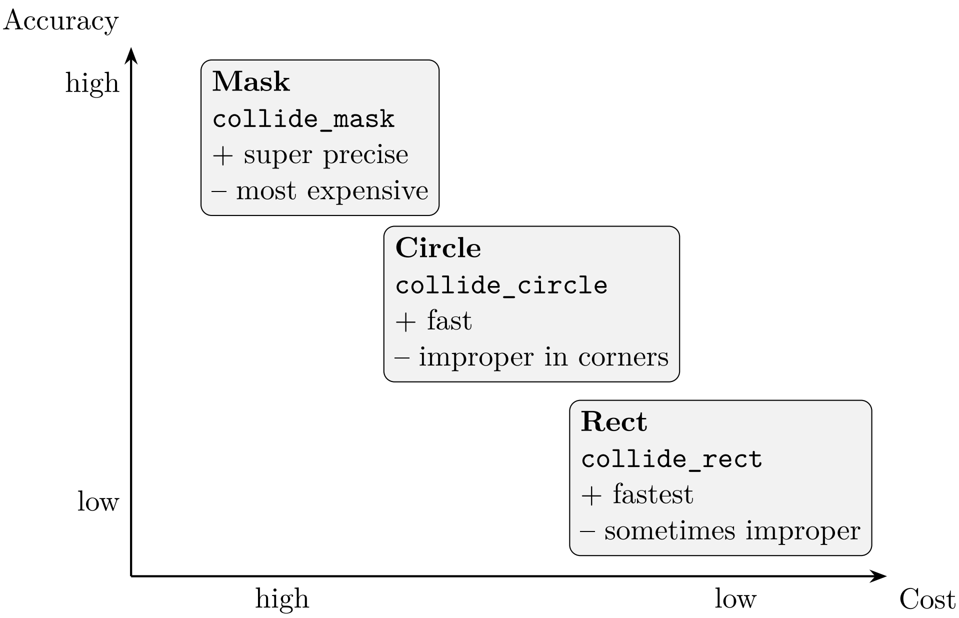

Figure 2.54: Trade-off Accuracy vs. Costs

The following Pygame elements were introduced

- `pygame.mask.from_surface()`:
  https://pyga.me/docs/ref/mask.html#pygame.mask.from_surface
- `pygame.sprite.collide_circle()`:
  https://pyga.me/docs/ref/sprite.html#pygame.sprite.collide_circle
- `pygame.sprite.collide_mask()`:

https://pyga.me/docs/ref/sprite.html#pygame.sprite.collide_mask

- `pygame.sprite.collide_rect()`: https://pyga.me/docs/ref/sprite.html#pygame.sprite.collide_rect
- `pygame.sprite.spritecollide()`: https://pyga.me/docs/ref/sprite.html#pygame.sprite.spritecollide

2.8.4 Homework

Figure 2.55: A simple collision game

Create a game with the following properties:

1. A player sprite is displayed at the bottom center of the playfield.
2. The player can move in all four directions using the keyboard, but cannot leave the playfield.
3. Thirty obstacles are placed on the playfield.
4. Ten obstacles each are detected using rectangle collision, circle collision, and mask-based collision detection.
5. The obstacles are placed in such a way that they do not overlap with each other or with the player.
6. The obstacles move downward at different speeds.

7. There is a score counter that starts at 0. It is to display on the top edge of the window.
8. If an obstacle leaves the playfield at the bottom, it reappears at the top and the score counter is increased by 1.
9. If an obstacle hits the player, the game is lost.
10. Bonus: Over time, the obstacles become faster and faster.
11. Bonus: A Game Over message and a restart option.

2.9 Time-based Actions

2.9.1 Introduction

So ... where are the files?

- https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/09%20Time/v01
- https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/09%20Time/v02
- https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/09%20Time/v03

In games, time-based actions are needed in many situations: a bomb drops every half second, a shield is active for 10 seconds, after 3 jumps the *jump* ability is not available for 5 minutes, animation frames should be displayed every 1/30 second, and so on.

Let us first look at the screen output of source code 2.101 on the facing pageff. shown in figure 2.56. The fireballs are obviously released in very quick succession, so that they appear like a chain. Because the enemy is moving horizontally, this results in a slanted line – which is clearly not the intended behaviour.

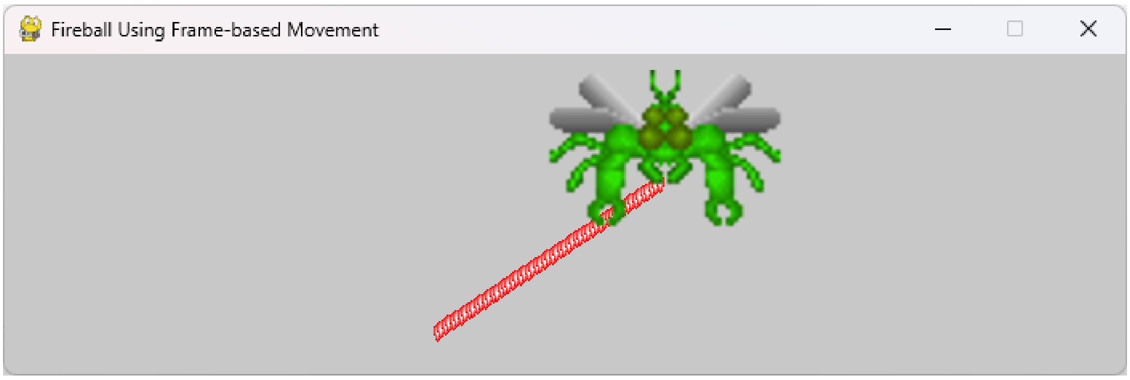

Figure 2.56: Fireball using frame-based movement

Before we take a closer look at time control itself, let us briefly look at the program. The `config.py` doesn't introduce anything new.

Listing 2.101: Time-based actions (1), `config.py`

```python
from os import path

import pygame

WINDOW = pygame.Rect((0, 0), (700, 200))
FPS = 60
DELTATIME = 1.0 / FPS
TITLE = "Fireball Using Counter-based Movement"
PATH: dict[str, str] = {}
PATH["file"] = path.dirname(path.abspath(__file__))
PATH["image"] = path.join(PATH["file"], "images")

def filepath(name: str) -> str:
    return path.join(PATH["file"], name)

def imagepath(name: str) -> str:
    return path.join(PATH["image"], name)
```

The `Enemy` class does not introduce anything particularly exciting either. With a spacing of 10 px, the enemy continuously moves back and forth from left to right and vice versa.

Listing 2.102: Time-based actions (2), Version 1.0 – `Enemy`

```python
class Enemy(pygame.sprite.Sprite):

    def __init__(self, filename: str) -> None:
        super().__init__()
        self.image = pygame.image.load(cfg.imagepath(filename)).convert_alpha()
        self.rect = pygame.FRect(self.image.get_rect())
        self.rect.topleft = (10, 10)
        self.direction = 1
        self.speed = pygame.math.Vector2(150, 0)

    def update(self, *args: Any, **kwargs: Any) -> None:
        newpos = self.rect.move(self.speed * cfg.DELTATIME * self.direction)
        if newpos.left < 10 or newpos.right >= cfg.WINDOW.right - 10:
            self.direction *= -1
        else:
            self.rect = newpos
```

The `Bullet` class is also largely a repetition of what we have seen before. What may be more interesting is line 39. The method `pygame.sprite.Sprite.kill()` is not a true self-destruction mechanism. Instead, this method removes the `Sprite` object from all sprite groups.

kill()

If all references to the object are lost as a result, the object is of course destroyed. However, if a reference still exists somewhere, the object will remain alive. In practice, `Sprite` objects are usually managed in groups (that is, in `pygame.sprite.Group` objects) and are therefore effectively destroyed by calling `kill()`.

You can see this effect in figure 2.56 on the facing page: the fireball disappears about 30 *px* before reaching the bottom edge of the screen.

Listing 2.103: Time-based actions (3), Version 1.0 – `Bullet`

```python
class Bullet(pygame.sprite.Sprite):

    def __init__(self, picturefile: str, startpos: Tuple[int, int]) -> None:
        super().__init__()
        self.image = pygame.image.load(cfg.imagepath(picturefile)).convert_alpha()
        self.rect = pygame.FRect(self.image.get_rect())
        self.rect.center = startpos
        self.direction = 1
        self.speed = pygame.math.Vector2(0, 100)

    def update(self, *args: Any, **kwargs: Any) -> None:
        self.rect.move_ip(self.speed * cfg.DELTATIME * self.direction)
        if self.rect.top > cfg.WINDOW.bottom - 30:
            self.kill()                             # self-destruction (remove sprite)
```

In the constructor of the `Game` class, a sprite group for the fireballs is created, as well as a `GroupSingle` object for the enemy. In `run()`, the usual game loop tasks are carried out by calling the appropriate methods.

tick()

delta time

I would like to briefly draw attention to line 61ff. By calling `pygame.time.Clock.tick()`, the game loop is timed – in this case to 1/60 of a second – and the *delta time* is calculated afterwards.

Listing 2.104: Time-based actions (4), Version 1.0 – Constructor and `run()` of `Game`

```python
42 class Game(object):
43 
44     def __init__(self) -> None:
45         pygame.init()
46         self.window = pygame.Window(size=cfg.WINDOW.size, title=cfg.TITLE)
47         self.screen = self.window.get_surface()
48         self.clock = pygame.time.Clock()
49 
50         self.enemy = pygame.sprite.GroupSingle(Enemy("alienbig1.png"))
51         self.all_bullets = pygame.sprite.Group()
52         self.running = False
53 
54     def run(self) -> None:
55         time_previous = time()
56         self.running = True
57         while self.running:
58             self.watch_for_events()
59             self.update()
60             self.draw()
61             self.clock.tick(cfg.FPS)           # frame rate limiting
62             time_current = time()
63             cfg.DELTATIME = time_current - time_previous
64             time_previous = time_current
65         pygame.quit()
```

The methods `watch_for_events()` and `draw()` also do not contain anything special.

Listing 2.105: Time-based actions (5), Version 1.0 – `watch_for_events()` and `draw()` of `Game`

```python
    def watch_for_events(self) -> None:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                self.running = False
```

```
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_ESCAPE:
                    self.running = False

    def draw(self) -> None:
        self.screen.fill((200, 200, 200))
        self.all_bullets.draw(self.screen)
        self.enemy.draw(self.screen)
        self.window.flip()
```

The `update()` method is only worth mentioning with regard to line **??**, because a new fireball is created (dropped) there by calling the `new_bullet()` method. The starting position is derived from the current position of the enemy. The horizontal center of the fireball and the enemy should be the same, while the vertical center is shifted slightly downward, which looks better visually.

Listing 2.106: Time-based actions (6), Version 1.0 – `update()` and `new_bullet()` of `Game`

```
        self.all_bullets.add(Bullet("shoot.png", self.enemy.sprite.rect.move(0, 20).center))
```

Back to the actual problem. As we saw above, the application is timed to 1/60 of a second by `FPS` and the call to `tick()` in line 61. In other words, up to 60 fireballs per second are currently created, which is nonsense.

A naive idea would be to reduce the frame rate. So, if I want to create a fireball only every half second, I would have to set the tick rate to 2. Try it! The result is disappointing: the entire game becomes slower. That is not the point.

Time-based counter

A next – and actually quite good – idea is to introduce a counter. The idea is: if the tick rate is 1/60, I count up to 30 and only then drop a fireball.

In the first step, two attributes are added to the `Game` class for this purpose (line 52 and line 53).

Listing 2.107: Time-based actions (7), Version 1.1 – Constructor of `Game`

```
51        self.all_bullets = pygame.sprite.Group()
52        self.time_counter = 0                    # Counter
53        self.time_range = 30                     # Threshold
54        self.running = False
```

In the `new_bullet()` method, these two values are now used to control the time interval between two drops. First, the counter is increased by 1 each time the method is called. Since the method is called once per iteration of the main program loop and each iteration is timed, this effectively counts the number of ticks.

If the counter exceeds its upper limit (30 in our example), half a second has passed since the last drop, and a new fireball is released.

Finally, the counter must be reset to 0, because we now have to wait for the next 30 ticks again. The result can be seen in figure 2.57: only two fireballs are visible now.

Listing 2.108: Time-based actions (8), Version 1.1 – `new_bullet()` of `Game`

```
    def new_bullet(self) -> None:
        self.time_counter += 1                      # Increment per frame
        if self.time_counter >= self.time_range:    # If threshold reached
            self.all_bullets.add(Bullet("shoot.png", self.enemy.sprite.rect.move(0,
                20).center))
            self.time_counter = 0                   # reset counter
```

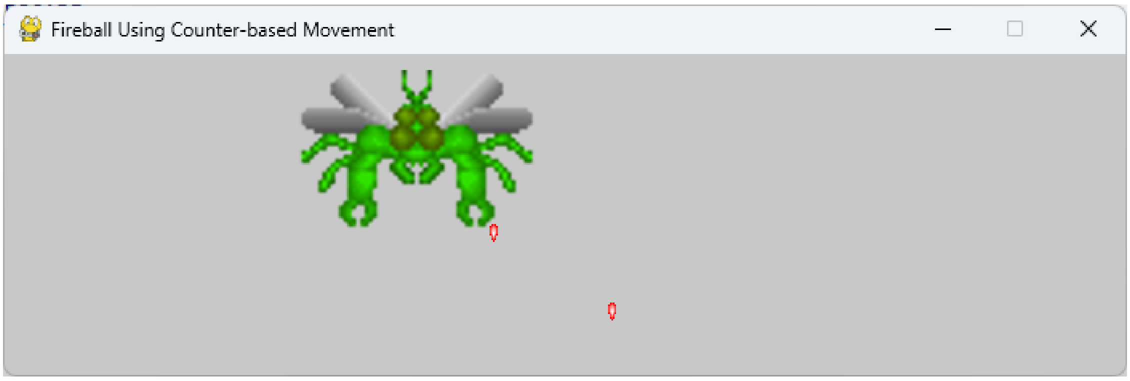

Figure 2.57: Fireball using counter-based movement

The advantages of this approach are clear: it is easy to implement, and the speed of the game itself is not affected.

However, there is a decisive disadvantage: this approach only works if the tick rate does not change and always behaves as expected. In reality, this is not guaranteed. As we remember, calling `tick()` ensures that the loop is executed *at most* 60 times per second. Under heavy load, it may run less often. In addition, many games determine the number of *frames per second* dynamically in order to adapt to different hardware performance. Therefore, coupling time control to the tick rate is not a truly stable solution.

get_ticks()

A better approach is to couple time control to a real time source. The method `pygame.time.get_ticks()` is very helpful here. This method returns the amount of time since the start of the game in Milliseconds (ms), and this value is independent of the performance of the hardware or the program.

Now the source code can be reworked. First, in line 52, the current number of *ms* since program start is measured, and in line 53 it is defined how many *ms* a time interval should last. We want to drop a fireball every half second, so this value is 500.

Listing 2.109: Time-based actions (9), Version 1.2 – Constructor of `Game`

```python
        self.enemy = pygame.sprite.GroupSingle(Enemy("alienbig1.png"))
        self.all_bullets = pygame.sprite.Group()
        self.time_stamp = pygame.time.get_ticks()   # Store timestamp
        self.time_duration = 500                    # Interval duration (ms)
        self.running = False
```

After that, `new_bullet()` checks whether the end of the interval has been reached. In line 89, the current time is measured again using `pygame.time.get_ticks()`. If this value is greater than the previous interval start plus the interval duration – which is the same as the interval end – then 500 ms have passed and a new fireball is dropped.

Now only the new interval start has to be determined, which is done in line 91.

Listing 2.110: Time-based actions (10), Version 1.2 – `new_bullet()` of `Game`

```python
88      def new_bullet(self) -> None:
89          if pygame.time.get_ticks() >= self.time_stamp + self.time_duration:  #
90              self.all_bullets.add(Bullet("shoot.png", self.enemy.sprite.rect.move(0,
                    20).center))
91              self.time_stamp = pygame.time.get_ticks()  # Start of next interval
```

2.9.2 More Input

Hint

Time control using events is introduced in section 2.12.2.1 on page 177.

2.9.2.1 The Class Timer

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/09%20Time/v04

Class Timer

Since we need this logic multiple times, I encapsulated it in the `Timer` class. The core of this class again consists of two attributes, which store the interval duration (`duration`) and the end of the interval (`next`). Unlike before, the interval start is no longer stored; instead, the interval end is saved – which slightly reduces the required computation.

The optional parameter `with_start` is particularly interesting. It allows us to control whether the code should wait until the first interval ends, or whether the very first call to `is_next_stop_reached()` should already return `True`.

What does this mean for our example? If `with_start` is set to `True`, the first fireball is dropped immediately in the very first loop iteration. If the value is `False`, the first fireball is only dropped after 500 ms.

In `is_next_stop_reached()`, it is checked whether the end of the interval has been reached, and, if necessary, a new interval end is calculated. The method returns `True` if the interval end has been reached or exceeded; otherwise, it returns `False`.

Listing 2.111: Time-based actions (11), Version 1.3 – Class `Timer`

```
class Timer(object):

    def __init__(self, duration: int, with_start: bool = True) -> None:
        self.duration = duration
        if with_start:
            self.next = pygame.time.get_ticks()
        else:
            self.next = pygame.time.get_ticks() + self.duration

    def is_next_stop_reached(self) -> bool:
        if pygame.time.get_ticks() > self.next:
            self.next = pygame.time.get_ticks() + self.duration
            return True
        return False
```

How is this timer used now? First, an appropriate object is created in the constructor (line 68); the two variables used previously are no longer needed.

Listing 2.112: Time-based actions (12), Version 1.3 – creating a `Timer` object

```
67          self.all_bullets = pygame.sprite.Group()
68          self.bullet_timer = Timer(500)              # Timer without initial delay
69          self.running = False
```

The `new_bullet()` method has now become simpler, since it no longer has to take care of the internal timer logic. It only checks in line 104 whether the interval end has been reached – and that's it!

Listing 2.113: Time-based actions (13), Version 1.3 – Using a `Timer` object verwenden

```
103     def new_bullet(self) -> None:
104         if self.bullet_timer.is_next_stop_reached():  # If interval boundary reached
105             self.all_bullets.add(Bullet("shoot.png", self.enemy.sprite.rect.move(0,
                    20).center))
```

For the sake of comparability – which may have been a bit confusing just now – I have placed the four workflows side by side in figure 2.58 on the next page.

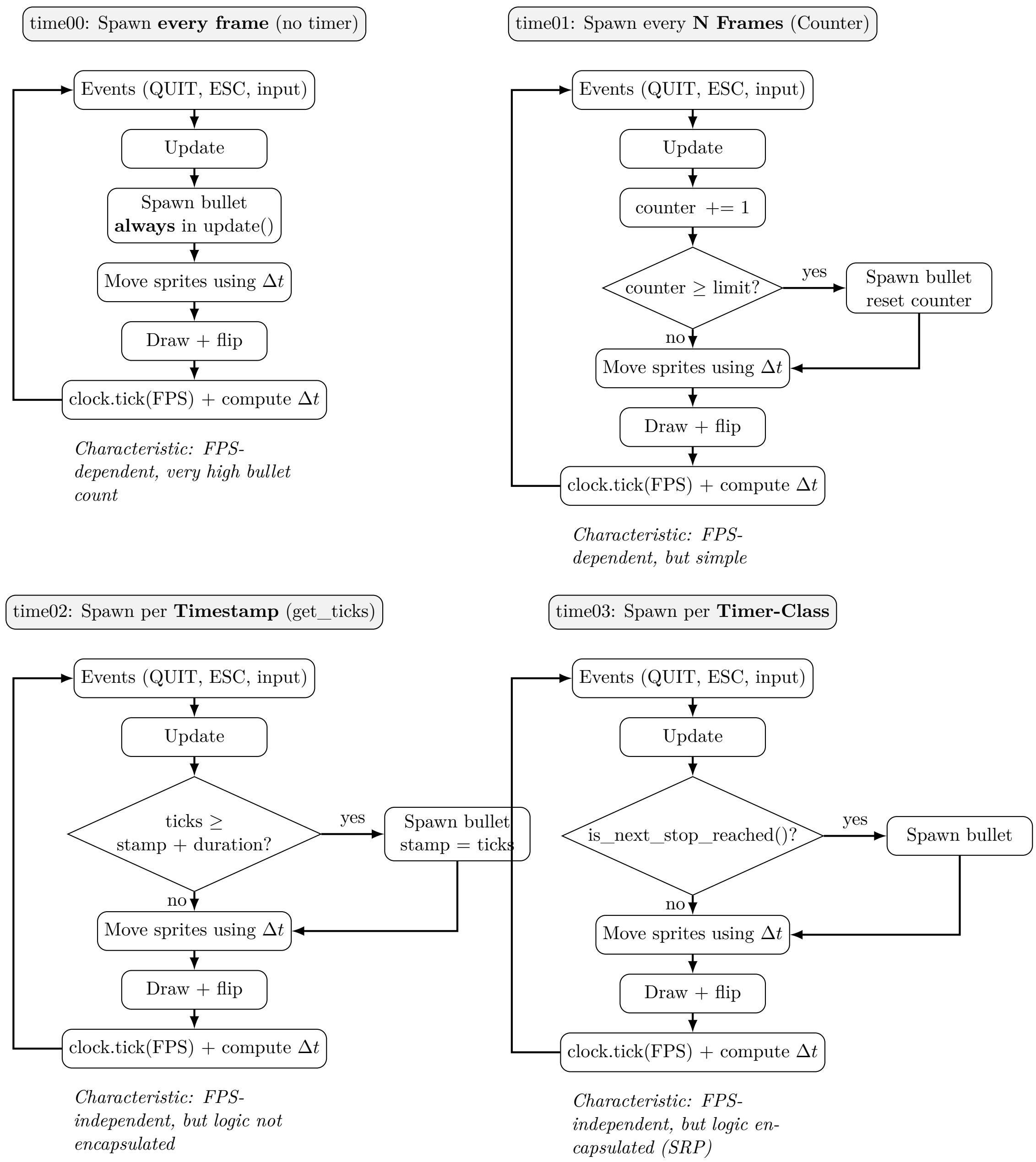

Figure 2.58: Comparison of the 4 algorithms

2.9.2.2 Accumulated Time

Sometimes it is sufficient to locally detect the progression of a time interval. For this purpose, one can use the `Timer` class, but it is also possible to accumulate the elapsed time and then compare the sum against a threshold.

Here is a small example:

```python
        elapsed += cfg.DELTATIME
        if elapsed >= 0.5:
                elapsed = 0.0
                spawn()
```

By using `DELTATIME`, a frame-rate–independent and fairly precise measurement of elapsed time is achieved. This logic is well suited for periodic actions or for animations.

2.9.2.3 Cool Down

If you want to let a certain amount of time pass – for example for shields or the time between two shots – the following small logic is commonly used:

```python
        if now() - last_shot >= cooldown:
                shoot()
                last_shot = now()
```

2.9.2.4 Start Delay

If you want to let a certain amount of time pass between two actions – for example because a start screen should be visible for a certain time – you can use the following approach:

```python
        start_time = pygame.time.get_ticks()
        if pygame.time.get_ticks() - start_time > 3000:
                do_something()
```

2.9.3 What was new?

Time-based events or time intervals should be made independent of the frame rate and should be based on the actual elapsed time. Since this is a frequently used logic, it is encapsulated in a separate class.

The following Pygame elements were introduced

- `pygame.time.get_ticks()`:
  https://pyga.me/docs/ref/time.html#pygame.time.get_ticks
- `pygame.sprite.Sprite.kill()`:
  https://pyga.me/docs/ref/sprite.html#pygame.sprite.Sprite.kill

2.10 Mouse

2.10.1 Introduction

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/10%20Mouse/example01

While many games are controlled using the keyboard or a controller, the mouse is also frequently used. In this script, basic mouse actions such as *clicking* and *position queries* are covered. Our example implements the following functionalities:

- A small transparent bubble appears in the center.
- When the mouse moves inside an inner rectangle, the bubble acts as the mouse cursor.
- When the mouse leaves the inner rectangle, the usual system mouse cursor appears.
- A left mouse click rotates the bubble 90° to the left.
- A right mouse click rotates the bubble 90° to the right.
- The mouse wheel is used to scale the size of the bubble.
- Clicking the mouse wheel terminates the application.

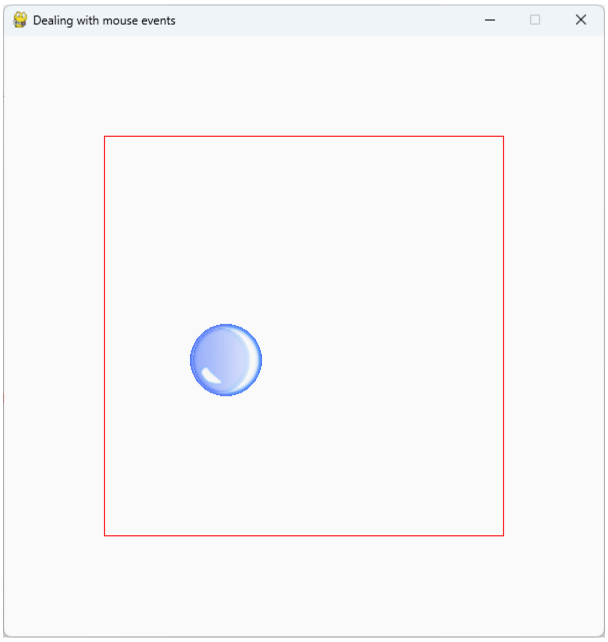

Figure 2.59: Example of actions with a mouse

The main action takes place in the `Game` class, since this is where the mouse actions are processed. Instead of using a separate `config.py` file, I implemented here static variables and methods in the `Game` class – that works as well. In the constructor, the usual suspects are initialized, and in line 67 the `Ball` object is created.

Listing 2.114: Mouse actions: statics and constructor of `Game`

```
55 class Game:
56     WINDOW = pygame.Rect((0, 0), (600, 600))
57     INNER_RECT = pygame.Rect(100, 100, WINDOW.width - 200, WINDOW.height - 200)
58     FPS = 60
59     DELTATIME = 1.0 / FPS
60
61     def __init__(self) -> None:
62         pygame.init()
63         self.window = pygame.Window(size=Game.WINDOW.size, title="Dealing with mouse events")
64         self.screen = self.window.get_surface()
65         self.clock = pygame.time.Clock()
66
67         self.ball = Ball()                          # Create ball object
68         self.running = True
```

The `run()` method also holds no surprises.

Listing 2.115: Mouse actions – `Game.run()`

```
    def run(self) -> None:
        time_previous = time()
        while self.running:
            self.watch_for_events()
            self.update()
            self.draw()
            self.clock.tick(Game.FPS)
            time_current = time()
            Game.DELTATIME = time_current - time_previous
            time_previous = time_current
        pygame.quit()
```

MOUSEBUTTONDOWN

In `watch_for_events()`, we encounter the first interesting parts. Just as `KEYDOWN` and `KEYUP` mark the pressing and releasing of keys, there are corresponding events for the mouse as well: `MOUSEBUTTONDOWN` and `MOUSEBUTTONUP`.

MOUSEBUTTONUP

In line 89, the value of `event.type` is checked and then it is determined which mouse button was pressed.

event.button

For this purpose, these two mouse events provide two attributes: `event.button` and `event.pos`. The numeric codes of `event.button` are shown in table 2.9 on page 153. Interestingly, there are no predefined constants here, unlike with keyboard input. After the check, the corresponding messages are sent to the `Ball` object.

If the left mouse button is pressed (line 90), a message is sent to the ball to rotate by 90° to the left, and if the right mouse button is pressed, to rotate by 90° to the right (hence −90°, see line 94).

The mouse wheel is also handled like a mouse button. Depending on the direction of rotation, a different numeric code is returned (see line 96 and line 98). If the mouse wheel is pressed – that is, clicked – the game should terminate. This is checked and implemented in line 92.

Using `event.pos`, one could immediately query the mouse position of this very moment – which we do not do here.

event.pos

Listing 2.116: Mouse actions – `Game.watch_for_events()`

```
82     def watch_for_events(self) -> None:
83         for event in pygame.event.get():
84             if event.type == QUIT:
85                 self.running = False
86             elif event.type == KEYDOWN:
87                 if event.key == K_ESCAPE:
88                     self.running = False
89             elif event.type == MOUSEBUTTONDOWN:     # Mouse button pressed
90                 if event.button == 1:               # Left
91                     self.ball.update(rotate=90)
92                 elif event.button == 2:             # Middle
93                     self.running = False
94                 elif event.button == 3:             # Right
95                     self.ball.update(rotate=-90)
96                 elif event.button == 4:             # Scroll up
97                     self.ball.update(scale=2)
98                 elif event.button == 5:             # Scroll down
99                     self.ball.update(scale=-2)
```

One requirement was that the system mouse cursor should only be visible outside the inner rectangle. Inside the rectangle, the ball is supposed to act as the mouse cursor. In line 104, this is achieved using the method `pygame.mouse.set_visible()`. This method controls whether the system mouse cursor – in whatever visual form – is shown or hidden.

set_visible()

The decision is based on whether the current mouse position lies inside the inner rectangle. The method `pygame.mouse.get_pos()` returns the current mouse position. This position is then simply passed into an already familiar collision test: `pygame.Rect.collidepoint()`. If the mouse position is inside the rectangle, this method returns `True`; otherwise, it returns `False`.

get_pos()

collidepoint()

Listing 2.117: Mouse actions – `Game.update()` and `Game.draw()`

```
101    def update(self):
102        newpos = pygame.mouse.get_pos()
103        self.ball.update(center=newpos)
104        if Game.INNER_RECT.collidepoint(pygame.mouse.get_pos()):  # Hide cursor?
105            pygame.mouse.set_visible(False)
106        else:
107            pygame.mouse.set_visible(True)
108        self.ball.update(go=True)
109
110    def draw(self) -> None:
111        self.screen.fill((250, 250, 250))
112        pygame.draw.rect(self.screen, "red", Game.INNER_RECT, 1)
113        self.ball.draw(self.screen)
114        self.window.flip()
```

The only class left is `Ball`. Although it no longer contains any direct mouse actions, the `update()` method now looks quite different from what we have seen in the previous examples.

And again: LSP

In earlier examples, methods such as `rotate()` or `resize()` were called directly from `watch_for_events()` or comparable methods of the `Game` class. This is perfectly fine in principle. However, problems arise once such subclasses of `pygame.sprite.Sprite` are added to a `pygame.sprite.Group` or a `pygame.sprite.GroupSingle`. These classes expect only `Sprite` objects as elements. Therefore, in terms of object-oriented programming, one should only use methods and attributes that are known to the parent class `pygame.sprite.Sprite` – for example, `update()`. Methods such as `rotate()` would be unknown to the sprite group.

Consider, for example, line 70 in source code 2.59 on page 83. The method `change_direction()` is completely unknown to the `GroupSingle` object `defender`, because it expects a `Sprite` and not a `Defender` object. Syntax checkers such as Pylance will report errors here.

Dispatching hub

One way to work around this problem is to use `update()` as a dispatching hub. In the class `pygame.sprite.Sprite`, this method is defined with the following signature:
`update(self, *args: Any, **kwargs: Any) -> None`

In other words, any number of freely definable parameters can be passed to this method. This is exactly what happens in our `update()` method. For rotation, the parameter `rotate` is passed with the corresponding angle; for scaling, the parameter `scale`; and in `update()` of `Game`, the parameter `go` is passed with the value `True`.

Each caller can therefore define its own parameters spontaneously and assign values to them. The `update()` method in the subclass – here `Ball` – only needs to check for these parameters.

In the first step, it is checked whether a parameter was provided, as shown in line 21, line 29, line 32, and line 35. Afterwards, the parameter value is forwarded to the corresponding method of the subclass. This way, the sprite group does not need to access methods of the subclass directly, but can rely on the method provided by the parent class.

Listing 2.118: Mouse actions – `Ball`

```python
class Ball(pygame.sprite.Sprite):
    def __init__(self) -> None:
        super().__init__()
        path = os.path.dirname(os.path.abspath(__file__))
        path = os.path.join(path, "images")
        fullfilename = os.path.join(path, "blue2.png")
        self.image_orig = pygame.image.load(fullfilename).convert_alpha()
        self.scale = 10
        self.image = pygame.transform.scale(self.image_orig, (self.scale, self.scale))
        self.rect = self.image.get_rect()

    def update(self, *args: Any, **kwargs: Any) -> None:
        if "go" in kwargs.keys():                      # Parameter present?
            if kwargs["go"]:
                self.rect.clamp_ip(Game.INNER_RECT)
                c = self.rect.center                    # Store previous center
                self.image = pygame.transform.scale(self.image_orig, (self.scale,
                    self.scale))
                self.rect = self.image.get_rect()
                self.rect.center = c                    # Reset center

        if "rotate" in kwargs.keys():                  #
            self.rotate(kwargs["rotate"])

        if "scale" in kwargs.keys():                   #
            self.resize(kwargs["scale"])

        if "center" in kwargs.keys():                  #
            self.set_center(kwargs["center"])

    def draw(self, screen: pygame.Surface) -> None:
        screen.blit(self.image, self.rect)

    def rotate(self, angle: float) -> None:
        self.image_orig = pygame.transform.rotate(self.image_orig, angle)

    def resize(self, delta: int) -> None:
        self.scale += delta
        if self.scale > Game.INNER_RECT.width:
            self.scale = Game.INNER_RECT.width
        elif self.scale < 5:
            self.scale = 5

    def set_center(self, center: Tuple[int, int]) -> None:
        self.rect.center = center
```

Two final notes on `pygame.transform.rotate()`

- Unlike many other systems that work with angles, the angle here is measured in degree rather than in radian.
- The approach used here works only because the bitmap represents a ball. In general, you should not repeatedly rotate the already rotated image. Instead, keep the original image unchanged and update only an angle variable. If you rotate the same image, the contents may be scaled up or down depending on the bitmap's shape, resulting in visual artifacts and severe pixel degradation.

2.10.2 More Input

2.10.2.1 In Which Window Took the Mouse Action Place?

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/10%20Mouse/example02

Window.id

event.window

Completely analogous to the example and explanation in section 2.6.2.2 on page 93,`id` it is also possible to determine in which window a mouse action occurred.

Listing 2.119: In which window took the mouse action place?

```
def main():
    pygame.init()
    window_first = pygame.Window(size=(300, 50),
        title="Main Window",
        position=(500, 50))
    window_second = pygame.Window(size=(300, 50),
        title="Side Window",
        position=(820, 50))
    screen_first = window_first.get_surface()
    screen_second = window_second.get_surface()
    clock = pygame.time.Clock()


    running = True
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
            elif event.type == pygame.WINDOWCLOSE:
                running = False
                event.window.destroy()
            elif event.type == pygame.MOUSEBUTTONDOWN:
                if event.window == window_first:    # Check which window the event belongs to
                    window_id = window_first.id
                    event.window.title = f"Main Window (Mouse Pressed: '{event.button}' at {event.pos})"
                elif event.window == window_second: #
                    window_id = window_second.id
                    event.window.title = f"Side Window (Mouse Pressed: '{event.button}' at {event.pos})"
                else:
                    window_id = None
                print(f"ID {window_id}: {event.window.title}")
        if running:
            screen_first.fill((0, 255, 0))
            window_first.flip()
            screen_second.fill((255, 0, 0))
            window_second.flip()
            clock.tick(60)

    pygame.quit()
```

Running the program produces the following console output when the corresponding keys are pressed:

```
ID 1: Main Window (Mouse Pressed: '1' at (55, 25))
ID 1: Main Window (Mouse Pressed: '3' at (101, 27))
ID 1: Main Window (Mouse Pressed: '2' at (171, 23))
ID 2: Side Window (Mouse Pressed: '1' at (74, 12))
ID 2: Side Window (Mouse Pressed: '2' at (123, 25))
ID 2: Side Window (Mouse Pressed: '3' at (224, 25))
```

2.10.2.2 Surprise: No Double Click

In Pygame, there is no built-in event for a mouse double click. Unlike classic desktop GUI frameworks, pygame only reports individual mouse button presses.

This means you cannot ask directly whether a double click happened. Instead, you have to detect it yourself.

The usual approach is to measure the time between two mouse clicks. If the same mouse button is pressed twice within a short time window, it is treated as a double click. Optionally, the mouse position can also be checked to make sure the cursor has not moved too far between clicks.

This may sound a bit low-level at first, but it is actually very common in game programming. Many higher-level interactions are built by combining simple events with a bit of timing logic.

Once implemented, this technique works reliably and gives you full control over what *double click* really means in your game or tool.

```python
import pygame
import time

DOUBLE_CLICK_TIME = 400  # ms

last_click_time = 0
last_button = None

for event in pygame.event.get():
        if event.type == pygame.MOUSEBUTTONDOWN:
                now = pygame.time.get_ticks()

                if (event.button == last_button and now - last_click_time <
                    DOUBLE_CLICK_TIME):
                        print("Double click detected!")

                last_click_time = now
                last_button = event.button
```

Options

- A `DOUBLE_CLICK_TIME` between 250 and 300 ms feels very snappy.
- A `DOUBLE_CLICK_TIME` between 400 and 500 ms is easier for beginners.
- In addition, you can check the distance between the `event.pos` values of both clicks.

2.10.2.3 Not by Event, but by Function

get_pressed()

In contrast to mouse button events, mouse button states can also be queried directly. Using the function `pygame.mouse.get_pressed()`, it is possible to check at any time which mouse buttons are currently being held down.

The function returns a tuple of boolean values that indicate whether the left, middle, or right mouse button is pressed. This allows continuous mouse input to be processed without relying on discrete events such as `MOUSEBUTTONDOWN` or `MOUSEBUTTONUP`.

This approach is particularly useful when mouse buttons should influence behaviour as long as they are held down, for example for dragging objects or continuous actions.

A completely analogous discussion can be found in section 2.6.2.4 on page 99.

2.10.3 What was new?

Mouse actions are processed in a similar way to keyboard events. The mouse position can be queried easily. It is often simpler to hide the mouse cursor and let a bitmap follow the mouse position than to define a new mouse cursor.

The following Pygame elements were introduced

- `pygame.constants`:
  https://pyga.me/docs/ref/locals.html
- `pygame.mouse.get_pressed()`
  https://pyga.me/docs/ref/mouse.html#pygame.mouse.get_pressed
- `pygame.MOUSEBUTTONDOWN`, `pygame.MOUSEBUTTONDOWN`:
  https://pyga.me/docs/ref/event.html
- List of mouse events: table 2.9 on the facing page

- `pygame.mouse.get_pos()`:
  https://pyga.me/docs/ref/mouse.html#pygame.mouse.get_pos
- `pygame.mouse.set_visible()`:
  https://pyga.me/docs/ref/mouse.html#pygame.mouse.set_visible
- `pygame.Rect.collidepoint()`:
  https://pyga.me/docs/ref/rect.html#pygame.Rect.collidepoint
- `pygame.transform.rotate()`:
  https://pyga.me/docs/ref/transform.html#pygame.transform.rotate

Table 2.9: List of mouse events

Constant	Description
0	undefined
1	left mouse button
2	middle mouse button/mouse wheel
3	right mouse button
4	scroll the mouse wheel up
5	Scroll the mouse wheel down

2.10.4 Homework

Time for a small game:

1. Create a fairly large playfield with a space look.
2. Your spaceship appears in the center.
3. Control the spaceship using the mouse. If one of the ⇧ keys is pressed, the mouse movement is interpreted as a control command. Hint: you need to compare the mouse position coordinates between two frames.
4. Rocks (asteroids) of different sizes fly into the screen from all sides.
5. The goal is to avoid being hit by a rock for as long as possible.
6. Every 10 seconds, the score is increased.
7. If I collide with a rock, I lose one life.
8. At the bottom, the score and the remaining lives are displayed.
9. I start with three lives. Add new lives repeatedly when certain score values are reached.

10. When a certain score is reached, I can activate a shield for 15 seconds using the right mouse button. Whether the shield is available should be shown in the bottom line.
11. BONUS: The nose of the spaceship always points in the direction of movement.

2.11 Sound

Without background sounds and/or music, many games would simply be boring. Therefore, I would like to present three different topics here: background music or ambient sounds, sound events, and stereo effects.

2.11.1 Introduction

2.11.1.1 Sound: Music

So ... where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/11%20Sound/music

The first example covers the following features:

- Background music is loaded and played in an endless loop.
- The volume can be adjusted using the mouse wheel.
- Pressing p pauses the background music or resumes playback.
- Pressing j fades out the background music.

I will not explain the imports, `config.py`, and the other familiar building blocks in detail anymore, as they have already appeared many times before.

Listing 2.120: Sound – `config.py`

```python
from os import path

import pygame

WINDOW: pygame.Rect = pygame.Rect(0, 0, 400, 200)   # Rect
FPS = 60
DELTATIME = 1.0 / FPS
PATH: dict[str, str] = {}
PATH["file"] = path.dirname(path.abspath(__file__))
PATH["image"] = path.join(PATH["file"], "images")
PATH["sound"] = path.join(PATH["file"], "sounds")
START_DISTANCE = 20
VOLUME_STEP = 0.05

def get_file(filename: str) -> str:
    return path.join(PATH["file"], filename)

def get_image(filename: str) -> str:
    return path.join(PATH["image"], filename)

def get_sound(filename: str) -> str:
    return path.join(PATH["sound"], filename)
```

init()

Before sound can be used, the corresponding subsystem must be initialized. This can be done explicitly using `pygame.mixer.init()`, or implicitly – as in the source code at line 10 – by calling `pygame.init()`.

background music

In the `sounds()` method, the preparatory steps for sound output are encapsulated. Background music is loaded into the mixer's internal memory using `pygame.mixer.music.load()`. However, loading the music does not start playback yet.

set_volume()

play()

Playback begins after the volume has been set in line 21 using `pygame.mixer.music.set_volume()`, by calling the method in line 22. The method `pygame.mixer.music.play()` accepts three parameters:

- The first parameter, `loops`, controls the number of repetitions; a value of -1 means that the music is repeated indefinitely.
- The second parameter, `start`, specifies the position at which playback should begin; the default is 0.0.
- fade

  If the music should start quietly and then become louder (fade), this can be achieved using the third parameter `fade`. Here, you can specify how many milliseconds are available for the fade-in; if nothing is specified, playback starts immediately at the target volume.

Listing 2.121: Sound – Constructor and `sounds()` of `Game`

```
 8 class Game:
 9     def __init__(self) -> None:
10         pygame.init()                              # Includes mixer
11         self.window = pygame.Window(size=cfg.WINDOW.size, title='Sound Background Music')
12         self.screen = self.window.get_surface()
13         self.clock = pygame.time.Clock()
14         self.font_bigsize = pygame.font.Font(pygame.font.get_default_font(), 40)
15         self.running = True
16         self.pause = False
17         self.sounds()
18 
19     def sounds(self) -> None:
20         pygame.mixer.music.load(cfg.get_sound("Lucifer.mid"))
21         pygame.mixer.music.set_volume(0.1)         #
22         pygame.mixer.music.play(-1, 0.0)           # Endles loop
```

The `watch_for_events()` method acts purely as a dispatcher. Depending on which key is pressed or which mouse element is used, the corresponding helper methods are called.

Listing 2.122: Sound – `Game.watch_for_events()`

```
    def watch_for_events(self) -> None:
        for event in pygame.event.get():
            if event.type == QUIT:
                self.running = False
            elif event.type == KEYDOWN:
                if event.key == K_ESCAPE:
                    self.running = False
            elif event.type == KEYUP:
                if event.key == K_f:
                    self.music_start_stop(fadeout=5000)
```

```
34                elif event.key == K_j:
35                    self.music_start_stop(loop=-1)
36                elif event.key == K_p:
37                    self.pause_alter()
38            elif event.type == pygame.MOUSEBUTTONUP:
39                if event.button == 4:  # up
40                    self.volume_alter(cfg.VOLUME_STEP)
41                elif event.button == 5:  # down
42                    self.volume_alter(-cfg.VOLUME_STEP)
```

I want to start the background music at some times and fade it out at others. This is handled by the helper method `music_start_stop()`. The background music is stopped using `pygame.mixer.music.fadeout()`. Here, you have to specify over how many milliseconds the music should gradually become quieter until it stops – in our example, this is 5000 ms. The method `pygame.mixer.music.play()` used to start the background music has already been explained above.

fadeout()

Listing 2.123: Sound – `Game.music_start_stop()`

```
45    def music_start_stop(self, **kwargs: Any) -> None:
46        if "fadeout" in kwargs.keys():
47            pygame.mixer.music.fadeout(kwargs["fadeout"])    #
48        if "loop" in kwargs.keys():
49            pygame.mixer.music.play(kwargs["loop"], 0.0)     #
```

Pressing `p` pauses the background music or resumes playback. The current state is stored in the attribute `pause`. This attribute then determines which of the two `music` methods is called in the `pause_alter()` method – either `pygame.mixer.music.pause()` or `pygame.mixer.music.unpause()`. Finally, in line 49, the `pause` flag is toggled.

pause()

unpause()

Listing 2.124: Sound – `Game.pause_alter()`

```
51    def pause_alter(self) -> None:
52        if self.pause:
53            pygame.mixer.music.unpause()                     #
54        else:
55            pygame.mixer.music.pause()
56        self.pause = not self.pause                          #
```

As the final feature, volume control is introduced. It is encapsulated in the `volume_alter()` method. Instead of passing an absolute volume value to this method, a delta value is provided.

First, this value is added to the `volume` variable. Afterwards, the value is clamped to the interval $[0, 1]$, and finally the new volume is set using `pygame.mixer.music.set_volume()`.

set_volume()

Listing 2.125: Sound – `volume_alter()` von `Game`

```
58    def volume_alter(self, delta: float) -> None:
59        volume = pygame.mixer.music.get_volume()
60        volume += delta
61        volume = pygame.math.clamp(volume, 0.0, 1.0)
62        pygame.mixer.music.set_volume(volume)                #
```

And finally, we deal with the remaining bits.

Listing 2.126: Sound – `draw()`, `update()`, `run()` of `Game`

```
    def draw(self) -> None:
        self.screen.fill("white")
        volume = pygame.mixer.music.get_volume()
        volume_surface = self.font_bigsize.render(f"Volume: {volume:3.2f}", True, "red")
        volume_rect = volume_surface.get_rect()
        volume_rect.center = cfg.WINDOW.center
        self.screen.blit(volume_surface, volume_rect)

        self.window.flip()

    def update(self):
        pass

    def run(self):
        self.running = True
        while self.running:
            self.watch_for_events()
            self.update()
            self.draw()
            self.clock.tick(cfg.FPS)
        pygame.quit()
```

2.11.1.2 Sound: Events

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/11%20Sound/effects

sound effects

For sound effects, a separate `Sound` object is created in each case (line 18ff.). The constructor of `pygame.mixer.Sound` is given the file name including the path. If you already have an open file object, you can pass that instead; in this case, however, you should provide a second parameter specifying the sound encoding, for example `.OGG` or `.MP3`. As with background music, loading a sound is not the same as playing it.

Listing 2.127: Sound – `Game.sound()`

```
17      def sounds(self) -> None:
18          self.bubble = pygame.mixer.Sound(cfg.get_sound("plopp.mp3"))    #
19          self.clash = pygame.mixer.Sound(cfg.get_sound("glas.wav"))      #
```

get_volume()

set_volume()

In line 39, the current volume is first stored in a variable using `pygame.mixer.Sound.get_volume()`. To ensure that both sounds are played at the same volume, this value is modified and then applied to both sounds using `pygame.mixer.Sound.set_volume()`.

Listing 2.128: Sound – `Game.volume_alter()`

```
    def volume_alter(self, delta: float) -> None:
        volume = self.bubble.get_volume()                       # For both
        volume += delta
        volume = pygame.math.clamp(volume, 0.0, 1.0)
        self.bubble.set_volume(volume)
        self.clash.set_volume(volume)
```

For simplicity, the sounds are played directly in `watch_for_events()` (see line 30 and line 32). The actual playback is done using `pygame.mixer.Sound.play()`. You can see that the `play()` method is called on the corresponding `Sound` object.

play()

The `play()` method provides three optional arguments:

- `loops`: number of repetitions (−1 means infinite playback, default)
- `maxtime`: maximum playback time in milliseconds (0 means unlimited, default)
- `fade_ms`: duration of the fade-in in milliseconds (default: 0)

If – as in this case –no arguments are provided, the sound starts playing immediately and stops automatically after it has finished playing. Any other sounds that are currently being played by other `Sound` objects are not interrupted. This means that multiple sounds can be played at the same time.

Listing 2.129: Sound – `Game.watch_for_events()()`

```
21      def watch_for_events(self) -> None:
22          for event in pygame.event.get():
23              if event.type == QUIT:
24                  self.running = False
25              elif event.type == KEYDOWN:
26                  if event.key == K_ESCAPE:
27                      self.running = False
28              elif event.type == pygame.MOUSEBUTTONUP:
29                  if event.button == 1:                       # left
30                      self.bubble.play()                      #
31                  elif event.button == 3:                     # right
32                      self.clash.play()                       #
33                  elif event.button == 4:                     # up
34                      self.volume_alter(0.05)
35                  elif event.button == 5:                     # down
36                      self.volume_alter(-0.05)
```

2.11.2 More Input

2.11.2.1 Stereo

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/11%20Sound/stereo

A small example is intended to illustrate the use of channels and stereo effects. The topic is too extensive to be presented in full detail, but I hope that this chapter provides a helpful introduction.

stereo panning

In figure 2.60, you can see a tank driving from left to right or from right to left. While driving, it can fire up to 5 shots. It would be nice if the driving sound indicated acoustically where the tank is currently located. That is, if the tank is more to the right, the driving sound or the shot should be louder on the right speaker than on the left speaker (stereo panning). When driving from right to left, the driving sound would therefore move along with the tank.

Figure 2.60: Example Stereo Sound

First, the necessary boilerplate, which should not require any further explanation:

Listing 2.130: Sound-Stereo – `config.py`

```
from os import path

import pygame

WINDOW: pygame.Rect = pygame.Rect(0, 0, 800, 224)
FPS = 60
DELTATIME = 1.0 / FPS
PATH: dict[str, str] = {}
PATH["file"] = path.dirname(path.abspath(__file__))
PATH["image"] = path.join(PATH["file"], "images")
```

```python
11 PATH["sound"] = path.join(PATH["file"], "sounds")
12 START_DISTANCE = 20
13 VOLUME_STEP = 0.05
14
15 def get_file(filename: str) -> str:
16     return path.join(PATH["file"], filename)
17
18 def get_image(filename: str) -> str:
19     return path.join(PATH["image"], filename)
20
21 def get_sound(filename: str) -> str:
22     return path.join(PATH["sound"], filename)
```

Listing 2.131: Sound-Stereo – Class `Ground`

```python
11 class Ground(pygame.sprite.Sprite):
12
13     def __init__(self) -> None:
14         super().__init__()
15         fullfilename = cfg.get_image("tankbrigade_part64.png")
16         tile = pygame.image.load(fullfilename).convert()
17         rect = tile.get_rect()
18         self.image = pygame.Surface(cfg.WINDOW.size)
19         for row in range(cfg.WINDOW.width // rect.width):
20             for col in range(cfg.WINDOW.height // rect.height):
21                 self.image.blit(tile, (row * rect.width, col * rect.height))
22         self.rect = self.image.get_rect()
```

Sound object

In line 18, a `Sound` object is created. This object is played to emphasize the movement of the tank with appropriate sounds. In the following line (line 44), the helper method `stereo()` is called (see below), and then the playback of the driving sound starts in an infinite loop (line 48).

It is noticeable that the output is not started using `pygame.mixer.Sound.play()`. Normally, this would be a good choice, since this command automatically selects one of the eight available Pygame sound channels.

channel

Selecting channels manually

However, it is also possible to address a Pygame channel directly and thus gain more control over the sound behavior. In line 45, a free `pygame.mixer.Channel` object is determined for this purpose. The method `pygame.mixer.find_channel()` returns the first pygame channel and stores it in the attribute `channel`.

Playback in line 48 is then no longer started via a method of the `Sound` object, but by using `pygame.mixer.Channel.play()`.

play()

This makes it possible to adjust volume and stereo panning dynamically while the sound is playing.

Listing 2.132: Sound-Stereo – Constructor of `Tank`

```python
class Tank(pygame.sprite.Sprite):

    def __init__(self) -> None:
        super().__init__()
        self.image_filename = (209, 190, 202, 214, 226, 238, 250, 262)
        self.images: dict[str, list[pygame.Surface]] = {"up": [], "down": [], "left": [],
            "right": []}
        for number in self.image_filename:
            fullfilename = cfg.get_image(f"tankbrigade_part{number}.png")
            picture = pygame.image.load(fullfilename).convert()
            picture.set_colorkey("black")
            self.images["up"].append(picture)
            self.images["down"].append(pygame.transform.rotate(picture, 180))
            self.images["left"].append(pygame.transform.rotate(picture, +90))
            self.images["right"].append(pygame.transform.rotate(picture, -90))
        self.direction = "right"
        self.imageindex = 0
        self.image = self.images[self.direction][self.imageindex]
        self.rect = pygame.FRect(self.image.get_rect())
        self.rect.left, self.rect.top = 3 * self.rect.width, 2 * self.rect.height
        self.sound_drive = pygame.mixer.Sound(cfg.get_sound("tank_drive1.wav"))  #
        self.channel = pygame.mixer.find_channel()  # Find a free sound channel
        if self.channel:
            self.stereo()                           #
            self.channel.play(self.sound_drive, -1) #
        self.speed = 50
```

The `update()` method is shown here only for completeness. It does not contain any code related to sound playback.

Listing 2.133: Sound-Stereo – `Tank.update()`

```python
    def update(self, *args: Any, **kwargs: Any) -> None:
        if "go" in kwargs.keys():
            if kwargs["go"]:
                self.update_imageindex()
                self.image = self.images[self.direction][self.imageindex]
                if self.direction == "up" or self.direction == "left":
                    self.speed = -50
                elif self.direction == "down" or self.direction == "right":
                    self.speed = 50
                if self.direction == "up" or self.direction == "down":
                    self.rect.move_ip(0, self.speed * cfg.DELTATIME)
                    if self.rect.top <= cfg.WINDOW.top:
                        self.turn("down")
                    if self.rect.bottom >= cfg.WINDOW.bottom:
                        self.turn("up")
                elif self.direction == "left" or self.direction == "right":
                    self.rect.move_ip(self.speed * cfg.DELTATIME, 0)
                    if self.rect.left <= cfg.WINDOW.left:
                        self.turn("right")
                    if self.rect.right >= cfg.WINDOW.right:
                        self.turn("left")
                self.stereo()
        if "turn" in kwargs.keys():
            self.turn(kwargs["turn"])
```

set_volume()

The `stereo()` method is surprisingly simple. The method `pygame.mixer.Channel-.set-_volume()` provides two parameters: *left* and *right*. Both parameters have a value range of $[0, 1]$.

Stereo Panning

As discussed before, we want the right speaker to play the engine sound louder the further to the right the tank is positioned, and vice versa (stereo panning) . To achieve this, I calculate the relative horizontal position of the tank's center with respect to the window width in line 48. This calculation also yields a value in the interval $[0, 1]$.

Once this value is known, the relative value for the left speaker can be determined in the following line by $left = 1 - right$. After that, both values are passed to the `set_volume()` method.

Hint

The method `pygame.mixer.Channel.set_volume()` allows different volume levels to be specified for the left and right Pygame channels, whereas the methods `pygame.mixer.Sound.set_volume()` and `pygame.mixer.music.set_volume()` do not.

Listing 2.134: Sound-Stereo – `Tank.stereo()`

```
76      def stereo(self) -> None:
77          volume_right = self.rect.centerx / cfg.WINDOW.width  #
78          volume_left = 1 - volume_right
79          self.channel.set_volume(volume_left, volume_right)
```

What else could this effect be used for? For example, think of two people talking to each other, sound sources in a room, and so on. Whenever audio is meant to make localization easier, or when individual sounds should stand out or be easier to distinguish, different volume levels – i.e., stereo – are a good option.

Nothing related to sound output happens in `turn()` and `update_imageindex()`.

Listing 2.135: Sound-Stereo – `Tank.turn()` and `Tank.update_imageindex()`

```
81      def turn(self, direction: str) -> None:
82          self.direction = direction
83
84      def update_imageindex(self) -> None:
85          if self.speed == 0:
86              self.imageindex = 0
87          else:
88              self.imageindex = (self.imageindex + 1) % len(self.images[self.direction])
```

The sound output of the `Bullet` could also have been implemented in the `Tank` class. However, I find it more natural to place it in `Bullet`. After all, it might later be extended to include an impact sound or an explosion.

Before the constructor, the static variable `sound_fire` is defined in line 93. Although there are many bullets, they all use the same firing sound. Reading this sound file repeatedly and creating a new object each time would therefore waste memory and reduce performance. Instead, starting at line 113, a kind of singleton check is performed. This ensures that the sound file is read and the corresponding object is created exactly once.

After that, a free channel is searched for, just as with the tank, and the volume of the left and right speakers is determined based on the position. Finally, the sound is played based on the horizontal position of the bullet.

Listing 2.136: Sound-Stereo – Class `Bullet`

```python
class Bullet(pygame.sprite.Sprite):

    SOUND_FIRE = None                                # Only one shared sound is needed

    def __init__(self, tank: Tank) -> None:
        super().__init__()
        bulletspeed = 300
        number: dict[str, int] = {"left": 49, "right": 61, "up": 37, "down": 73}
        directions = {
            "left": pygame.Vector2(-bulletspeed, 0),
            "right": pygame.Vector2(bulletspeed, 0),
            "up": pygame.Vector2(0, -bulletspeed),
            "down": pygame.Vector2(0, bulletspeed),
        }
        fullfilename = os.path.join(cfg.PATH["image"],
            f"tankbrigade_part{number[tank.direction]}.png")
        self.image = pygame.image.load(fullfilename).convert()
        self.image.set_colorkey("black")
        self.rect = self.image.get_rect()
        self.direction = tank.direction
        self.rect.center = tank.rect.center
        self.speed = directions[tank.direction]

        if Bullet.SOUND_FIRE == None:                #
            Bullet.SOUND_FIRE = pygame.mixer.Sound(cfg.get_sound("tank_fire1.wav"))
        volume_right = self.rect.centerx / cfg.WINDOW.width
        volume_left = 1 - volume_right
        self.channel: pygame.mixer.Channel = pygame.mixer.find_channel()
        if self.channel:
            self.channel.set_volume(volume_left, volume_right)
            self.channel.play(Bullet.SOUND_FIRE)

    def update(self, *args: Any, **kwargs: Any) -> None:
        self.rect.move_ip(self.speed * cfg.DELTATIME)
        if not cfg.WINDOW.contains(self.rect):
            self.kill()
```

The remaining source code is shown here only for the sake of completeness.

Listing 2.137: Sound-Stereo – Rest

```python
class Game:

    def __init__(self) -> None:
        pygame.init()
        self.window = pygame.Window(size=cfg.WINDOW.size, title="Steroe Panning Sound
            Example")
        self.screen = self.window.get_surface()
        self.clock = pygame.time.Clock()

        self.ground = pygame.sprite.GroupSingle(Ground())
        self.tankreference = Tank()
        self.tank = pygame.sprite.GroupSingle(self.tankreference)
        self.all_bullets = pygame.sprite.Group()
        self.running = True

```

```
    def watch_for_events(self) -> None:
        for event in pygame.event.get():
            if event.type == QUIT:
                self.running = False
            elif event.type == KEYDOWN:
                if event.key == K_ESCAPE:
                    self.running = False
                elif event.key == K_UP:
                    self.tank.update(turn="up")
                elif event.key == K_DOWN:
                    self.tank.update(turn="down")
                elif event.key == K_LEFT:
                    self.tank.update(turn="left")
                elif event.key == K_RIGHT:
                    self.tank.update(turn="right")
                elif event.key == K_SPACE:
                    self.fire()

    def fire(self) -> None:
        if len(self.all_bullets) < 5:
            self.all_bullets.add(Bullet(self.tankreference))

    def draw(self) -> None:
        self.ground.draw(self.screen)
        self.tank.draw(self.screen)
        self.all_bullets.draw(self.screen)
        self.window.flip()

    def update(self) -> None:
        self.tank.update(go=True)
        self.all_bullets.update()

    def run(self) -> None:
        time_previous = time()
        self.running = True
        while self.running:
            self.watch_for_events()
            self.update()
            self.draw()
            self.clock.tick(cfg.FPS)
            time_current = time()
            cfg.DELTATIME = time_current - time_previous
            time_previous = time_current
        pygame.quit()
```

2.11.2.2 Sound Formats and Technical Basics

Pygame does not support all audio formats equally well. The following formats have proven to be particularly reliable:

`.wav` – uncompressed, fast to load, ideal for sound effects

`.ogg` – compressed, well suited for music

`.mp3` – limited support, not recommended

Don'ts

- Mono sounds are often used for sound effects because they require less memory and can be positioned spatially more effectively.
- Large files and high sample rates can negatively affect loading times and performance.

2.11.2.3 Volume Hierarchies and Sound Mixing

Games often use multiple volume levels:

1. Master volume (everything)
2. Music volume
3. Effect volume

These can be combined:

```
master_volume = 0.8
effects_volume = 0.5

sound.set_volume(master_volume * effects_volume)
```

This makes it easy to implement audio settings for game menus later on.

2.11.2.4 Mono Sounds and Stereo Panning

Mono sounds are particularly suitable for position-dependent audio. Only mono sounds can be cleanly distributed between the left and right speakers.

```
channel.set_volume(left, right)
```

Don'ts

Stereo sounds already contain spatial information and may produce unexpected results when additional panning is applied.

2.11.2.5 Sound Lifetime and Resource Management

Sounds should not be reloaded for every event. Instead, they should be loaded once and reused.

```
class Bullet:
        sound_fire = None

        def __init__(self):
        if sound_fire is None:
                sound_fire = pygame.mixer.Sound("fire.wav")
```

This saves memory and avoids unnecessary loading times.

2.11.2.6 Event-driven Sound Output

Sounds should be played in an event-driven manner, not frame-based.

It is incorrect to call `sound.play()` directly or indirectly inside the update loop. Instead, sounds should be triggered by events:

```
elif event.key == K_SPACE:
        sound.play()
```

2.11.2.7 Looping and Transitions

Loops are used for continuous sounds (engines, wind, music):

```
channel.play(sound, loops=-1)
```

Smooth transitions can be achieved using fade-in and fade-out effects:

```
sound.fadeout(1000)   # 1 second
```

2.11.2.8 Muting and Pausing

Many games offer an option to mute or pause sound globally.

```
pygame.mixer.pause()
...
pygame.mixer.unpause()
```

Alternatively, this can be done via volume control:

```
pygame.mixer.music.set_volume(0)
```

This is especially important for pause menus or when the game window loses focus.

2.11.2.9 Typical Errors and Debugging

Common problems with sound in Pygame include:

- Mixer not initialized
- Sound played too frequently
- No free channels available
- Distorted sound (incorrect format)
- Meaningless increase of the number of sound channels, such as `pygame.mixer.set_num_channels(16)`

Checking the current state often helps:

```
print(pygame.mixer.music.get_busy())
```

2.11.3 What was new?

Two options are available for sound support. One option is background music, while the other uses individual sounds played on different channels and, if possible, distributed across the left and right speakers.

The following Pygame elements were introduced

- `pygame.mixer.Channel` : https://pyga.me/docs/ref/music.html#pygame.mixer.Channel
- `pygame.mixer.Channel.play()`: https://pyga.me/docs/ref/mixer.html#pygame.mixer.Channel.play
- `pygame.mixer.Channel.set_volume()`: https://pyga.me/docs/ref/mixer.html#pygame.mixer.Channel.set_volume

- `pygame.mixer.find_channel()` : https://pyga.me/docs/ref/music.html#pygame.mixer.find_channel
- `pygame.mixer.init()`: https://pyga.me/docs/ref/mixer.html#pygame.mixer.init
- `pygame.mixer.set_num_channels()`: https://pyga.me/docs/ref/mixer.html#pygame.mixer.set_num_channels
- `pygame.mixer.music.fadeout()`: https://pyga.me/docs/ref/music.html#pygame.mixer.music.fadeout
- `pygame.mixer.music.get_busy()` : https://pyga.me/docs/ref/music.html#pygame.mixer.music.get_busy
- `pygame.mixer.music.get_volume()`: https://pyga.me/docs/ref/music.html#pygame.mixer.music.get_volume
- `pygame.mixer.music.load()`: https://pyga.me/docs/ref/music.html#pygame.mixer.music.load
- `pygame.mixer.music.pause()`: https://pyga.me/docs/ref/music.html#pygame.mixer.music.pause
- `pygame.mixer.music.play()`: https://pyga.me/docs/ref/music.html#pygame.mixer.music.play
- `pygame.mixer.music.set_volume()`: https://pyga.me/docs/ref/music.html#pygame.mixer.music.set_volume
- `pygame.mixer.music.unpause()`: https://pyga.me/docs/ref/music.html#pygame.mixer.music.unpause
- `pygame.mixer.Sound`: https://pyga.me/docs/ref/mixer.html#pygame.mixer.Sound
- `pygame.mixer.Sound.get_volume()`: https://pyga.me/docs/ref/mixer.html#pygame.mixer.Sound.get_volume
- `pygame.mixer.Sound.play()`: https://pyga.me/docs/ref/mixer.html#pygame.mixer.Sound.play
- `pygame.mixer.Sound.set_volume()`: https://pyga.me/docs/ref/mixer.html#pygame.mixer.Sound.set_volume

2.12 Events

2.12.1 Introduction

We have already used events (event) in two places without examining them in more detail. On the one hand, this happened when we talked about the keyboard in chapter 2.6 on page 89, and on the other hand when we discussed the mouse in chapter 2.10 on page 145.

Here, we will take a closer look at three aspects

- What information is actually contained in an event?
- How can I create an event myself?
- How can events be generated periodically?

2.12.1.1 What Information is Contained in an Event?

So ... where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/12%20Events/event00

The program shown in source code 2.138 simply creates a gray window and prints the event to the console using `print()` in line 31.

Listing 2.138: Events – outputting information

```
29      def watch_for_events(self) -> None:
30          for event in pygame.event.get():
31              print(event)                            # Print event information
32              if event.type == QUIT:
33                  self.running = False
34              elif event.type == KEYDOWN:
35                  if event.key == K_ESCAPE:
36                      self.running = False
```

If you now move the mouse back and forth, press a few keys, or close the application, something like the following will appear in the console. Many redundant lines have been removed here:

Listing 2.139: Events – console output

```
<Event(769-KeyUp {'unicode': 'd', 'key': 100, 'mod': 4096, 'scancode': 7, 'window': None})>
<Event(768-KeyDown {'unicode': 'a', 'key': 97, 'mod': 4096, 'scancode': 4, 'window': None})>
<Event(771-TextInput {'text': 'a', 'window': None})>
<Event(768-KeyDown {'unicode': ' ', 'key': 32, 'mod': 4096, 'scancode': 44, 'window': None})>
<Event(771-TextInput {'text': ' ', 'window': None})>
<Event(1024-MouseMotion {'pos': (299, 143), 'rel': (-1, 0), 'buttons': (0, 0, 0), 'touch':
    False, 'window': None})>
<Event(1024-MouseMotion {'pos': (297, 143), 'rel': (-2, 0), 'buttons': (0, 0, 0), 'touch':
    False, 'window': None})>
<Event(1025-MouseButtonDown {'pos': (230, 118), 'button': 1, 'touch': False, 'window':
    None})>
<Event(1026-MouseButtonUp {'pos': (230, 118), 'button': 1, 'touch': False, 'window': None})>
<Event(1027-MouseWheel {'flipped': False, 'x': 0, 'y': 1, 'precise_x': 0.0, 'precise_y':
    1.0, 'touch': False, 'window': None})>
<Event(1025-MouseButtonDown {'pos': (230, 118), 'button': 5, 'touch': False, 'window':
    None})>
<Event(1026-MouseButtonUp {'pos': (230, 118), 'button': 5, 'touch': False, 'window': None})>
<Event(1027-MouseWheel {'flipped': False, 'x': 0, 'y': -1, 'precise_x': 0.0, 'precise_y':
    -1.0, 'touch': False, 'window': None})>
<Event(1024-MouseMotion {'pos': (572, 0), 'rel': (3, -1), 'buttons': (0, 0, 0), 'touch':
    False, 'window': None})>
<Event(32768-ActiveEvent {'gain': 0, 'state': 1})>
<Event(32784-WindowLeave {'window': None})>
<Event(32787-WindowClose {'window': None})>
<Event(256-Quit {})>
```

At first, it becomes apparent that the event information is provided in the form of a dictionary. The first entry (the number with a hyphen followed by a name) can be accessed via `event.type`. So that you do not have to memorize these numbers, Pygame provides corresponding constants; an overview for the keyboard can be found in table 2.7 on page 101, and for the mouse in table 2.9 on page 153.

The key/value pairs inside the curly braces contain the information associated with the event. For keyboard events, this includes, for example, the representation as a Unicode character or its Unicode number. Mouse events are sensibly provided with the position and the button number. Clicking the *window close* button in the upper right corner triggers several events; the last four of the list are shown here.

We will soon see that, for user-defined events, this information can be defined according to our own requirements.

2.12.1.2 How can I Create and Use User-defined Events?

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/12%20Events/event01

As an example, I will use two simple buttons here, each of which should generate an event when the left mouse button is clicked. Inside the screen, `NOFSTARTPARTICLES` many particles move around. Using the `Stop` and `Start` buttons, the particles can be stopped and started again.

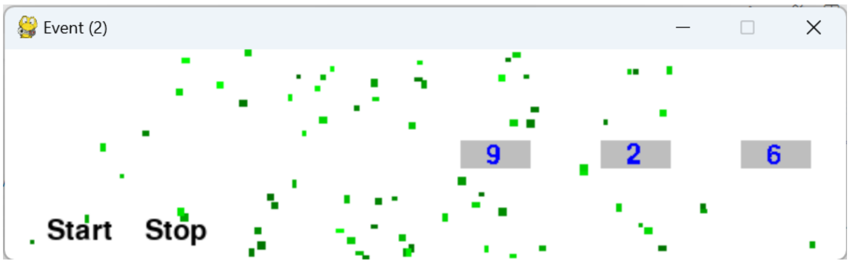

Figure 2.61: User-defined events

As an additional feature, a kind of counter is implemented. The boxes in the center absorb the particles and count them. The logic works as follows: Each time a particle hits a box, a counting event is triggered. In this process, a 1 is always added to the box on the far right.

When the rightmost box reaches the value 10, it generates an overflow to the next digit to its left and resets itself to 0. This process continues from right to left. In this way, the boxes display the total number of particles that have already been absorbed.

Now let us look at the whole setup in detail. In the console output above (see page 171), a unique number can be seen for each event, which can be used to identify the event.

> Pygame reserves a range of numbers for user-defined events between the constants `pygame.USEREVENT` and `pygame.NUMEVENTS - 1`.

For each user-defined event, such a unique number must be assigned. The simplest approach is to define these centrally using `USEREVENT +` n. You can find corresponding examples in line 11 and line 12.

I encapsulate these definitions in a static class for no other reason than that it allows me to make good use of the editor's auto-completion (line 10).

Listing 2.140: Events (2) – `config.py`

```
1  import pygame
2
3  WINDOW = pygame.Rect((0, 0), (600, 150))
4  FPS = 60
5  DELTATIME = 1.0 / FPS
6  STARTNOFPARTICLES = 999
7  NOFBOXES = 3
8  BOXWIDTH = 50
9
10 class MyEvents:                                      # Only for autocompletion (convenience)
11     BUTTONPRESSED = pygame.USEREVENT + 0             # Event id for button presses
12     OVERFLOW = pygame.USEREVENT + 1                  # Event id for overflow
```

The `Button` class should also be understandable for the most part. The first interesting section can be found in line 25. Here, a new `pygame.event.Event` object is created. As the first parameter, the previously mentioned ID must be specified. After that, any number of additional pieces of information can be passed as event data. In our example, the button text is included so that it can later be determined which button was pressed.

Event

Afterwards, in line 26, the event is dispatched using `pygame.event.post()`.

post()

Listing 2.141: Events (2) – Class `Button`

```
12 class Button(pygame.sprite.Sprite):
13 
14     def __init__(self, text: str, position: Tuple[int], *groups: Tuple[pygame.sprite.Group])
        -> None:
15         super().__init__(*groups)
16         self.font = pygame.font.SysFont(None, 30)
17         self.centerxy = (cfg.WINDOW.centerx, self.font.get_height() // 2)
18         self.text = text
19         self.image = self.font.render(self.text, True, "black")
20         self.rect = self.image.get_rect(topleft=(position))
21 
22     def update(self, *args: Any, **kwargs: Any) -> None:
23         if "action" in kwargs.keys():
24             if kwargs["action"] == "pressed":
25                 evt = pygame.event.Event(MyEvents.BUTTONPRESSED, text=self.text)  #
26                 pygame.event.post(evt)                #
27         return super().update(*args, **kwargs)
```

The `Particle` class consists of a lot of source code with little that is new. Particles of random size, color, direction, and speed move across the screen and may bounce off the edges. They do not contain any event-specific functionality. The attribute `halted` is used to stop the particle or let it move again after the buttons have been pressed.

Listing 2.142: Events (2) – Class `Particle`

```
class Particle(pygame.sprite.Sprite):

    def __init__(self, *groups: Tuple[pygame.sprite.Group]) -> None:
        super().__init__(*groups)
        self.image = pygame.Surface((randint(3, 6), randint(3, 6)))
        self.image.fill((0, randint(100, 255), 0))
        self.rect = pygame.FRect(self.image.get_rect())
        self.rect.topleft = (
            randint(30, cfg.WINDOW.right - 30),
            randint(30, cfg.WINDOW.bottom - 30),
        )
        self.speed = randint(100, 400)
        self.direction = pygame.Vector2(choice((-1, 1)), choice((-1, 1)))
        self.halted = False

    def update(self, *args: Any, **kwargs: Any) -> None:
        if "action" in kwargs.keys():
            if kwargs["action"] == "move":
                if not self.halted:
                    self._move()
            elif kwargs["action"] == "Start":
                self.halted = False
            elif kwargs["action"] == "Stop":
                self.halted = True
```

```

    def _move(self) -> None:
        self.rect.move_ip(self.speed * self.direction * cfg.DELTATIME)
        if self.rect.left < cfg.WINDOW.left or self.rect.right > cfg.WINDOW.right:
            self.direction[0] *= -1
        if self.rect.top < cfg.WINDOW.top or self.rect.bottom > cfg.WINDOW.bottom:
            self.direction[1] *= -1
        self.rect.clamp_ip(cfg.WINDOW)
```

With Box, a kind of digit box is implemented. The constructor receives a position and an index as parameters. The meaning of the parameter `position` should be clear. Using `index`, it can later be determined which box caused an overflow to the next higher power of ten.

In `update()`, the internal counter `count` is increased by 1 each time. If the value 10 is reached (line 79), an event is generated and the index is passed as event data. This allows the main program to determine which box now needs to receive an `update()` call.

Listing 2.143: Events (2) – Klasse `Box`

```
64 class Box(pygame.sprite.Sprite):
65 
66     def __init__(self, index: int, position: Tuple[int], *groups:
        Tuple[pygame.sprite.Group]) -> None:
67         super().__init__(*groups)
68         self.image = pygame.Surface((cfg.BOXWIDTH, 20))
69         self.rect = self.image.get_rect(center=position)
70         self.font = pygame.font.SysFont(None, 30)
71         self.counter = 0
72         self.index = index
73         self.fill()
74 
75     def update(self, *args: Any, **kwargs: Any) -> None:
76         if "counter" in kwargs.keys():
77             if kwargs["counter"] == "inc":
78                 self.counter += 1
79                 if self.counter == 10:               # Overflow
80                     evt = pygame.event.Event(MyEvents.OVERFLOW, index=self.index)
81                     pygame.event.post(evt)
82                     self.counter = 0
83                 self.fill()
84         return super().update(*args, **kwargs)
85 
86     def fill(self) -> None:
87         self.image.fill("gray")
88         number = self.font.render(f"{self.counter}", False, "Blue")
89         self.image.blit(number, (18, 1))
```

And now the main program: In the constructor, the buttons, boxes, and particles are created and assigned to sprite groups.

Listing 2.144: Events (2) – Constructor of `Game`

```
class Game:

    def __init__(self) -> None:
        pygame.init()
        self.window = pygame.Window(size=cfg.WINDOW.size, title="Event (1)",
            position=WINDOWPOS_CENTERED)
```

```
        self.screen = self.window.get_surface()
        self.clock = pygame.time.Clock()
        self.running = True
        self.all_sprites = pygame.sprite.Group()
        self.all_particles = pygame.sprite.Group()
        self.generate_particles(cfg.STARTNOFPARTICLES)
        self.all_buttons = pygame.sprite.Group()
        self.all_buttons.add(Button("Start", (30, cfg.WINDOW.bottom - 30), self.all_sprites))
        self.all_buttons.add(Button("Stop", (100, cfg.WINDOW.bottom - 30), self.all_sprites))
        self.all_boxes = pygame.sprite.Group()
        self.generate_boxes(cfg.NOFBOXES)
        self.running = True
```

The `run()` method is almost boring.

Listing 2.145: Events (2) – `Game.run()`

```
    def run(self) -> None:
        time_previous = time()
        while self.running:
            self.watch_for_events()
            self.update()
            self.draw()
            self.clock.tick(cfg.FPS)
            time_current = time()
            cfg.DELTATIME = time_current - time_previous
            time_previous = time_current
        pygame.quit()
```

User-defined events are handled in exactly the same way as predefined ones.

First, you check the `type`, and then you process the event data. In line 132, it is checked whether one of the two buttons was pressed. Afterwards, the message is forwarded to the particles via the event field `text`, telling them whether they should stop or keep moving. The same idea is used starting at line 134. First, it is checked whether a box has overflowed, and then the next box is informed – using the event field `index` – that it has to increase by 1.

Listing 2.146: Events (2) – `Game.watch_for_events()`

```
122     def watch_for_events(self) -> None:
123         for event in pygame.event.get():
124             if event.type == QUIT:
125                 self.running = False
126             elif event.type == KEYDOWN:
127                 if event.key == K_ESCAPE:
128                     self.running = False
129             elif event.type == MOUSEBUTTONDOWN:
130                 if event.button == 1:
131                     self.check_button_pressed(event.pos)
132             elif event.type == MyEvents.BUTTONPRESSED:  #
133                 self.all_particles.update(action=event.text)
134             elif event.type == MyEvents.OVERFLOW:    #
135                 if event.index < cfg.NOFBOXES - 1:
136                     self.all_boxes.sprites()[event.index + 1].update(counter="inc")
```

The rest is shown here for completeness.

Listing 2.147: Events (2) – The Rest of `Game`

```
    def update(self):
        self.all_buttons.update()
        self.all_particles.update(action="move")
        self.check_boxcollision()

    def draw(self) -> None:
        self.screen.fill("white")
        self.all_sprites.draw(self.screen)
        self.window.flip()

    def generate_boxes(self, number: int) -> None:
        for i in range(number):
            self.all_boxes.add(Box(i, (cfg.WINDOW.right - 50 - i * 100, cfg.WINDOW.centery),
                self.all_sprites))

    def generate_particles(self, number: int) -> None:
        for i in range(number):
            self.all_particles.add(Particle(self.all_sprites))

    def check_button_pressed(self, position: Tuple[int]) -> None:
        for b in self.all_buttons.sprites():
            if b.rect.collidepoint(position):
                b.update(action="pressed")

    def check_boxcollision(self) -> None:
        c = pygame.sprite.groupcollide(self.all_particles, self.all_boxes, True, False)
        for _ in c:
            self.all_boxes.sprites()[0].update(counter="inc")
```

Finally, I would like to briefly explain the whole mechanism of the counter again using figure 2.62 on the next page, in order to make it clearer how the event `MyEvent.OVERFLOW` works in this context.

- t_0: First, a standard collision check determines that a particle has hit the rectangle of the rightmost field (see figure 2.61 on page 172, the box with the 6). As a result, the method `update(counter="inc")` is called.

- t_1: This causes the event `MyEvents.OVERFLOW` to be triggered in the `Box` class with the index value 0. This event is caught in `watch_for_event()` and forwarded to the appropriate box – that is, the one with index `index+1` – together with the instruction to also execute `update(counter="inc")` there.

- t_2: Since this box currently contains the value 9, `MyEvents.OVERFLOW` is triggered again in this box, but now with the next index value, namely 1. This event is again caught in `watch_for_event()` and forwarded to box 2 with `update(counter="inc")`.

- t_3: The value of the leftmost box is currently 0 and is increased by 1 by the call to `update(counter="inc")`. Since no overflow is generated in this case, the chain of events stops here.

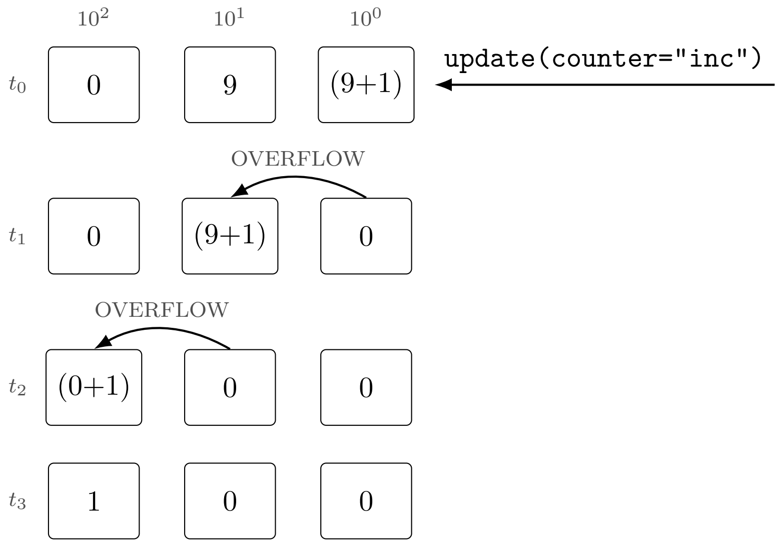

Figure 2.62: How the counter works

2.12.2 More Input

2.12.2.1 How can periodic events be generated?

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/00%20Introduction/12%20Events/event02

This is actually quite simple. The previous example is extended so that new particles are created at intervals of 500 ms. To achieve this, a new ID `NEWPARTICLES` is first defined for the user event.

Listing 2.148: Events (3) – `config.py`

```
class MyEvents:
    BUTTONPRESSED = pygame.USEREVENT + 0
    OVERFLOW = pygame.USEREVENT + 1
    NEWPARTICLES = pygame.USEREVENT + 2


WINDOW = pygame.Rect((0, 0), (600, 150))
FPS = 60
DELTATIME = 1.0 / FPS
STARTNOFPARTICLES = 500
NEWNOFPARTICLES = 10
NOFBOXES = 5
BOXWIDTH = 50
```

set_timer()

In the constructor of `Game`, a periodic timer is set in line ?? using `pygame.time.set_timer()`. This timer fires the corresponding event ID every 500 ms.

Listing 2.149: Events (3) – `Timer` in the constructor of `Game`

```
        self.generate_particles(cfg.STARTNOFPARTICLES)
        self.all_buttons = pygame.sprite.Group()
        self.all_buttons.add(Button("Start", (30, cfg.WINDOW.bottom - 30), self.all_sprites))
```

Like the other events, this one is caught in `watch_for_event()` (line ??) and processed. In this case, this is done by calling the method `generate_particles()`.

Listing 2.150: Events (3) – Catching a periodical event

```
            elif event.type == MyEvents.BUTTONPRESSED:
                self.all_particles.update(action=event.text)
```

2.12.2.2 Structuring the Event Loop Correctly

Best practice

In each frame, the event loop should follow a clear and consistent structure:

1. Retrieve all events from the event queue
2. Process the events
3. Update the game state
4. Render the scene

```
...
        for event in pygame.event.get():
                if event.type == pygame.QUIT:
                        running = False
        handle_event(event)
...
        update()
        draw()
```

Common mistakes

- ⚠ Failing to regularly process events can cause the application window to become unresponsive.
- ⚠ Forgetting to call `pygame.event.get()` or processing events only sporadically.

⚠ Using `event.wait()` in the main loop, which can block rendering and updates.

2.12.2.3 Choosing the Right Event Retrieval Method

Pygame provides different ways to access events:

- `pygame.event.get()` retrieves all pending events (standard choice for games)
- `pygame.event.poll()` retrieves a single event
- `pygame.event.wait()` blocks until an event occurs

2.12.2.4 Avoid Generating Events Every Frame

Events should represent state changes, not continuous states

Common mistake: Posting custom events or playing sounds inside the `update()` method every frame, which can flood the event queue and cause performance issues. Generate events only when a condition changes (edge-triggered behavior).

```
if not was_pressed and is_pressed:
    pygame.event.post(pygame.event.Event(MyEvents.FIRE))
    was_pressed = is_pressed
```

2.12.2.5 Defining Event Data Clearly and Consistently

When creating custom events, meaningful and consistent event data should be attached.

```
pygame.event.post(pygame.event.Event(MyEvents.BUTTON, {"action": "start"}))
```

Use clear, descriptive keys (e.g. action, index, pos) and stick to a consistent naming scheme (snake_case). Using inconsistent field names across different events, making event handling error-prone.

2.12.2.6 Managing User-Defined Event IDs

Best practice

Define all custom event IDs centrally, for example in a configuration file.

```
BUTTON_EVENT   = pygame.USEREVENT + 1
OVERFLOW_EVENT = pygame.USEREVENT + 2
NEWPARTICLES   = pygame.USEREVENT + 3
```

2.12.2.7 Use `set_timer()` Correctly!

Best practice

Set timers only once (e.g. in the constructor) and disable timers when they are no longer needed.

```
        pygame.time.set_timer(NEWPARTICLES, 0)
```

Setting the same timer multiple times and forgetting to disable timers when restarting a game or switching scenes.

2.12.2.8 Filtering Events for Performance

In event-heavy applications, it may be useful to restrict which events are allowed.

```
    pygame.event.set_allowed([pygame.QUIT,
                              pygame.KEYDOWN,
                              pygame.MOUSEBUTTONDOWN,
                              NEWPARTICLES])
```

Use event filtering sparingly and only when necessary. Blocking too many events can accidentally preventing essential input from being processed.

2.12.2.9 Event-Based Input vs. State-Based Input

There are two complementary approaches to input handling:

1. Event-based input: Reacts to discrete events (`KEYDOWN`, `KEYUP`) and is ideal for actions like shooting, opening menus, or triggering sounds.

2. State-based input: Uses continuous state queries (`key.get_pressed()`) and can be used for movement and continuous control.

Combine both approaches appropriately.

2.12.2.10 Window Focus and Application State

Best practice

Games should respond appropriately when the window loses focus or is minimized. Pause the game or mute sound when focus is lost and always handle the QUIT event reliably.

2.12.2.11 Debugging Events Effectively

Printing events to the console is useful during development but should be done selectively.

```
for event in pygame.event.get():
        if event.type != pygame.MOUSEMOTION:
                print(event)
```

Common mistake

Logging every mouse movement event, which can overwhelm the console and reduce performance.

2.12.2.12 Structuring Event Handling Code

Best practice

As projects grow, large if–elif blocks become hard to maintain. Use handler functions or dispatch tables.

```
handlers = {
        pygame.KEYDOWN: handle_keydown,
        pygame.MOUSEBUTTONDOWN: handle_mouse,
        NEWPARTICLES: handle_newparticles,
}

for event in pygame.event.get():
        handlers.get(event.type, handle_default)(event)
```

This approach improves readability and scalability.

2.12.3 What was new?

The advantage of user-defined events becomes very clear here. If this were implemented in a different way, the objects would have to know about each other. For example, all boxes would have to know their predecessor or successor via references in order to report an overflow. While this can also be a valid approach, using events decouples the classes, and the main program can control and organize the forwarding of information via the event data.
In particular, clicking on the buttons can be implemented very easily using events.

The following Pygame elements were introduced

- `USEREVENT` : https://pyga.me/docs/ref/event.html#pygame.event
- `NUMEVENTS` : https://pyga.me/docs/ref/event.html#pygame.event
- `pygame.event.Event`: https://pyga.me/docs/ref/event.html#pygame.event.Event
- `pygame.event.get()` : https://pyga.me/docs/ref/event.html#pygame.event.get
- `pygame.event.poll()` : https://pyga.me/docs/ref/event.html#pygame.event.poll
- `pygame.event.post()`: https://pyga.me/docs/ref/event.html#pygame.event.post
- `pygame.event.wait()`: https://pyga.me/docs/ref/event.html#pygame.event.wait
- `pygame.time.set_allowed()`: https://pyga.me/docs/ref/time.html#pygame.time.set_allowed
- `pygame.time.set_timer()`: https://pyga.me/docs/ref/time.html#pygame.time.set_timer
- `pygame.WINDOWPOS_CENTERED`: https://pyga.me/docs/ref/window.html#pygame.Window.position

3 Techniques

3.1 Animation

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/01%20Techniques/01%20Animation

An animation is essentially a small *movie* inside a game. Examples of useful animations include movements, explosions, pulsing effects, and changes in appearance. Here, I would like to present two examples: a small movement and an explosion.

3.1.1 The Running Cat

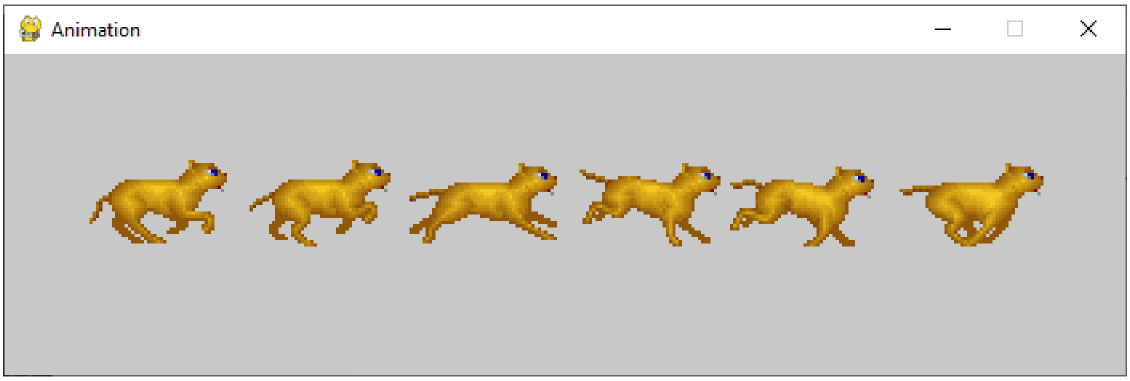

Figure 3.1: Animation of a cat: frame sprites

You can see the individual frames of the movement example in figure 3.1. If these individual sprites are displayed one after another at a certain speed, they appear as a smooth movement. The following rule applies: the more individual frames are used, the smoother the animation appears.

At first the `config.py`:

Listing 3.1: The running cat, `config.py`

```
from os import path

import pygame

WINDOW = pygame.Rect((0, 0), (300, 200))
```

```
FPS = 60
DELTATIME = 1.0 / FPS
TITLE = "Animation"
PATH: dict[str, str] = {}
PATH["file"] = path.dirname(path.abspath(__file__))
PATH["image"] = path.join(PATH["file"], "images")

@staticmethod
def filepath(name: str) -> str:
    return path.join(PATH["file"], name)

@staticmethod
def imagepath(name: str) -> str:
    return path.join(PATH["image"], name)
```

The source code in source code 3.2 differs from the chapter above (see section 2.9.2.1 on page 141 by only one feature. The `Timer` class has been extended by the method `change_duration()`. This method makes it possible to change the duration of the time interval at runtime, with a lower limit of 0 ms. We will use this feature shortly to manually adjust the animation speed.

Listing 3.2: The running cat (1), Version 1.0 – `Timer`

```
class Timer:

    def __init__(self, duration: int, with_start: bool = True):
        self.duration = duration
        if with_start:
            self.next = pygame.time.get_ticks()
        else:
            self.next = pygame.time.get_ticks() + self.duration

    def is_next_stop_reached(self) -> bool:
        if pygame.time.get_ticks() > self.next:
            self.next = pygame.time.get_ticks() + self.duration
            return True
        return False

    def change_duration(self, delta: int = 10):
        self.duration += delta
        if self.duration < 0:
            self.duration = 0
```

If we want to animate something, this animation does not require just a single sprite for display, but several. For this reason, in addition to the `image` attribute, I introduced another one: the list `images`. Using a `for`-loop starting at line 34, I now load all bitmaps of the animation into this list.

We now need an attribute that keeps track of which of the 6 sprites should currently be displayed: `imageindex`. If the images are stored in the `images` array in the same order in which they are supposed to be displayed, `imageindex` only needs to be incremented. We also need a `Timer` object so that the animation does not run absurdly fast – we start here with 100 ms.

In the `update()` method, the `imageindex` attribute is incremented by 1 depending on the `Timer` object, and the corresponding bitmap is then assigned to the `image` attribute so that the familiar `Sprite` features can be used. The method `change_animation_time()`

simply forwards its parameter to the `Timer` object. With this, all preparatory steps are essentially complete.

Listing 3.3: The running cat (2), Version 1.0 – `Cat`

```
class Cat(pygame.sprite.Sprite):

    def __init__(self) -> None:
        super().__init__()
        self.images: list[pygame.Surface] = []
        for i in range(6):                               # Load animation sprites
            bitmap = pygame.image.load(cfg.imagepath(f"cat{i}.bmp")).convert()
            bitmap.set_colorkey("black")
            self.images.append(bitmap)
        self.imageindex = 0
        self.image: pygame.Surface = self.images[self.imageindex]
        self.rect: pygame.Rect = self.image.get_rect()
        self.rect.center = cfg.WINDOW.center
        self.animation_time = Timer(100)

    def update(self, *args: Any, **kwargs: Any) -> None:
        if "animation_delta" in kwargs.keys():
            self.change_animation_time(kwargs["animation_delta"])
        if self.animation_time.is_next_stop_reached():
            self.imageindex += 1
            if self.imageindex >= len(self.images):
                self.imageindex = 0
            self.image = self.images[self.imageindex]
            # implement game logic here

    def change_animation_time(self, delta: int) -> None:
        self.animation_time.change_duration(delta)
```

The `CatAnimation` class is merely the usual encapsulation of the main program. In line 67, the `Cat` object is created and placed into a `GroupSingle`.

Listing 3.4: The running cat (3), Version 1.0 – Constructor and `run()`

```
58 class CatAnimation:
59 
60     def __init__(self) -> None:
61         pygame.init()
62         self.window = pygame.Window(size=cfg.WINDOW.size, title=cfg.TITLE)
63         self.screen = self.window.get_surface()
64         self.clock = pygame.time.Clock()
65 
66         self.font = pygame.font.Font(pygame.font.get_default_font(), 12)
67         self.cat = pygame.sprite.GroupSingle(Cat()) # My cat sprite
68         self.running = False
69 
70     def run(self) -> None:
71         time_previous = time()
72         self.running = True
73         while self.running:
74             self.watch_for_events()
75             self.update()
76             self.draw()
77             self.clock.tick(cfg.FPS)
78             time_current = time()
79             cfg.DELTATIME = time_current - time_previous
80             time_previous = time_current
81         pygame.quit()
```

In `watch_for_events()`, the only noteworthy aspect is that + and the - key are used to manipulate the animation speed. To increase the animation speed, the time interval of the `Timer` object has to be reduced, hence `-10`. To slow down the animation, the time interval of the `Timer` object has to be increased, hence `+10`.

Listing 3.5: The running cat (4), Version 1.0 – `watch_for_events()`

```
    def watch_for_events(self) -> None:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                self.running = False
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_ESCAPE:
                    self.running = False
                elif event.key == pygame.K_PLUS:
                    self.cat.sprite.update(animation_delta=-10)
                elif event.key == pygame.K_MINUS:
                    self.cat.sprite.update(animation_delta=10)
```

The remaining source code (source code 3.6) should be self-explanatory. When you start the program, an animated cat movement will be displayed. Feel free to try changing the animation speed.

Listing 3.6: The running cat (5), Version 1.0 – `update()` and `draw()`

```

    def draw(self) -> None:
        self.screen.fill("gray")
        self.cat.draw(self.screen)
        text_image = self.font.render(f"animation time: {self.cat.sprite.animation_time.duration}", True, "white")
        text_rect = text_image.get_rect()
        text_rect.centerx = cfg.WINDOW.centerx
        text_rect.bottom = cfg.WINDOW.bottom - 50
        self.screen.blit(text_image, text_rect)
        self.window.flip()
```

3.1.2 The Class Animation

As with time control, I am bothered by the fact that the animation logic is spread across the `Cat` class, which in my opinion violates the SRP. So let us simply build a dedicated animation class (see source code 3.7 on the next page).

Let us take a look at the constructor parameters:

- **namelist**: A list of file names without path information. These are resolved automatically using the entries in `config.py`. The order of the file names must correspond to the animation order.
- **endless**: This flag controls whether the animation repeats indefinitely. `True` means that after the last sprite, the animation starts again with the first one. `False` means that the last sprite remains displayed.

- **animationtime**: The delay between individual sprites in ms.
- **colorkey**: This parameter handles the case where sprites may not have transparency and therefore require an explicit transparency color (see page 47). If no value is provided, the transparency of the loaded sprite is kept as is. If a color value is provided, it is applied using `set_colorkey()` in line 20.

In the `next()` method, the next `imageindex` is calculated and the corresponding sprite is returned. For this purpose, the internal `Timer` object is used so that the sprites appear with a defined time interval. The `imageindex` attribute is increased by 1 and then checked to see whether the end of the sprite list has been reached. If the animation is set to *endless*, the `imageindex` is reset to 0; otherwise, the last image of the list is displayed permanently.

> **Question to the audience**
>
> Why was `imageindex` initialized to -1 in the constructor?

A feature that is often needed has been implemented in the `is_ended()` method. Frequently, the code that triggered the animation needs to know whether the animation has finished. We will make use of this later on.

Listing 3.7: The running cat (6), Version 1.1 – `Animation`

```
 9 class Animation:
10
11     def __init__(self, namelist: list[str], endless: bool, animationtime: int, colorkey:
           tuple[int, int, int] | None = None) -> None:
12         self.images: list[pygame.Surface] = []
13         self.endless = endless
14         self.timer = Timer(animationtime)
15         for filename in namelist:
16             if colorkey == None:
17                 bitmap = pygame.image.load(cfg.imagepath(filename)).convert_alpha()
18             else:
19                 bitmap = pygame.image.load(cfg.imagepath(filename)).convert()
20                 bitmap.set_colorkey(colorkey)      # Enable transparency
21             self.images.append(bitmap)
22         self.imageindex = -1
23
24     def next(self) -> pygame.Surface:
25         if self.timer.is_next_stop_reached():
26             self.imageindex += 1
27             if self.imageindex >= len(self.images):
28                 if self.endless:
29                     self.imageindex = 0
30                 else:
31                     self.imageindex = len(self.images) - 1
32         return self.images[self.imageindex]
33
34     def is_ended(self) -> bool:
35         if self.endless:
36             return False
37         return self.imageindex >= len(self.images) - 1
```

This simplifies the `Cat` class, allowing it to focus again on its – admittedly still nonexistent – game logic. The `Animation` object is created here in line 65. The file names can be generated very easily, since they are numbered consecutively. The cat is supposed to run endlessly, with a time interval of 100 ms between the sprites. In `update()`, the `next()` method is then simply called.

Listing 3.8: The running cat (7), Version 1.1 – `Cat`

```
61 class Cat(pygame.sprite.Sprite):
62
63     def __init__(self) -> None:
64         super().__init__()
65         self.animation = Animation([f"cat{i}.bmp" for i in range(6)], True, 100, (0, 0, 0))
               #
66         self.image: pygame.Surface = self.animation.next()
67         self.rect: pygame.Rect = self.image.get_rect()
68         self.rect.center = cfg.WINDOW.center
69
70     def update(self, *args: Any, **kwargs: Any) -> None:
71         if "animation_delta" in kwargs.keys():
72             self.change_animation_time(kwargs["animation_delta"])
73         self.image = self.animation.next()
74         # implement game logic here
75
76     def change_animation_time(self, delta: int) -> None:
77         self.animation.timer.change_duration(delta)
```

3.1.3 The Exploding Rock

My second example spawns rocks (meteors) at random positions and at random time intervals. Each rock is also given a certain lifetime – again chosen randomly. After that, it explodes. This explosion is animated.

Let us first take a look at the `Rock` class. In line 66, a random number is generated, which is then used in the following line to load one of four possible rock bitmaps. After that, the coordinates of the rock's center are determined using a random number generator, while keeping a certain distance from the screen borders. In line 71, the `Animation` object is created. Here again, the file names of the animation bitmaps are loaded in the order of the animation. You can see these bitmaps in figure 3.2 on the facing page.

Since the animation should not repeat, the corresponding parameter is set to `False` here. After the explosion, the rock is supposed to disappear. The delay between the individual frames is set to 50 ms. In line 72, the lifetime of the rock is again determined randomly and a corresponding `Timer` object is created – as you can see, these are quite useful and can be reused often. The flag `bumm` is a marker that indicates whether the rock is currently exploding.

The `update()` method has now become quite interesting. First, the `Timer` object is used to check whether the end of the lifetime has been reached. If not, nothing happens here, although one could implement movement or some other meaningful behaviour in the `else` branch. If the lifetime has been reached, the corresponding flag is set. Depending on this, the animation is then started.

What is the purpose of the three lines starting at line 80? They serve purely visual purposes. The dimensions of the explosion sprites are not always the same, and the `rect` object always aligns them to the upper-left corner, which would result in a visible jitter. To avoid this, the old center position is stored, the new rectangle of the next animation sprite is calculated, and its center is set to the previous position. This keeps the animation nicely aligned to the original center of the rock.

Finally, it is checked whether the animation has finished. If so, the sprite is no longer needed and can be removed from the sprite group using `kill()`.

kill()

Listing 3.9: The exploding rock (1) – `Rock`

```
62 class Rock(pygame.sprite.Sprite):
63
64     def __init__(self):
65         super().__init__()
66         rocknb = random.randint(6, 9)                 # Rock image number
67         self.image = pygame.image.load(cfg.imagepath(f"felsen{rocknb}.png")).convert_alpha()
68         self.rect = self.image.get_rect()
69         self.rect.centerx = random.randint(self.rect.width, cfg.WINDOW.width -
               self.rect.width)
70         self.rect.centery = random.randint(self.rect.height, cfg.WINDOW.height -
               self.rect.height)
71         self.anim = Animation([f"explosion0{i}.png" for i in range(1, 5)], False, 50)  #
72         self.timer_lifetime = Timer(random.randint(100, 2000), False)  # Lifetime timer
73         self.bumm = False
74
75     def update(self, *args: Any, **kwargs: Any) -> None:
76         if self.timer_lifetime.is_next_stop_reached():
77             self.bumm = True
78         if self.bumm:
79             self.image = self.anim.next()
80             c = self.rect.center                      # Center position
81             self.rect = self.image.get_rect()
82             self.rect.center = c
83         if self.anim.is_ended():
84             self.kill()
```

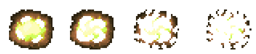

Figure 3.2: The exploding rock: frame sprites

The `ExplosionAnimation` class should no longer pose any difficulty for you. There are only a few places that I would like to briefly address. In line 96, a `Timer` object is created that is supposed to spawn two rocks per second, and in line 121 this timer is checked.

Listing 3.10: The exploding rock (2) – `ExplosionAnimation`

```
class ExplosionAnimation(object):

    def __init__(self) -> None:
        pygame.init()
        self.window = pygame.Window(size=cfg.WINDOW.size, title=cfg.TITLE)
        self.screen = self.window.get_surface()
        self.clock = pygame.time.Clock()

        self.all_rocks = pygame.sprite.Group()
        self.timer_newrock = Timer(500)             # Timer
        self.running = False

    def run(self) -> None:
        time_previous = time()
        self.running = True
        while self.running:
            self.watch_for_events()
            self.update()
            self.draw()
            self.clock.tick(cfg.FPS)
            time_current = time()
            cfg.DELTATIME = time_current - time_previous
            time_previous = time_current
        pygame.quit()

    def watch_for_events(self) -> None:
        for event in pygame.event.get():
            if event.type == QUIT:
                self.running = False
            elif event.type == KEYDOWN:
                if event.key == K_ESCAPE:
                    self.running = False

    def update(self) -> None:
        if self.timer_newrock.is_next_stop_reached():  # 500ms?
            self.all_rocks.add(Rock())
        self.all_rocks.update()

    def draw(self) -> None:
        self.screen.fill("black")
        self.all_rocks.draw(self.screen)
        self.window.flip()
```

Hint

There is also the source file `animation03.py`. In this variant, the rocks move and explode when they collide with each other. Take a look!

3.2 Tiles Are Beautiful

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/01%20Techniques/02%20Tilemap

Very often, the visual appearance of games consists of many small and large tiles that are assembled in an appropriate way. These tiles are usually combined into larger bitmaps (SpriteLib) and then have to be cut out correctly by the game developer. In figure 3.3, you can find such a simple sprite library. Spritelib

I will now show you how tiles can be cut out of a sprite library and used to assemble your own worlds. The required information is stored in a CSV files. I will also use several example levels (not to be confused with difficulty levels or floors within the game world) in order to address different types of sprites in different ways.

Hint

There is an excellent tool that helps you create such levels and integrate them into a game. It is called Tiled and can be downloaded from MapEditor. Since there are already sufficiently detailed and high-quality introductions available for this software, I will omit a description here.

3.2.1 Our Example

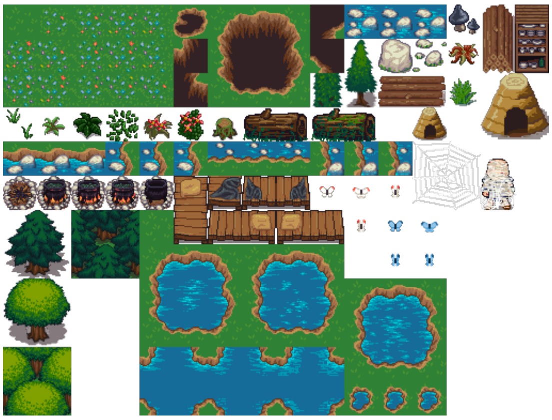

Figure 3.3: Sprite library (original)

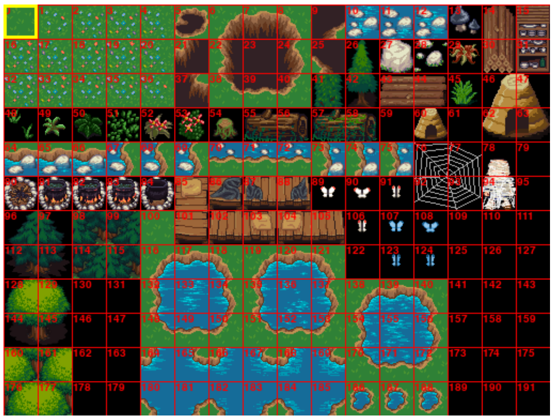

Figure 3.4: Sprite library (prepared)

In figure 3.3, we can see such a sprite library. It contains tiles for building a landscape consisting of lakes, meadows, and forests, along with some gadget. In figure 3.4, I have Gadget made the individual tiles visible by adding a grid and numbering them. The numbers will become important again later. Our tiles have a width and a height of 32 px each; they are arranged in 16 columns and 12 rows.

The goal is a game surface as shown in figure 3.5. The playing field – coincidentally – also has 16 columns and 12 rows.

Figure 3.5: Example of tile based playground

3.2.2 A Green Meadow

In the first step, I will show how a single tile can be used to fill the entire game area. So let us get started. In source code 3.11, we first find the usual suspects. The parameters `TILESIZE`, `TILEMAP_NOF_COLS`, `TILEMAP_NOF_ROWS`, and `TILEMAP_WINDOW` should also be self-explanatory.

Listing 3.11: Forest – `config.py`

```
import pygame

FPS = 60
DELTATIME = 1.0 / FPS
TILESIZE = pygame.math.Vector2(32, 32)
TILEMAP_NOF_COLS = 16
TILEMAP_NOF_ROWS = 12
TILEMAP_WINDOW = pygame.Rect(0, 0,
                                  TILEMAP_NOF_COLS * TILESIZE.x,
                                  TILEMAP_NOF_ROWS * TILESIZE.y)
```

Also `Game` is very easy to understand.

Listing 3.12: Forest – `Game`

So far, everything happens inside the `WindowGame` class, which will of course change later on. In the constructor, the window is created with all its parameters and the sprite library is loaded. In `draw()`, a single tile is now cut out of the sprite library and stored in the variable `image`. The crucial part here is the parameters passed to `subsurface()`. Starting at position $(0, 0)$ in the sprite library – i. e. the top-left corner – a rectangle of size 32 px × 32 px is cut out. This corresponds to the yellow-bordered rectangle 0 in figure 3.4 on page 191.

subsurface()

Using the two `for` loops, this image is then distributed across the entire surface. In each loop iteration, the variable `position` is calculated from the row number `row` ($[0-12]$) and the column number `col` ($[0-16]$). Multiplying the column number by the tile width yields the x-position, and multiplying the row number by the tile height yields the y-position at which the tile should be drawn.

The result then looks as shown in figure 3.6 on page 196.

Listing 3.13: Forest – `WindowGame`

```
7  class WindowGame:
8
9      def __init__(self) -> None:
10         self.window = pygame.Window(size=cfg.TILEMAP_WINDOW.size)
11         self.screen : pygame.Surface = self.window.get_surface()
12         self.spritelib = pygame.image.load("images/forest_tiles.png").convert_alpha()
13         self.rect = self.screen.get_frect()
14         self.window.title = "Tilemap Example"
15         self.clock = pygame.time.Clock()
16
17     def draw(self) -> None:
18         self.screen.fill("black")
19         image = self.spritelib.subsurface(pygame.Rect((0, 0), cfg.TILESIZE))
20         for row in range(cfg.TILEMAP_NOF_ROWS):
21             for col in range(cfg.TILEMAP_NOF_COLS):
22                 position = col * cfg.TILESIZE.x, row * cfg.TILESIZE.y
23                 self.screen.blit(image, position)
24         self.window.flip()
```

However, I would like to be able to select and display any arbitrary tile. For example, instead of a monotonous meadow, we might want it to be composed of the tiles with the numbers $0, 1, 2, 3, 4, 16, 17, 18, 19, 20, 32, 33, 34, 35, 36$. This practically calls for a separate class.

In the constructor (see source code 3.14 on the next page), only the bitmap of the sprite library is loaded. The method `subsurface()` expects the tile number as a parameter. The tiles are assumed to be numbered from left to right and from top to bottom. This makes it fairly easy to compute the column number and the row number from the tile number.

The column number is the value that remains after all complete rows have been removed. Example: The tile number is 34. I want to determine the column number. All complete rows above tile 34 are irrelevant. Therefore, with 16 columns, from

$$34 \to 18 \to 2.$$

Mathematically, this is the remainder (modulo) of an integer division. Do you remember? Elementary school? *15 divided by 6 is 2 remainder 3.* Or, in our case,

$$34 \bmod 16 = 2.$$

This is exactly what happens in line 13; the only difference is that the result is then multiplied by the number of pixels per column in order to obtain the left position of the tile.

The row number – that is, the row in which tile 34 is located – is determined in a similar, but slightly different way. In other words: how many complete rows – i. e. 16 tiles – are contained in the tile number? For this, we use integer division:

$$34 \div 16 = 2.$$

And since in computer science we always start counting at 0, the tile is indeed located in row 2. Take a look at figure 3.4 on page 191! In line 13, the row number is then multiplied by the row height to obtain the top position of the tile. The rest should be self-explanatory.

Listing 3.14: Forest – `Spritelib`

```
8  class Spritelib:
9      def __init__(self, filename: str) -> None:
10         self.image = pygame.image.load(filename).convert_alpha()
11 
12     def subsurface(self, tilenumber: int) -> pygame.Surface:
13         left = (tilenumber % cfg.TILEMAP_NOF_COLS) * cfg.TILESIZE.x #
14         top = (tilenumber // cfg.TILEMAP_NOF_COLS) * cfg.TILESIZE.y #
15         tile_rect = pygame.Rect((left, top), cfg.TILESIZE)
16         return self.image.subsurface(tile_rect)
```

Now we integrate the new class into `WindowGame`. The constructor is extended by determining random tile numbers:

Listing 3.15: Forest – Extension of the Constructor of `WindowGame`

```
    def __init__(self) -> None:
        self.window = pygame.Window(size=cfg.TILEMAP_WINDOW.size)
        self.screen : pygame.Surface = self.window.get_surface()
        self.rect = self.screen.get_frect()
        self.window.title = "Tilemap Example"
        self.clock = pygame.time.Clock()
        self.spritelib = Spritelib("images/forest_tiles.png")
        self.tiles = []
        for row in range(cfg.TILEMAP_NOF_ROWS):
            for col in range(cfg.TILEMAP_NOF_COLS):
                self.tiles.append(choice((0,1,2,3,4,16,17,18,19,20,32,33,34,35,36)))
```

In `WindowGame.draw()`, we now only need to implement access to the tile number. The result then looks as shown in figure 3.7 on the next page.

Listing 3.16: Forest – Selecting specific tile numbers in `Game.draw()`

```
    def draw(self) -> None:
        self.screen.fill("black")
        for row in range(cfg.TILEMAP_NOF_ROWS):
            for col in range(cfg.TILEMAP_NOF_COLS):
                index = self.tiles[row * cfg.TILEMAP_NOF_COLS + col]
                image = self.spritelib.subsurface(index)
                position = col * cfg.TILESIZE.x, row * cfg.TILESIZE.y
                self.screen.blit(image, position)
        self.window.flip()
```

So, what can we do already?

1. We can provide a sprite library to all components of the game via a separate class.
2. We can determine the row – and thus the vertical pixel position in the sprite library – from the tile number.

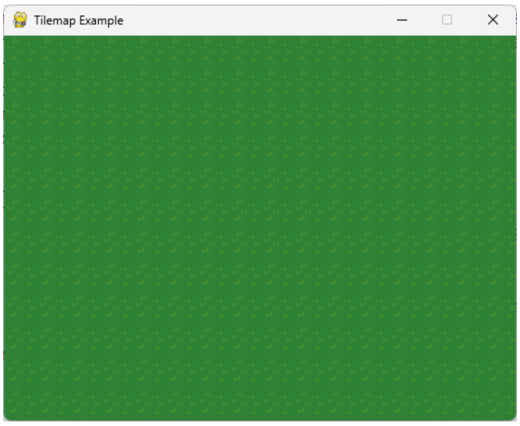

Figure 3.6: Forest playground (1)

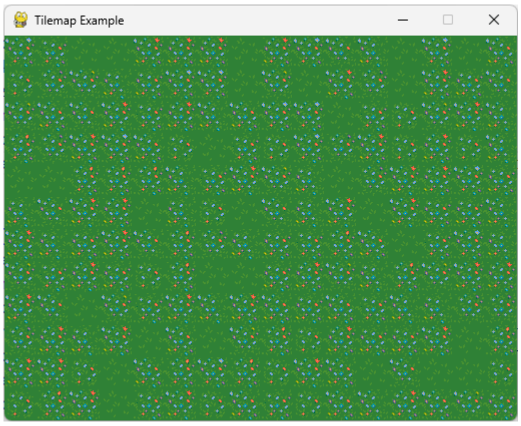

Figure 3.7: Forest playground (2)

3. We can determine the column – and thus the horizontal pixel position in the sprite library – from the tile number.
4. We can compute the upper-left pixel position in the target window – that is, the game playground – from a row and column number.

That is already quite a lot.

3.2.3 Tile Numbers and Two-Dimensional Arrays

In most cases, tiles should not be determined by a random generator. Instead, the tile numbers should be specified explicitly in some way. Two-dimensional arrays are well suited for this purpose, since they allow convenient access using row and column indices.

Nothing could be simpler, you might think. And the reader would be right ;-)

First, let us replace the random selection with a meaningful assignment using a two-dimensional array. The tile numbers are no longer random, but already correspond to those that we want to see later.

Listing 3.17: Forest – Tile numbers in a 2D array

```
class WindowGame:

    def __init__(self) -> None:
        self.window = pygame.Window(size=cfg.TILEMAP_WINDOW.size)
        self.screen : pygame.Surface = self.window.get_surface()
        self.rect = self.screen.get_frect()
        self.window.title = "Tilemap Example"
        self.clock = pygame.time.Clock()
        self.spritelib = Spritelib("images/forest_tiles.png")
        self.tiles = []
        self.tiles.append([99,98,113,17,18,17,18,18,17,18,51,17,17,13,34,34])
        self.tiles.append([115,114,97,60,18,20,34,29,34,34,34,13,34,36,116,117])
        self.tiles.append([99,115,113,17,18,20,116,117,117,117,64,117,120,117,164,133])
        self.tiles.append([115,113,18,18,18,20,132,133,133,136,10,136,133,136,155,155])
```

```
        self.tiles.append([99,97,18,17,104,105,132,136,185,152,71,152,152,149,180,133])
        self.tiles.append([115,113,18,13,17,20,132,136,137,55,56,0,0,53,132,155])
        self.tiles.append([99,97,19,18,18,20,132,136,137,0,0,0,0,0,132,133])
        self.tiles.append([115,113,18,51,18,20,75,10,69,0,50,0,53,0,135,133])
        self.tiles.append([99,97,18,17,17,20,132,136,137,0,0,82,0,0,135,133])
        self.tiles.append([115,113,128,129,18,20,132,136,134,0,0,0,0,0,135,155])
        self.tiles.append([99,97,144,145,13,20,132,136,137,87,103,0,13,0,73,11])
        self.tiles.append([115,113,19,18,19,36,132,133,169,139,139,139,139,139,164,136])
```

The output is now simplified, as can be seen starting at line 45.

Listing 3.18: Forest – `WindowGame.draw()` using a 2D array

```
42        self.screen.fill("black")
43        for row in range(cfg.TILEMAP_NOF_ROWS):
44            for col in range(cfg.TILEMAP_NOF_COLS):
45                index = self.tiles[row][col]        #
46                image = self.spritelib.subsurface(index)
47                position = col * cfg.TILESIZE.x, row * cfg.TILESIZE.y
48                self.screen.blit(image, position)
49        self.window.flip()
```

The result is a bit unsatisfying.

- Since there is only space for a single tile number at each tile position, background tiles are missing. As a result, an ugly black border becomes visible around these tiles.
- The semantic meaning of the tiles is completely lost. This may be acceptable for pure background tiles, but what if some tiles represent ground, others obstacles, and yet others gadgets with a special meaning?

I usually get by with these three semantic layers and encapsulate the whole concept in a `Map` class.

In the constructor, tile numbers are specified only for these three layers. For collision detection, the main program—or wherever it makes sense—can then always check whether a tile number is contained in `layer_data[0]`, `layer_data[1]`, or `layer_data[2]`. The result with regard to the black backgrounds has also improved, as can be seen in figure 3.9 on page 199; since I can now first draw a green meadow as a background in level 0 and later place a tent or a tree in level 1 or 2.

Listing 3.19: Forest – `Map` with a 2D array

```
    def __init__(self) -> None:
        self.layer_data = []
        self.layer_data.append([])
        self.layer_data[0].append([18,17,18,17,18,17,18,18,17,18,18,17,17,19,34,34])
        self.layer_data[0].append([17,18,17,17,18,20,34,34,34,34,34,34,34,36,0,0])
        self.layer_data[0].append([18,1,18,17,18,20,0,0,0,0,64,0,0,0,0,0])
        self.layer_data[0].append([17,17,18,18,18,20,0,0,0,0,10,0,0,0,0,0])
        self.layer_data[0].append([17,18,18,17,18,20,0,0,0,0,71,0,0,0,0,0])
        self.layer_data[0].append([18,17,18,17,17,20,0,0,0,0,0,0,0,0,0,0])
```

```
        self.layer_data[0].append([18,1,19,18,18,20,0,0,0,0,0,0,0,0,0,0])
        self.layer_data[0].append([18,18,18,18,18,20,75,10,69,0,0,0,0,0,0,0])
        self.layer_data[0].append([18,18,18,17,17,20,0,0,0,0,0,0,0,0,0,0])
        self.layer_data[0].append([18,18,17,1,18,20,0,0,0,0,0,0,0,0,0,0])
        self.layer_data[0].append([18,18,18,18,17,20,0,0,0,0,0,0,0,0,73,11])
        self.layer_data[0].append([18,18,19,18,19,36,0,0,0,0,0,0,0,0,0,0])
        self.layer_data.append([])
        self.layer_data[1].append([99,98,113,-1,-1,-1,-1,-1,-1,-1,51,-1,-1,-1,-1,-1,-1])
        self.layer_data[1].append([115,114,97,-1,-1,-1,-1,29,-1,-1,-1,-1,-1,-1,116,117])
        self.layer_data[1].append([99,115,113,-1,-1,-1,116,117,117,117,-1,117,120,117,164,133])
        self.layer_data[1].append([115,113,-1,-1,-1,-1,132,133,133,136,-1,136,133,136,155,155])
        self.layer_data[1].append([99,97,-1,-1,104,105,132,136,185,152,-1,152,152,149,180,133])
        self.layer_data[1].append([115,113,-1,-1,-1,-1,132,136,137,55,56,-1,-1,53,132,155])
        self.layer_data[1].append([99,97,-1,-1,-1,-1,132,136,137,-1,-1,-1,-1,-1,132,133])
        self.layer_data[1].append([115,113,-1,51,-1,-1,-1,-1,-1,-1,50,-1,53,-1,135,133])
        self.layer_data[1].append([99,97,-1,-1,-1,-1,132,136,137,-1,-1,82,-1,-1,135,133])
        self.layer_data[1].append([115,113,128,129,-1,-1,132,136,134,-1,-1,-1,-1,-1,135,155])
        self.layer_data[1].append([99,97,144,145,-1,-1,132,136,137,87,103,-1,-1,-1,-1,-1])
        self.layer_data[1].append([115,113,-1,-1,-1,-1,132,133,169,139,139,139,139,139,164,136])
        self.layer_data.append([])
        self.layer_data[2].append([-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,13,-1,-1])
        self.layer_data[2].append([-1,-1,-1,60,-1,-1,-1,-1,-1,-1,-1,13,-1,-1,-1,-1])
        self.layer_data[2].append([-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1])
        self.layer_data[2].append([-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1])
        self.layer_data[2].append([-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1])
        self.layer_data[2].append([-1,-1,-1,13,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1])
        self.layer_data[2].append([-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1])
        self.layer_data[2].append([-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1])
        self.layer_data[2].append([-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1])
        self.layer_data[2].append([-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1])
        self.layer_data[2].append([-1,-1,-1,-1,13,-1,-1,-1,-1,-1,-1,-1,13,-1,-1,-1])
        self.layer_data[2].append([-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1])

    def get_layer_data(self, layer_index: int) -> list[list[int]]:
        return self.layer_data[layer_index]
```

In `WindowGame`, the object of the class is now created:

Listing 3.20: Forest – `Map` object in `WindowGame`

```
        self.map = Map()
```

In `draw()`, only the `Map` object is accessed from now on. Unassigned tiles (see line 82) are skipped in the process.

Listing 3.21: Forest – `WindowGame.draw()` using a `Map` object

```
76          self.screen.fill("black")
77          for layer_index in range(3):
78              tiles = self.map.get_layer_data(layer_index)
79              for row in range(cfg.TILEMAP_NOF_ROWS):
80                  for col in range(cfg.TILEMAP_NOF_COLS):
81                      index = tiles[row][col]
82                      if index > -1:              #
83                          image = self.spritelib.subsurface(index)
84                          position = col * cfg.TILESIZE.x, row * cfg.TILESIZE.y
85                          self.screen.blit(image, position)
86          self.window.flip()
```

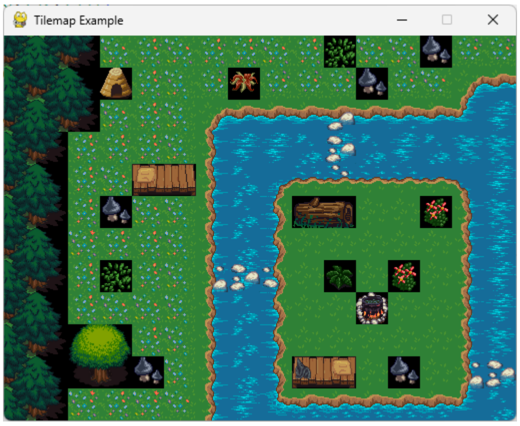

Figure 3.8: Forest playground (3)

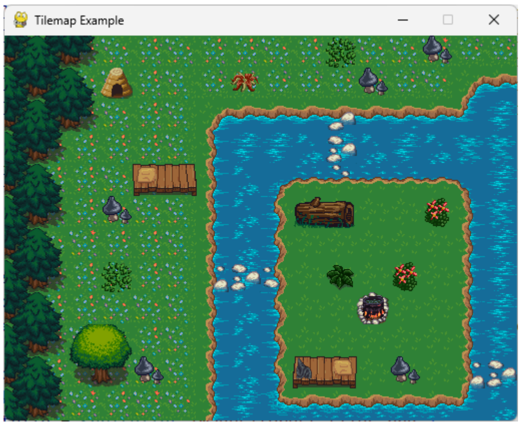

Figure 3.9: Forest playground (4)

Hard-coding the tile numbers directly in the source code is, of course, a nightmare.

It is much better to store them in separate CSV files. The appeal of this approach is that it allows the use of external programs such as *Tiled* for level design. Even self-made tools like `forest00.py` (which can be found in the GitHub repository of this script) are often sufficient and much more efficient than specifying the data directly in the source code.

So let us quickly switch to using CSV files. First, the `csv` module needs to be imported.

Listing 3.22: Forest – `csv` import

```
import csv
```

In the constructor, the CSV files located in the `level` subdirectory are now loaded. Please note that creating the `Map` object now requires specifying how many levels are available as CSV files.

Listing 3.23: Forest – Constructor of `Map` using `csv`

```
    def __init__(self, levels:int) -> None:
        self.level_data = []
        for level in range(levels):
            self.level_data.append([])
            with open(f'levels/level_{level}.csv', 'r') as datei:
                csvReader = csv.reader(datei, delimiter=',')
                for row in csvReader:
                    self.level_data[level].append([int(tile) for tile in row])
```

That should be sufficient at this point :-)

3.3 Very Large Worlds

In many games, the playable universe is too large to be displayed entirely within a single game window or across the whole screen. Therefore, solutions are required to determine how a section of the game world should be displayed relative to the player's position.

3.3.1 A Large Example World

So . . . where are the files?

- https://github.com/adamsralf/pygame_book/tree/main/src/01%20Techniques/03%20WorldScrolling/V01
- https://github.com/adamsralf/pygame_book/tree/main/src/01%20Techniques/03%20WorldScrolling/V02

Let us take a look at a simple example without any actual gameplay mechanics, so that it does not become too distracting. Our world consists of a large number of square tiles that differ only in their color. The closer a tile is to the center, the whiter its yellow color becomes.

The complete source code is split across several files to maintain clarity. Let us start with the file `globals.py`. By now, `FPS` and `DELTATIME` should be self-explanatory. The constant `TILESIZE_WORLD` defines the width and height of a tile in the large world; in our case 24 px. `NOF_COLS` and `NOF_ROWS` specify the number of columns and rows—that is, how many tiles per row and per column exist in the large world. As a result, `WORLD` becomes a rectangle with a width of 2160 px and a height of 1680 px in line 9; this is larger than what can be displayed on most monitors.

The setting `TILE_WITH_BORDER` controls whether the tiles should have an inner border. A value of 0 means *no*, while a value > 0 specifies the width of the inner border. This makes the individual tiles visible again; otherwise, the world would appear as a large color gradient.

Finally, there is `NOF_MOBS`. I let a few blue rectangles move aimlessly around the world so that something is happening visually and the scene appears a bit more representative of a real game; after all, real games usually contain more than just static elements and the player.

Listing 3.24: Big World – `config.py`

```
1 from pygame import FRect, Vector2
2
3 FPS = 60
4 DELTATIME = 1.0 / FPS
```

```
TILESIZE_WORLD = Vector2(24, 24)
TILESIZE_WINDOW = Vector2(6, 6)
NOF_COLS = 90
NOF_ROWS = 70
WORLD: FRect = FRect(0, 0, NOF_COLS * TILESIZE_WORLD.y, NOF_ROWS * TILESIZE_WORLD.y)    #
WINDOW: FRect = FRect(0, 0, NOF_COLS * TILESIZE_WINDOW.x, NOF_ROWS * TILESIZE_WINDOW.y) #
TILE_WITH_BORDER = 0
NOF_MOBS = 50
```

Next, we take a look at the three classes `Tile`, `Player`, and `Mob` in `objects.py`. The `Tile` class is representative of any type of sprite placed in the world. These can be static background elements, walls in the foreground, or other movable objects. For our considerations here, this distinction does not matter.

In the constructor of `Tile`, an image with the size `TILESIZE_WORLD` is created. Starting at line 12, the relative distance of the tile to the center is calculated. This is possible because the position is passed as a parameter. The value range of `rel_dist_center` lies within the interval $[0, 1]$. The blue component for the yellow color is then computed in line 18 and used to color the tile in the following line.

The `Player` class represents a simple wanderer through the world. It is drawn as a simple red circle centered within a tile-sized image. The radius is chosen such that the circle fills the tile as much as possible. In `update()`, either the position of the tile in the world is set directly or a new position is calculated based on the chosen direction. Note that the position refers not to the top-left corner, but to the center of the circle.

Finally, there is the `Mob` class. A blue rectangle is placed at a random position – at a reasonable distance from the borders. The direction vectors are chosen from a uniform distribution, and the speed is also randomly selected between 100 px and 500 px. The size of the rectangle is likewise determined by a random padding. In `update()`, the mob is moved. If it has completely wandered out of the world, it reappears at the opposite edge.

Listing 3.25: Big World – `Tile`, `Player`, and `Mob`

```
1  from random import choice, randint
2  from typing import Any
3
4  import config as cfg
5  import pygame
6
7
8  class Tile(pygame.sprite.Sprite):
9      def __init__(self, position: tuple[float, float]) -> None:
10         super().__init__()
11         self.image = pygame.Surface(cfg.TILESIZE_WORLD)
12         # Yellow -> White according to the distance to the center
13         v1 = pygame.Vector2(position)
14         v2 = pygame.Vector2(cfg.WORLD.center)
15         distance = v2.distance_to(v1)
16         max_distance = v2.length()
17         rel_dist_center = min(1.0, distance / max_distance)
18         blue_value = int(255 * (1 - rel_dist_center))       #
19         color = (255, 255, blue_value)
20         self.image.fill(color)
21         if cfg.TILE_WITH_BORDER > 0:
```

```
            pygame.draw.rect(self.image, "Black",
                             ((0,0), cfg.TILESIZE_WORLD),
                             cfg.TILE_WITH_BORDER)
        self.rect = self.image.get_rect(topleft=position)


class Player(pygame.sprite.Sprite):

    def __init__(self, position: tuple[float, float]) -> None:
        super().__init__()
        self.image : pygame.Surface = pygame.Surface(cfg.TILESIZE_WORLD)
        self.image.set_colorkey((0, 0, 0))

        self.radius = int(cfg.TILESIZE_WORLD.x // 2)
        pygame.draw.circle(self.image, "red", (self.radius, self.radius), self.radius)
        self.rect : pygame.FRect = self.image.get_frect(center=position)

        self.speed = 400.0  # pixels per second
        self.directions = {
            "left": pygame.Vector2(-1, 0),
            "right": pygame.Vector2(1, 0),
            "up": pygame.Vector2(0, -1),
            "down": pygame.Vector2(0, 1),
            "stop": pygame.Vector2(0, 0),
        }
        self.direction = self.directions["stop"]

    def update(self, *args: Any, **kwargs: Any) -> None:
        if "position" in kwargs:
            self.rect.center = kwargs["position"]
        if "move" in kwargs:
            self.direction = self.directions[kwargs["move"]]
        new_position = pygame.Vector2(self.rect.center) + self.direction * (
            self.speed * cfg.DELTATIME
        )
        self.rect.center = new_position
        self.rect.clamp_ip(cfg.WORLD)
        return super().update(*args, **kwargs)


class Mob(pygame.sprite.Sprite):

    def __init__(self) -> None:
        super().__init__()
        x1 = int(cfg.TILESIZE_WORLD.x + 10)
        x2 = int(cfg.WORLD.width - (cfg.TILESIZE_WORLD.x + 10))
        y1 = int(cfg.TILESIZE_WORLD.y + 10)
        y2 = int(cfg.WORLD.height - (cfg.TILESIZE_WORLD.y + 10))
        position = (randint(x1, x2), randint(y1, y2))
        self.image = pygame.Surface(cfg.TILESIZE_WORLD, pygame.SRCALPHA)
        color = (0, 0, randint(10, 255))
        pad = randint(0, int(cfg.TILESIZE_WORLD.x//2 - 1))
        pygame.draw.rect(self.image,
                         color,
                         ((pad,pad), (cfg.TILESIZE_WORLD.x - 2*pad,
                                      cfg.TILESIZE_WORLD.y - 2*pad)))
        self.rect = self.image.get_frect()
        self.rect.topleft = position
        self.direction = pygame.Vector2(choice((-1, 1)), choice((-1,1)))
        self.speed = randint(100, 500) # px/sec

    def update(self, *args: Any, **kwargs: Any) -> None:
        self.rect.move_ip(self.direction * self.speed * cfg.DELTATIME)
        if self.rect.right < cfg.WORLD.left:
            self.rect.left = cfg.WORLD.right
        elif self.rect.left > cfg.WORLD.right:
```

```
            self.rect.right = cfg.WORLD.left
        if self.rect.bottom < cfg.WORLD.top:
            self.rect.top = cfg.WORLD.bottom
        elif self.rect.top > cfg.WORLD.bottom:
            self.rect.bottom = cfg.WORLD.top
        return super().update(*args, **kwargs)
```

Our output window – i. e. the first rather unworthy attempt – is defined in the file `windows.py`. Not much happens here. A window with the size defined in `config.py` is created. Finally, the window title is adjusted so that it conveys a bit of information about its properties. In `draw()`, the window is filled with black, and then all world objects – background tiles as well as moving objects, including the player – are rendered. I added the method `save()` only for this script, so that I can capture images of the current states, for example to show them here.

Listing 3.26: Big World – `WindowPlain`

```
import config as cfg
import pygame


class WindowPlain:

    def __init__(self, tiles:pygame.sprite.Group, mobs:pygame.sprite.Group) -> None:
        self.tiles = tiles
        self.mobs = mobs
        self.window = pygame.Window(size=cfg.WINDOW.size)
        self.window.position = (0 * (cfg.WINDOW.width + 60),
                                0 * (cfg.WINDOW.height) + 30)
        self.screen : pygame.Surface = self.window.get_surface()
        self.rect = self.screen.get_frect()
        self.window.title = f"Plain Window (size={self.rect.size})"
        self.clock = pygame.time.Clock()

    def draw(self):
        self.tiles.draw(self.screen)
        self.mobs.draw(self.screen)
        self.window.flip()

    def save(self):
        pygame.image.save(self.screen, "plain_image.png")
```

In `camera_demo.py`, the class `Game` is now defined and the call to `main()` is performed. In the constructor of `Game`, the functions `create_tiles()` and `create_mobs()` are called. These functions create the tiles and the moving objects and assign them the correct positions within the world; more on this later. In addition, the `WindowPlain` object is created as the output window, as well as the player – i. e. a `Player` object.

In `run()`, the basic structure of the main program loop that I typically use in my games is defined. There are plenty of explanations for this structure earlier and later in the script.

Listing 3.27: Big World – Constructor and `run()` of `Game`

```
class Game:

    def __init__(self) -> None:
        pygame.init()
        self.clock = pygame.time.Clock()
        self.create_tiles()
        self.create_mobs()
        self.window_plain = WindowPlain(self.tiles, self.mobs)
        self.world_image = pygame.Surface(cfg.WORLD.size)
        self.player = Player(cfg.WORLD.center)
        self.mobs.add(self.player)
        self.running = True

    def run(self) -> None:
        time_previous = time()
        while self.running:
            self.watch_for_events()
            if self.running:
                self.update()
                self.draw()
                self.clock.tick(cfg.FPS)
                time_current = time()
                cfg.DELTATIME = time_current - time_previous
                time_previous = time_current
        pygame.quit()
```

The methods shown in source code 3.28 also require no further explanation.

Listing 3.28: Big World – `watch_for_events()`, `update()`, and `draw()` of `Game`

```
    def watch_for_events(self) -> None:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                self.running = False
            elif event.type == pygame.WINDOWCLOSE:
                self.running = False
                event.window.destroy()
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_ESCAPE:
                    self.running = False
                elif event.key == pygame.K_UP:
                    self.player.update(move="up")
                elif event.key == pygame.K_DOWN:
                    self.player.update(move="down")
                elif event.key == pygame.K_LEFT:
                    self.player.update(move="left")
                elif event.key == pygame.K_RIGHT:
                    self.player.update(move="right")
                elif event.key == pygame.K_s:
                    self.save()

            elif event.type == pygame.KEYUP:
                if event.key in (pygame.K_UP, pygame.K_DOWN, pygame.K_LEFT, pygame.K_RIGHT):
                    self.player.update(move="stop")

    def update(self) -> None:
        self.player.update()
        self.mobs.update()

    def draw(self) -> None:
        self.window_plain.draw()
        self.tiles.draw(self.world_image)
```

```
        self.mobs.draw(self.world_image)
```

The methods `create_tiles()` and `create_mobs()` create the game objects – that is, in our case, the static tiles and the moving game elements. The method `save()` triggers saving both the entire world and the game window as PNG files. As mentioned above, this functionality exists only to allow images to be included in this script (for example, figure 3.12 on the next page).

Listing 3.29: Big World – `create_tiles()`, `create_mobs()`, and `save()` of `Game`

```
    def create_tiles(self) -> None:
        self.tiles = pygame.sprite.Group()
        for row in range(cfg.NOF_ROWS):
            for col in range(cfg.NOF_COLS):
                x = col * cfg.TILESIZE_WORLD.x
                y = row * cfg.TILESIZE_WORLD.y
                self.tiles.add(Tile((x, y)))

    def create_mobs(self) -> None:
        self.mobs = pygame.sprite.Group()
        for _ in range(cfg.NOF_MOBS):
            self.mobs.add(Mob())

    def save(self):
        pygame.image.save(self.world_image, "world_image.png")
        self.window_plain.save()
```

If we take a look at the screen outputs of the current source code in figure 3.10 and figure 3.11, the fundamental problem becomes immediately apparent. The window `PlainWindow` is far too small to display the entire world (see figure 3.12 on the following page). We can only see the top-left corner of the large world – once without borders and once with borders. The visible section is so small that even the color gradient is hardly recognizable. Here, the borders help to make the many tiles visible. Later on, we will no longer need them.

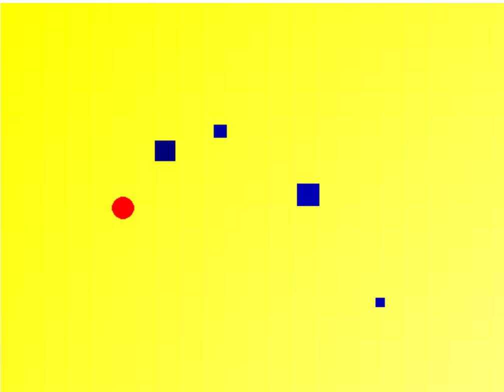

Figure 3.10: Tiles without borders

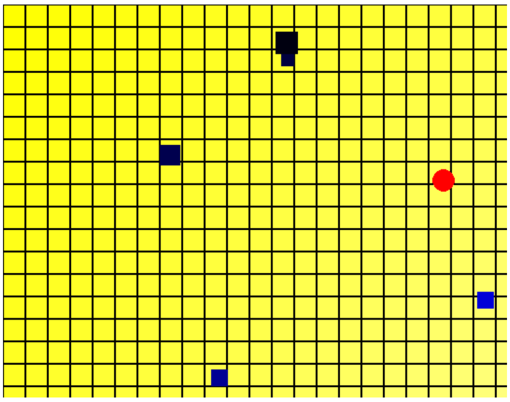

Figure 3.11: Tiles with borders

Visibility culling

Before continuing with the different top-down views, I would like to address a performance issue. To do so, I extend the window title to display the actual `FPS` and increase

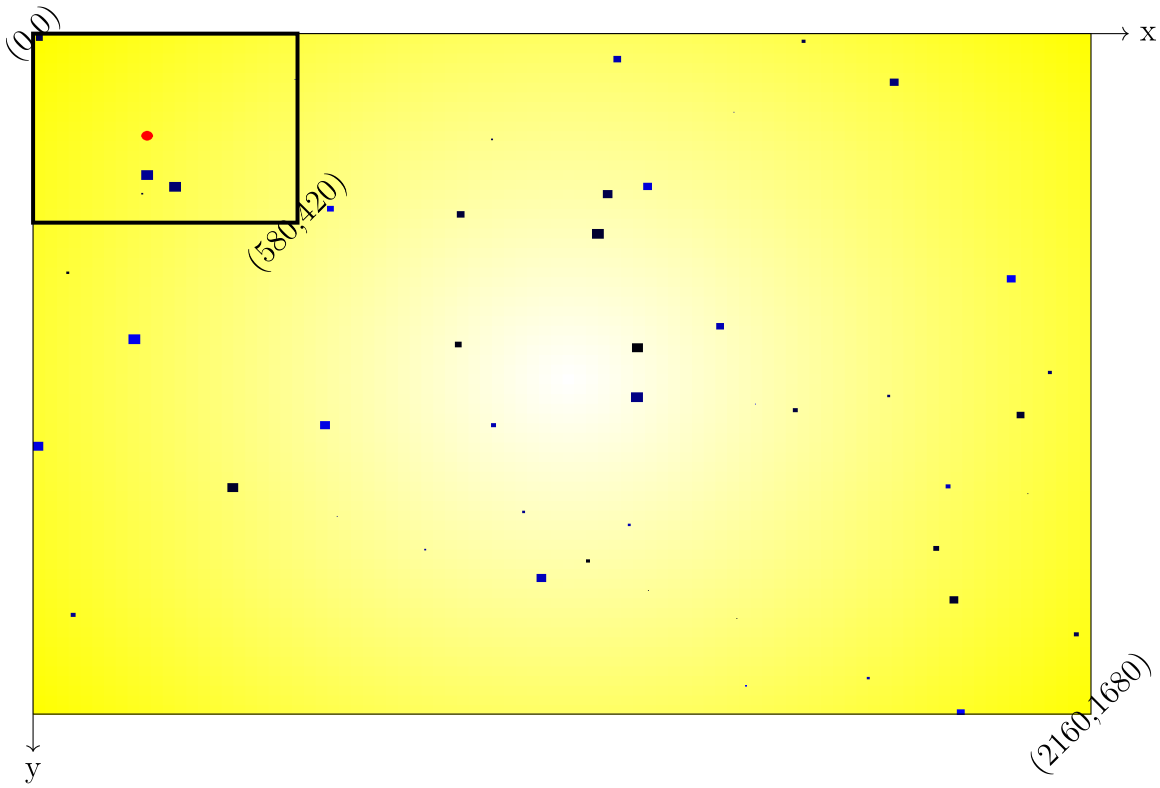

Figure 3.12: PlainWindow as a Viewport of the World

`FPS` in `config.py` to 600 fps – not because this is a realistic value for a game, but because I want to determine how many frames are actually achieved. I then store this actually achieved number of frames in a text file.

If we take a look at the method `draw()` in source code 3.26 on page 203, we see that all tiles and all moving objects are rendered into the target window – that is, drawn – even though the vast majority of these objects are not visible within the window at all.

One possible approach is to first check all sprites, or rather their rectangles, to see whether they are actually located inside the output window. The list `a` is created by iterating over all sprites in the group and, for each one, using `colliderect()` to check whether it lies within or at least touches the rectangle of the window.

Listing 3.30: Big World – `WindowPlain.draw()` with Visibility culling

```
    def draw(self):
        self.screen.fill("Black")
        a = [r for r in self.tiles.sprites() if cfg.WINDOW.colliderect(r.rect)]
        for sprite in a:
            self.screen.blit(sprite.image, sprite.rect)
        #self.tiles.draw(self.screen_plain)
        a = [r for r in self.mobs.sprites() if cfg.WINDOW.colliderect(r.rect)]
        for sprite in a:
            self.screen.blit(sprite.image, sprite.rect)
        #self.mobs.draw(self.screen_plain)
```

```
        self.window.flip()
```

Afterwards, I performed the same performance measurement as before, and the result can be seen in figure 3.13. When visibility checking is enabled, significantly more frames per second are achieved than without it. Conclusion: it is worth using.

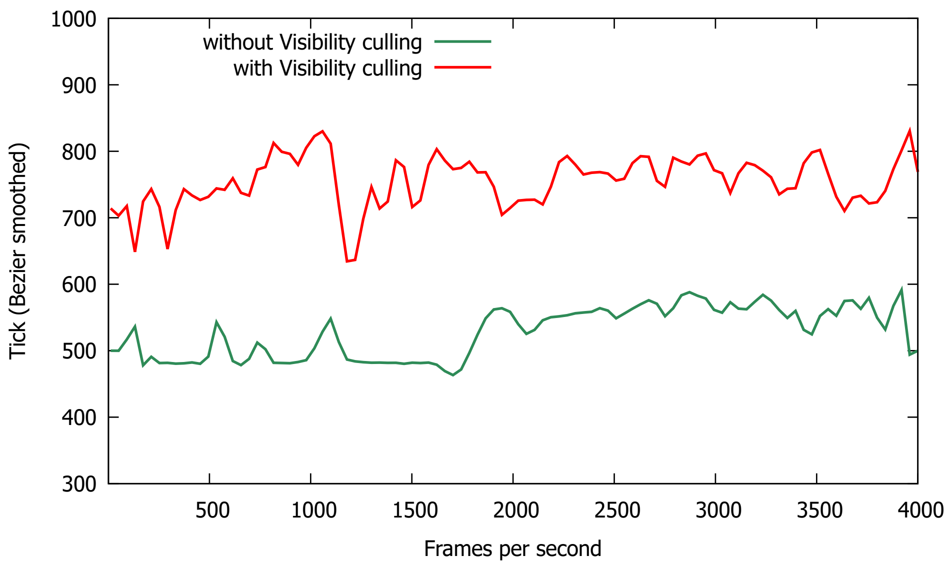

Figure 3.13: Performance without and with visibility culling

3.3.2 Top-Down View / Bird's-Eye View

So … where are the files?

- https://github.com/adamsralf/pygame_book/tree/main/src/01%20Techniques/03%20WorldScrolling/V03
- https://github.com/adamsralf/pygame_book/tree/main/src/01%20Techniques/03%20WorldScrolling/V04

First of all, one might want to have a complete overview of the entire world. This is not strictly necessary. It is not uncommon for a game to never allow the player to see the whole world at once. However, many games offer a top-down view (Bird's-Eye View).

A first naive approach would be to scale down the world using `transform.scale_by()` (see line 50).

This approach has the following advantages and disadvantages

- Advantage: Very easy to implement.
- Disadvantage: In every frame, the entire oversized bitmap has to be created.
- Disadvantage: Scaling can produce undesirable artifacts, since we have no control over which pixels are lost during scaling.
- Disadvantage: Scaling can create objects that are only one pixel in size or even smaller, and therefore hardly visible or not visible at all.

Listing 3.31: Big World – `WindowBirdEyeView`

```
class WindowBirdEyeView:

    def __init__(self, world_screen:pygame.Surface) -> None:
        self.world_screen = world_screen
        zx = cfg.WINDOW.width / cfg.WORLD.width
        zy = cfg.WINDOW.height / cfg.WORLD.height
        self.zoom = pygame.Vector2(zx, zy)
        self.window = pygame.Window(size=cfg.WINDOW.size)
        self.window.position = (1 * (cfg.WINDOW.width + 60),
                                0 * (cfg.WINDOW.height) + 30)
        self.screen : pygame.Surface = self.window.get_surface()
        self.rect = self.screen.get_frect()
        self.window.title = f"Birdeye (zoom=({self.zoom.x:0.2f}, {self.zoom.y:0.2f})"

    def draw(self):
        image = pygame.transform.scale_by(self.world_screen, self.zoom) #
        self.screen.blit(image)
        self.window.flip()

    def save(self):
        pygame.image.save(self.screen, "birdeye_image.png")
```

I therefore recommend that each sprite provides two or possibly even several variants of its image. This makes it easier to identify objects quickly in the Bird's-Eye View as well.

Keep in mind that the purpose of the top-down view is not to allow the game to be played in a full overview after all, but rather to provide orientation, to locate important points of interest, and possibly to identify friends and enemies.

1. Each stationary tile is simply displayed in a scaled-down version.
2. The player is represented by a smaller red circle.
3. The other moving objects are displayed as equally sized blue squares.

One more thing I would like to have is the ability to recognize the visible section of the `PlainWindow` as a rectangle within the Bird's-Eye View.

Let us implement this: First, the Bird's-Eye View window is integrated into the main program (see source code 3.32 and source code 3.33). Please note that at no point in the source code is a surface for the entire world created at its original size anymore.

Listing 3.32: Big World – `BirdEyeView` in the constructor of `Game`

```
        self.window_plain = WindowPlain(self.tiles, self.mobs)
        self.window_birdeye = WindowBirdEyeView(self.tiles, self.mobs)
        self.player = Player(cfg.WORLD.center)
```

In `draw()`, an additional option is prepared, namely the visualization of which part of the entire world is currently covered by `PlainWindow`. To do this, I pass the rectangle of `PlainWindow` and the border color to the `draw()` method of `BirdEyeView` in line 68 (see figure 3.15 on the next page).

Listing 3.33: Big World – `BirdEyeView` in `Game.draw()`

```
64     def draw(self) -> None:
65         self.window_plain.draw()
66         self.window_plain.window.title = f"Plain Window (size={self.window_plain.rect.size},
               fps={self.clock.get_fps():.0f})"
67
68         self.window_birdeye.draw([{"rect":self.window_plain.rect, "color":"blue"}] )#
```

This data is then used in `draw()` of `BirdEyeView`, and the rectangle(s) are drawn starting at line 54.

Listing 3.34: Big World – `BirdEyeView.draw()`

```
49     def draw(self, rects:list):
50         for sprite in self.tiles:
51             self.screen.blit(sprite.image_small, self.zoom_rect(sprite.rect))
52         for sprite in self.mobs:
53             self.screen.blit(sprite.image_small, self.zoom_rect(sprite.rect))
54         if rects:                                               # Are rects to draw?
55             for item in rects:
56                 pygame.draw.rect(self.screen, item["color"], self.zoom_rect(item["rect"]), 2)
57         self.window.flip()
```

One more note on performance: Due to the preparations carried out – namely the one-time creation of a smaller, symbolic representation of the game elements – a significant performance improvement was achieved as well. Analogous to the measurements above, I performed a corresponding benchmark over 660 frames. The result can be read from figure 3.16 on the following page.

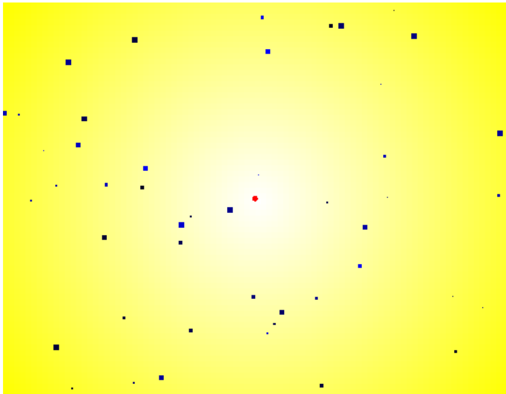

Figure 3.14: BirdEye (scaled)

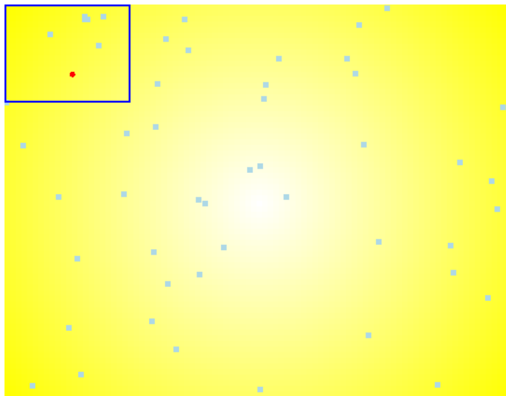

Figure 3.15: Bird's-Eye View (simplified and with visibility indicator)

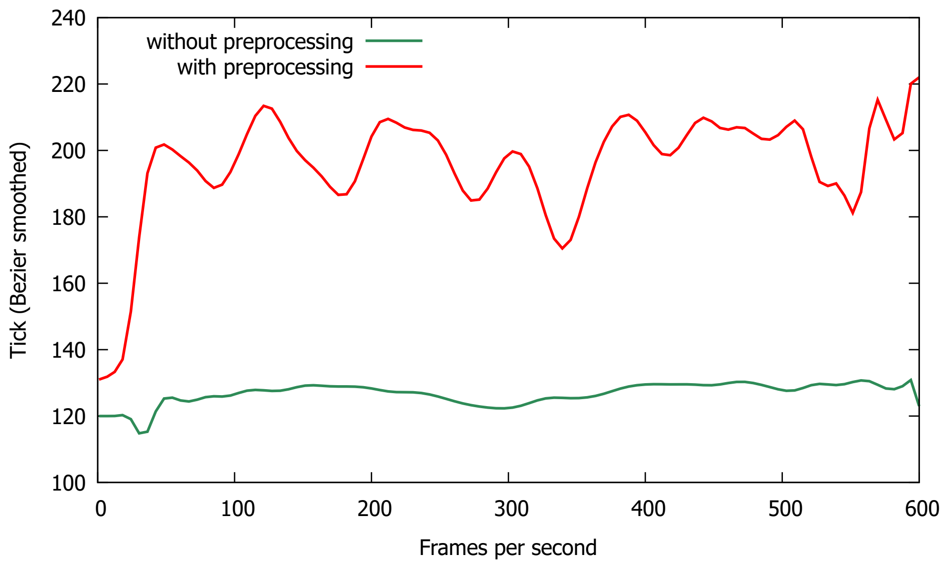

Figure 3.16: Performance without and with preprocessing

3.3.3 Player Centered Camera

So . . . where are the files?

- https://github.com/adamsralf/pygame_book/tree/main/src/01%20Techniques/03%20WorldScrolling/V05
- https://github.com/adamsralf/pygame_book/tree/main/src/01%20Techniques/03%20WorldScrolling/V06
- https://github.com/adamsralf/pygame_book/tree/main/src/01%20Techniques/03%20WorldScrolling/V07

As the next type of view, I would like to introduce the *Player-Centered Camera*. In this view, the player appears to be fixed at the center, while the elements of the game move according to the player's subjective direction of movement.

The basic idea behind a solution is actually quite simple – so do not be intimidated by the mathematics! Each player has a position in the world measured from the top-left corner, given by the vector $\vec{P}_W = (P_{Wx}, P_{Wy})$. The point P_W has a distance to the center of the world—that is, the large game world. Why is this important? Because the player is later supposed to appear at the center of the window.

Therefore, we need to find a correction value (*offset*) that transforms the player's world coordinates into the center coordinates of the camera view window. This offset must be subtracted from the world coordinates, since the coordinates of the view window are much smaller than those of the game world. Offset

Let $\vec{P}_V$ be the player position in the camera view window and $\vec{O}_{ff}$ the correction value. In other words: the position in the world minus the correction value yields the position in the window. Let us now transform this relationship so that we can compute the correction value.

$$
\begin{aligned}
\vec{P}_W - \vec{O}_{ff} &= \vec{P}_V && \| + \vec{O}_{ff} && (3.1)\\
\vec{P}_W &= \vec{P}_V + \vec{O}_{ff} && \| - \vec{P}_V\\
\vec{P}_W - \vec{P}_V &= \vec{O}_{ff} && \| reverseorder\\
\vec{O}_{ff} &= \vec{P}_W - \vec{P}_V && \| coordinatenotation\\
\begin{pmatrix} O_{ffx} \\ O_{ffy} \end{pmatrix} &= \begin{pmatrix} P_{Wx} \\ P_{Wy} \end{pmatrix} - \begin{pmatrix} P_{Vx} \\ P_{Vy} \end{pmatrix} && \| vectorsubtraction\\
\begin{pmatrix} O_{ffx} \\ O_{ffy} \end{pmatrix} &= \begin{pmatrix} P_{Wx} - P_{Vx} \\ P_{Wy} - P_{Vy} \end{pmatrix} && && (3.2)
\end{aligned}
$$

Equation 3.3 reflects the fact that the new position of our player is supposed to be exactly the center of the window, that is, half the width and half the height:

$$\begin{pmatrix} P_{Vx} \\ P_{Vy} \end{pmatrix} = \begin{pmatrix} S_{Vx}/2 \\ S_{Vy}/2 \end{pmatrix} \tag{3.3}$$

Now we substitute equation 3.3 into equation 3.2:

$$\begin{pmatrix} O_{ffx} \\ O_{ffy} \end{pmatrix} = \begin{pmatrix} P_{Wx} - S_{Vx}/2 \\ P_{Wy} - S_{Vy}/2 \end{pmatrix} \tag{3.4}$$

With this, we have indeed derived – using nothing more than straightforward mathematics – the formula for computing the translation, that is, the offset.

It is time for a bit of source code. Let us first prepare everything in the main program. Even though we have not implemented the class yet, we can simply treat the new window like the other two and work with copy/paste. In line 20, the window is created. Please note that this has to happen after creating the `Player`, because we need its position.

Listing 3.35: Big World – Centered Camera in `Game`

```
11      def __init__(self) -> None:
12          pygame.init()
13          self.clock = pygame.time.Clock()
14          self.create_tiles()
15          self.create_mobs()
16          self.window_plain = WindowPlain(self.tiles, self.mobs)
17          self.window_birdeye = WindowBirdEyeView(self.tiles, self.mobs)
18          self.player = Player(cfg.WORLD.center)
19          self.mobs.add(self.player)
20          self.window_centered = WindowCenteredCamera(self.player, self.tiles, self.mobs) #
21          self.running = True
```

In the `draw()` method, three additions are necessary. In line 74, the rectangle for the Bird's-Eye View is added, showing the visible section of the new window. After that, `draw()` is called as with the other windows and the title line is updated.

Listing 3.36: Big World – Centered Camera in `Game.draw()`

```
66      def draw(self) -> None:
67          self.window_plain.draw()
68          self.window_plain.window.title = f"Plain Window (size={self.window_plain.rect.size},
                fps={self.clock.get_fps():.0f})"
69
70          self.window_centered.draw()                                                      #
71          self.window_centered.window.title = f"Centered Window
                (offset={self.window_centered.offset})"
72
73          self.window_birdeye.draw([{"rect":self.window_plain.rect, "color":"blue"},
74                                    {"rect":self.window_centered.rect, "color":"green"}]) #
```

And `save()` also needs to be extended by line 92:

Listing 3.37: Big World – Centered Camera in `Game.save()`

```
    def save(self):
        self.window_plain.save()
        self.window_birdeye.save()
        self.window_centered.save()                                    #
```

Now let us move on to the fun part: the `WindowCenteredCamera` class. First, the self-explanatory `__init__()`. In addition, two attributes are defined here: `self.offset` and `self.player`. Using the offset, I will later compute the new coordinates, and the player is used to compute the offset.

Listing 3.38: Big World – Constructor of `WindowCenteredCamera`

```
    def __init__(self, player:pygame.sprite.Sprite, tiles:pygame.sprite.Group,
        mobs:pygame.sprite.Group) -> None:
        self.tiles = tiles
        self.mobs = mobs
        self.player = player
        self.offset = pygame.Vector2(0, 0) #
        self.window = pygame.Window(size=cfg.WINDOW.size)
        self.window.position = (2 * (cfg.WINDOW.width + 60),
                                0 * (cfg.WINDOW.height) + 30)
        self.screen : pygame.Surface = self.window.get_surface()
        self.rect = self.screen.get_frect()
        self.clock = pygame.time.Clock()
```

The offset is computed in the method `scroll()`. Keep equation 3.4 on the preceding page next to the source code. The implementation should be self-explanatory, as it serves as an example of how easily mathematical expressions can be translated into source code.

Why is the last line actually needed? It is not required for scrolling itself. However, by providing this value, I can inspect the rectangle of the visible world section in the Bird's-Eye View.

Listing 3.39: Big World – `WindowCenteredCamera.scroll()`

```
    def scroll(self) -> None:
        self.offset.x = self.player.rect.x - self.rect.width / 2
        self.offset.y = self.player.rect.y - self.rect.height / 2
        self.rect.topleft = self.offset
```

Now we have everything in place to implement the remaining parts. Let us start with two helper methods so that the coordinate transformations do not have to be implemented multiple times. In `world2camera()`, the coordinates of the game objects in the large world are transformed into the coordinates of the centered view. This calculation corresponds exactly to the initial idea shown in equation 3.1 on page 211.

Listing 3.40: Big World – `WindowCenteredCamera.world2camera()`

```
    def world2camera(self, rect: pygame.FRect) -> pygame.FRect:
        return pygame.FRect(rect.topleft - self.offset, rect.size)
```

What remains is the method `draw()`. This method looks almost the same as the `draw()` method of the other class. However, here the coordinates of the game objects are transformed using `camera2world()` before the visibility check. Take a moment to think about why this is necessary!

Listing 3.41: Big World – `WindowCenteredCamera.draw()`

```python
    def draw(self) -> None:
        self.screen.fill("lightgrey")
        a = [r for r in self.tiles.sprites() if
             cfg.WINDOW.colliderect(self.world2camera(r.rect))]
        for sprite in a:
            self.screen.blit(sprite.image, self.world2camera(sprite.rect))
        a = [r for r in self.mobs.sprites() if
             cfg.WINDOW.colliderect(self.world2camera(r.rect))]
        for sprite in a:
            self.screen.blit(sprite.image, self.world2camera(sprite.rect))
        self.window.flip()
```

If we now run the source code and move the player to the top-left corner of the game world, we obtain the views shown in figure 3.17 and figure 3.18. In the left image, the screen section visible in the right image can be identified by the green rectangle. The green rectangle only appears to be smaller than the blue one; in fact, three quarters of the view lie outside the visible area of the Bird's-Eye View.

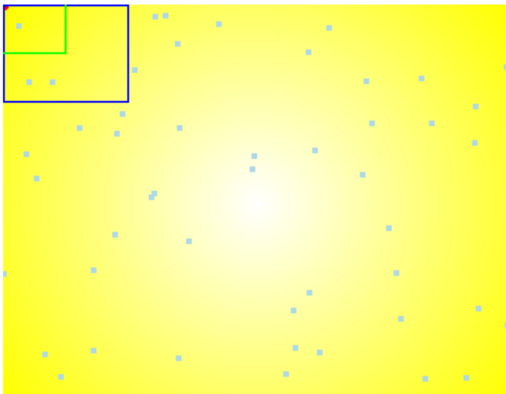

Figure 3.17: Bird's-Eye View: Green = Centered

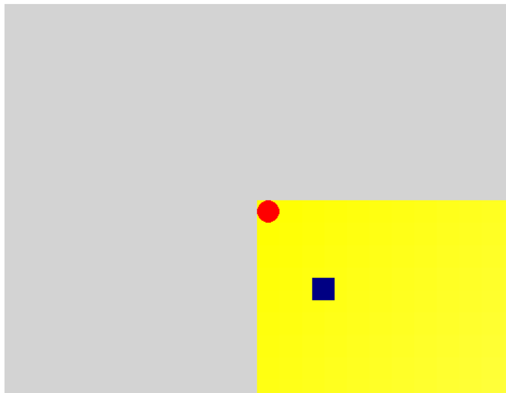

Figure 3.18: Centered Camera – with border error

We therefore need to adjust the method `scroll()` so that the borders are not exceeded. We now see that the offset is limited to 0 at the top and on the left, meaning it cannot become negative. This would otherwise indicate that we are extending beyond the world to the top or to the left. The same logic applies to the right and bottom edges. Here, it is checked whether the right edge of the object exceeds the right edge of the world, and analogously whether this also happens at the bottom. This procedure is known as clamp.

Clamp

Listing 3.42: Big World – `WindowCenteredCamera.scroll()` with clamping

```
    def scroll(self) -> None:
        # Clamp left/top so we do not scroll past the world edges
        self.offset.x = max(0, self.player.rect.x - self.rect.width / 2)
        self.offset.y = max(0, self.player.rect.y - self.rect.height / 2)

        # Clamp right/bottom so we do not scroll past the world edges
        self.offset.x = min(cfg.WORLD.right - self.rect.width, self.offset.x)
        self.offset.y = min(cfg.WORLD.bottom - self.rect.height, self.offset.y)

        self.rect.topleft = self.offset
```

In figure 3.20, we no longer see any border artifacts. Instead, the player's position has shifted from the center toward the edge—exactly as intended. Note: In figure 3.19, the blue border of `PlainWindow` can no longer be seen, since both views now cover the same section.

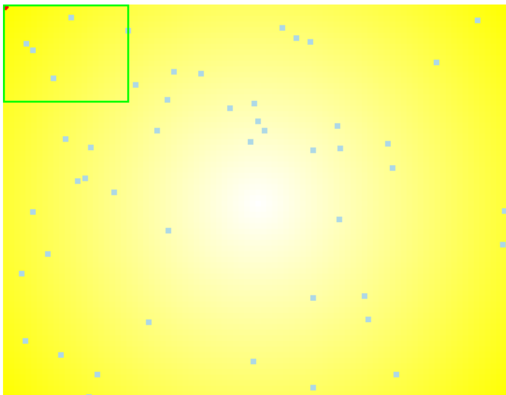

Figure 3.19: BirdEyeView: Grün=Centered

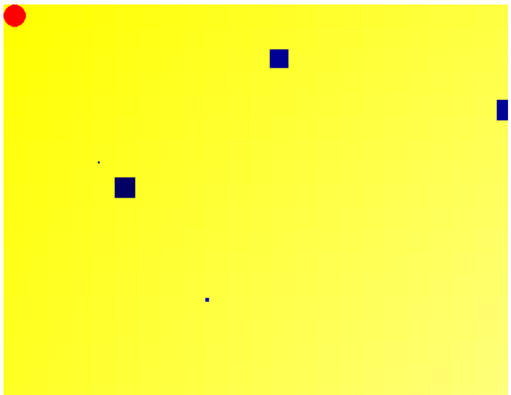

Figure 3.20: Centered Camera – without border error

There is one more thing I would like to address

In the method `draw()` (see source code 3.41 on the preceding page), the method `world2camera()` is called for every single game object; that is, thousands of coordinates are transformed. Would it not be more efficient to transform the world coordinates just once and then compare them with the coordinates of the game objects?

To this end, we introduce a new method, appropriately called `camera2world()`. It is, so to speak, the inverse of `world2camera()`.

Listing 3.43: Big World – `WindowCenteredCamera.camera2world()` with clamping

```
    def world2camera(self, rect: pygame.FRect) -> pygame.FRect:
        return pygame.FRect(rect.topleft - self.offset, rect.size)
```

I now integrate this into `draw()`. I have left the old computation commented out above, so that the difference can be seen more clearly. Once again, simple reasoning has led to a performance gain (see figure 3.21).

Listing 3.44: Big World – `WindowCenteredCamera.draw()` with clamping

```
    def draw(self):
        self.screen.fill("Black")
        #a = [r for r in self.tiles.sprites() if
            Settings.WINDOW.colliderect(self.world2camera(r.rect))]
        w = self.camera2world(cfg.WINDOW)
        a = [r for r in self.tiles.sprites() if w.colliderect(r.rect)]
        for sprite in a:
            self.screen.blit(sprite.image, self.world2camera(sprite.rect))
        #a = [r for r in self.mobs.sprites() if
            Settings.WINDOW.colliderect(self.world2camera(r.rect))]
        a = [r for r in self.mobs.sprites() if w.colliderect(r.rect)]
        for sprite in a:
            self.screen.blit(sprite.image, self.world2camera(sprite.rect))
        self.window.flip()
```

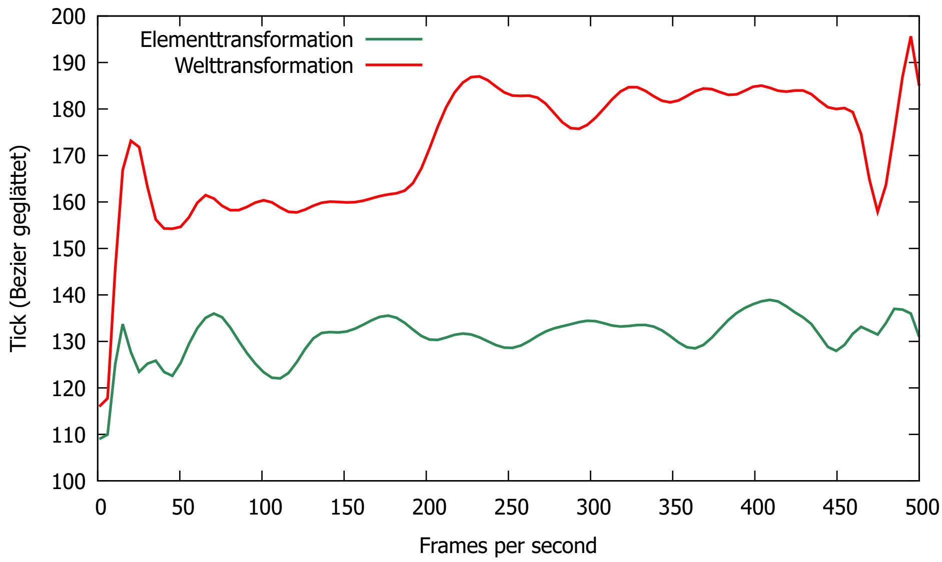

Figure 3.21: Performance with element-based and world-based transformation

3.3.4 Page Scrolling/Edge Scrolling

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/01%20Techniques/03%20WorldScrolling/V08

Keeping the player permanently centered can lead to a restless and confusing visual effect, depending on the game's visual style. It is also associated with many transformations that are often unnecessary for the gameplay itself. Page-wise scrolling (edge scrolling or page scrolling) represents a good compromise. The player initially moves normally within the visible game area. Only when a minimum distance (padding) to one of the borders is undershot does the background shift in the corresponding direction – that is, the view scrolls to the next page.

Edge

Page

Let us take a look at the constructor of `WindowPagewise`. Lost of the elements have already been explained above. What is new is the attribute `inner_rect` and the parameter `padding`. This parameter controls the distance between the inner rectangle and the boundaries of the view—that is, the space between the inner brown rectangle and the outer green rectangle in figure 3.22 on the next page. The parameter is – purely arbitrarily – an integer here and serves as a factor for calculating the distance. As a second factor, I chose the width and height of the player. Semantically, this means that a value of 2 results in a padding of two player widths or heights.

Listing 3.45: Big World – Constructor of `WindowPagewise`

```
class WindowPagewise:

    def __init__(self, player:pygame.sprite.Sprite, tiles:pygame.sprite.Group,
        mobs:pygame.sprite.Group, padding:int = 1) -> None:
        self.tiles = tiles
        self.mobs = mobs
        self.player = player
        self.offset = pygame.Vector2(0, 0)
        self.window = pygame.Window(size=cfg.WINDOW.size)
        self.window.position = (0 * (cfg.WINDOW.width + 60),
                                1 * (cfg.WINDOW.height) + 30)
        self.screen : pygame.Surface = self.window.get_surface()
        self.rect = self.screen.get_frect()
        self.clock = pygame.time.Clock()
        self.inner_rect = pygame.FRect(
            cfg.WINDOW.left + padding * player.rect.width,
            cfg.WINDOW.top + padding * player.rect.height,
            cfg.WINDOW.width - padding * 2 * player.rect.width,
            cfg.WINDOW.height - padding * 2 * player.rect.height,
        )
```

The actual work happens in the method `scroll()`. Here as well, the basic logic is fairly simple. If the player's rectangle lies within the inner rectangle, no scrolling needs to take place at all; the player simply moves normally. Once the player leaves the inner area, scrolling has to be performed. How do we test whether the player is still inside the inner rectangle? By checking whether the player no longer collides with the inner rectangle (line 148).

Listing 3.46: Big World – `WindowPagewise.scroll()`

```
146     def scroll(self) -> None:
147         player_in_view = self.world2camera(self.player.rect)
148         if not player_in_view.colliderect(self.inner_rect):     # nicht mehr innerhalb?
149             self.offset.x = max(0, self.player.rect.x - cfg.WINDOW.centerx)
150             self.offset.y = max(0, self.player.rect.y - cfg.WINDOW.centery)
151             self.offset.x = min(cfg.WORLD.right - cfg.WINDOW.width, self.offset.x)
152             self.offset.y = min(cfg.WORLD.bottom - cfg.WINDOW.height, self.offset.y)
153         self.rect.topleft = self.offset
```

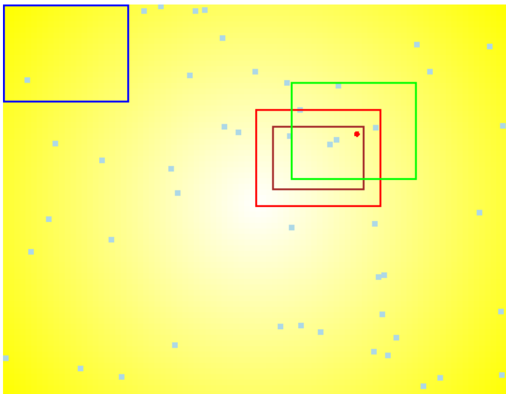

Figure 3.22: Bird's-Eye: Centered, Pagewise, InnerRect

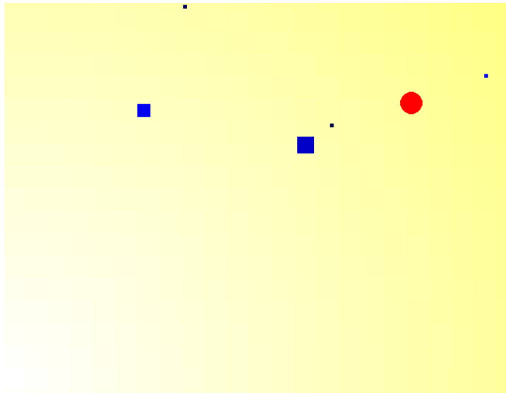

Figure 3.23: Page / Edge Scrolling

Integrating the new class is done in exactly the same way as integrating `WindowCenteredCamera`. Only the visualization of the rectangles in the Bird's-Eye View requires some explanation here: In line 82, the rectangle of the view is rendered in red. The second rectangle, shown in brown, is the inner rectangle whose boundary triggers scrolling when it is crossed. The coordinates for these rectangles are computed in advance in line 80.

Listing 3.47: Big World – `Game.draw()`

```
68
69      def draw(self) -> None:
70          self.window_plain.draw()
71          self.window_plain.window.title = f"Plain Window (size={self.window_plain.rect.size},
                fps={self.clock.get_fps():.0f})"
72
73          self.window_pagewise.draw()                                                    #
74          xy = int(self.window_pagewise.offset.x), int(self.window_pagewise.offset.y)
75          self.window_pagewise.window.title = f"Pagewise Window (offset={xy})"
76
77          self.window_centered.draw()
78          self.window_centered.window.title = f"Centered Window
                (offset={self.window_centered.offset})"
79
80          inner = self.window_pagewise.camera2world(self.window_pagewise.inner_rect)     #
81          self.window_birdeye.draw([{"rect":self.window_plain.rect, "color":"blue"},
```

```
                                      {"rect":self.window_pagewise.rect, "color":"red"},     #
                                      {"rect":inner, "color":"brown"},                       #
                                      {"rect":self.window_centered.rect, "color":"green"}] )
```

3.3.5 Auto Scrolling/Endless Scrolling

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/01%20Techniques/03%20WorldScrolling/V09

Another variant is auto scrolling. In this approach, the background moves automatically and continuously in a fixed direction, while the player usually remains centered and can only evade vertically—for example by jumping. It is also quite common for the background to move downward, requiring the player to jump to higher platforms.

Let us therefore take a look at the new class `WindowAuto`. What is new or special here is that the constructor receives a direction parameter. This consists of two numbers: one for horizontal movement and one for vertical movement. The values represent the speed in px/s, while the sign determines the direction: positive values indicate movement to the right and downward, negative values movement to the left and upward.

Listing 3.48: Big World – Constructor of `WindowAuto`

```

class WindowAuto:

    def __init__(self, player:pygame.sprite.Sprite, tiles:pygame.sprite.Group,
        mobs:pygame.sprite.Group, direction:Vec2Like) -> None:
        self.tiles = tiles
        self.mobs = mobs
        self.player = player
        self.offset = pygame.Vector2(0, 0) #
        self.window = pygame.Window(size=cfg.WINDOW.size)
        self.window.position = (1 * (cfg.WINDOW.width + 60),
                                1 * (cfg.WINDOW.height) + 30)
        self.screen : pygame.Surface = self.window.get_surface()
        self.rect = self.screen.get_frect()
        self.direction = pygame.math.Vector2(direction)
```

Here as well, essentially only the method `scroll()` needs to be adapted. Only the first line is of interest. According to the task, the offset is adjusted using the specified `direction`. Multiplying by `DELTATIME` allows the unit px/s to be used instead of px/frame.

Listing 3.49: Big World – `WindowAuto.scroll()`

```
    def scroll(self) -> None:
        self.offset += self.direction * cfg.DELTATIME
```

```
        self.offset.x = max(0, self.offset.x)
        self.offset.y = max(0, self.offset.y)
        self.offset.x = min(cfg.WORLD.right - cfg.WINDOW.width, self.offset.x)
        self.offset.y = min(cfg.WORLD.bottom - cfg.WINDOW.height, self.offset.y)
```

For the sake of completeness: do not forget to adjust the name in `save()` ;-)

3.3.6 As a Strategy Pattern

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/01%20Techniques/03%20WorldScrolling/Pattern

Although I hope that all the techniques presented so far have been explained clearly, it is worthwhile to further decouple the algorithms from the concrete game scenario of my *large world*. After all, these techniques occur repeatedly, making it worthwhile to formulate a reusable solution.

If we compare the three scenarios closely, we find that they differ only in the method `scroll()`, with one or two additional attributes being required in each case. In other words, we can encapsulate the algorithms in separate classes and then formulate a solution using the strategy pattern. The resulting architecture is shown in figure 3.24.

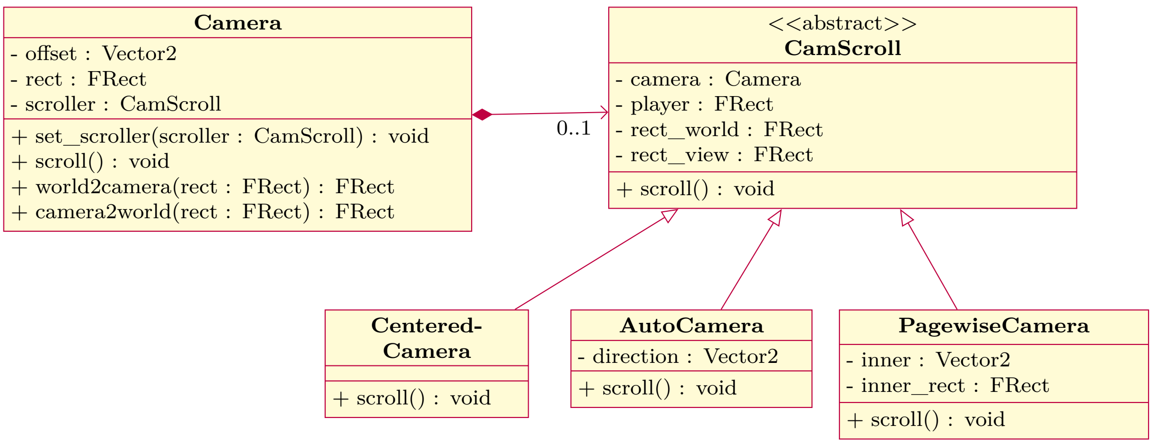

Figure 3.24: Strategy pattern applied on `Camera` and scroll strategies in `cameraview.py`

The class `Camera` is the class that we will later use in our game. It contains all attributes and methods required to render an oversized world. It also contains a *placeholder* for the actual scrolling behavior: the attribute `scroller`. In the method `scroll()`, the scrolling behavior of the behavior class is then invoked.

`CamScroll(ABC)` is the abstract interface class for the behavior classes and essentially consists only of the abstract method `scroll()`. The four attributes are required for computing the scrolling behavior.

The three concrete behavior classes are `CenteredCamera`, `AutoCamera`, and `PagewiseCamera`. Here, the method `scroll()` is implemented. The logic of these implementations corresponds exactly to the approaches shown above and should therefore be easy to understand.

A video demonstration can be found here: https://youtu.be/A2uXPimynnc.

Further resources include:

- https://www.youtube.com/watch?v=XmSv2V69Y7A
- https://www.youtube.com/watch?v=ARt6DLP38-Y
- https://www.youtube.com/watch?v=FDJU8lIObVE

4 Examples

4.1 Pong

The ultimate beginner classic. This game has been played in countless variations since 1972. Because the rules are simple and easy to understand, it is perfectly suited as a first programming project.

We will develop this game step by step in a systematic way, assuming that the techniques from chapter 2 are already familiar. I will deliberately omit docstring comments in the source code, since everything is explained in the text and including them here would only make the listings unnecessarily long. They are, of course, included in the final version.

Note: At the very beginning, I once asked ChatGPT to generate a Pong game for me. It was quite impressive to see that it produced a fully working game.

4.1.1 Requirement 1: Standards

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/01%20Pong/v01

Requirement 1 Standard functionality

1. *The window has an appropriate size.*
2. *The background is a dark red playing field with a dashed center line.*
3. *The game can be exited using the* Esc *key or by clicking the red "X".*
4. *The game runs at a speed independent of the* FPS.

And off we go. Here, the `config.py`. I assume that you have sufficient Python knowledge to extend it as needed.

Listing 4.1: Pong (Requirement 1) – `config.py`

```
from pygame import Rect

WINDOW = Rect(0, 0, 1000, 600)
FPS = 60
DELTATIME = 1.0 / FPS
```

The background is not loaded from a bitmap this time, but created on the fly. There is no deep reason for this – apart from showing that bitmaps do not always have to come from image files (see section 2.3.2.3 on page 54). Instead, they can be generated dynamically as well.

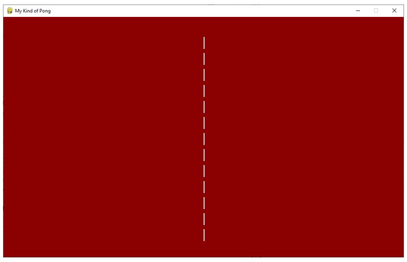

Fig. 4.1: Pong: the background

To do this, a `Surface` object with the size of the screen is created first. It is then filled with a dark red color, meant to resemble a clay court. In `paint_net()`, starting at line 21, the net is drawn as a sequence of small white rectangles.

Listing 4.2: Pong (Requirement 1) – the class `Background`

```
8  class Background(pygame.sprite.Sprite):
9      def __init__(self, *groups: Tuple[pygame.sprite.Group]) -> None:
10         super().__init__(*groups)
11         self.image = pygame.Surface(cfg.WINDOW.size).convert()
12         self.rect = self.image.get_rect()
13         self.image.fill("darkred")
14         self.paint_net()
15 
16     def paint_net(self) -> None:
17         net_rect = pygame.Rect(0, 0, 0, 0)
18         net_rect.centerx = cfg.WINDOW.centerx
19         net_rect.top = 50
20         net_rect.size = (3, 30)
21         while net_rect.bottom < cfg.WINDOW.bottom:  # Net as a seq of rectangles
22             pygame.draw.rect(self.image, "grey", net_rect, 0)
23             net_rect.move_ip(0, 40)
```

The class `Game` consists of the basic building blocks that we have already seen in chapter 2. In `__init__()`, Pygame is initialized, the window and the clock are created, and the control flag for the main game loop is set up. The background is stored in a `GroupSingle` object. The remaining methods should be fairly self-explanatory.

Listing 4.3: Pong (Requirement 1) – the class `Game`

```
class Game:
    def __init__(self):
        pygame.init()
        self.window = pygame.Window(size=cfg.WINDOW.size, title="My Kind of Pong",
            position=pygame.WINDOWPOS_CENTERED)
        self.screen = self.window.get_surface()
        self.clock = pygame.time.Clock()
        self.background = pygame.sprite.GroupSingle(Background())
        self.running = True

    def run(self):
        time_previous = time()
        while self.running:
            self.watch_for_events()
            self.update()
            self.draw()
            self.clock.tick(cfg.FPS)
            time_current = time()
```

```
            cfg.DELTATIME = time_current - time_previous
            time_previous = time_current
        pygame.quit()

    def update(self):
        pass

    def draw(self):
        self.background.draw(self.screen)
        self.window.flip()

    def watch_for_events(self):
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                self.running = False
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_ESCAPE:
                    self.running = False
```

At this point, the application is not functional yet, but it already displays the background as you can see in figure 4.1 on the facing page.

4.1.2 Requirement 2: The Paddles

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/01%20Pong/v02

Requirement 2 Paddles

1. *There is one rectangular paddle on the left side and one on the right side.*
2. *The paddles have a width of 15 px and a height of one tenth of the screen height.*
3. *The paddles have a speed of* $\frac{screen\ height}{2}$ px/s.
4. *Each paddle is positioned at a distance of 50 px from the left or right edge, measured from its center.*
5. *The left paddle is moved upward using* `W` *and downward using* `S`.
6. *The right paddle is moved upward using* `↑` *and downward using* `↓`.
7. *The paddles cannot leave the playing field.*

In line 32, the size of the paddle is calculated (requirements 2.1 and 2.2). Starting at line 33, the paddles are positioned. Vertically, they always start in the center of the screen. Horizontally, their start position depends on whether we are dealing with the left or the right paddle. In both cases, they are placed slightly away from the edge, exactly as specified in requirement 2.4.

The paddle speed is set in line 39 according to requirement 2.3. Just like the background, this bitmap is not loaded from a file but created directly in the code (line 41) and filled with a bright yellow color.

Listing 4.4: Pong (Requirement 2) – The constructor of `Paddle`

```
26 class Paddle(pygame.sprite.Sprite):
27     BORDERDISTANCE = {"horizontal": 50, "vertical": 10}
28     DIRECTION = {"up": -1, "down": 1, "halt": 0}
29 
30     def __init__(self, player: str, *groups: Tuple[pygame.sprite.Group]) -> None:
31         super().__init__(*groups)
32         self.rect = pygame.FRect(0, 0, 15, cfg.WINDOW.height // 10)# Size
33         self.rect.centery = cfg.WINDOW.centery                         # Position
34         self.player = player
35         if self.player == "left":
36             self.rect.left = Paddle.BORDERDISTANCE["horizontal"]
37         else:
38             self.rect.right = cfg.WINDOW.right - Paddle.BORDERDISTANCE["horizontal"]
39         self.speed = cfg.WINDOW.height // 2                            # Speed
40         self.direction = Paddle.DIRECTION["halt"]   #
41         self.image = pygame.Surface(self.rect.size)            # Surface
42         self.image.fill("yellow")
```

The method `update()` is responsible for distributing the tasks. With regard to movement, the attribute `self.direction` is adjusted accordingly (starting at line 48). If the paddle is supposed to change its position, the method `move()` is called in line 46.

Listing 4.5: Pong (Requirement 2) – `Paddle.update()`

```
44     def update(self, *args: Any, **kwargs: Any) -> None:
45         if "action" in kwargs.keys():
46             if kwargs["action"] == "move":                             # Change Postion
47                 self.move()
48             elif kwargs["action"] in Paddle.DIRECTION.keys():          # Direction
49                 self.direction = Paddle.DIRECTION[kwargs["action"]]
50         return super().update(*args, **kwargs)
```

All that remains is the method `move()`. It looks more complicated than it actually is. After checking whether there is anything to do at all, the new vertical position is calculated in line 54 (the horizontal position remains unchanged). After that, it is checked whether the paddle has left the playing field. If so, the paddle is moved back to the top or bottom edge accordingly.

Listing 4.6: Pong (Requirement 2) – `Paddle.move()`

```
52     def move(self) -> None:
53         if self.direction != Paddle.DIRECTION["halt"]:
54             self.rect.move_ip(0, self.speed * self.direction * cfg.DELTATIME)  #
55             if self.direction == Paddle.DIRECTION["up"]:
56                 self.rect.top = max(self.rect.top, Paddle.BORDERDISTANCE["vertical"])
57             elif self.direction == Paddle.DIRECTION["down"]:
58                 self.rect.bottom = min(self.rect.bottom, cfg.WINDOW.height -
                       Paddle.BORDERDISTANCE["vertical"])
```

Now the paddles need to be integrated into the `Game` class. In line 69, a sprite group is created first, which will hold all sprites except the background. After that, the two paddles are created and immediately added to the sprite group via constructor arguments.

Listing 4.7: Pong (Requirement 2) – Constructor of `Game`

```
61 class Game:
62     def __init__(self):
63         pygame.init()
64         self.window = pygame.Window(size=cfg.WINDOW.size, title="My Kind of Pong",
               position=pygame.WINDOWPOS_CENTERED)
65         self.screen = self.window.get_surface()
66         self.clock = pygame.time.Clock()
67         self.background = pygame.sprite.GroupSingle(Background())
68         self.all_sprites = pygame.sprite.Group()
69         self.paddle = {}  # Schläger
70         self.paddle["left"] = Paddle("left", self.all_sprites)
71         self.paddle["right"] = Paddle("right", self.all_sprites)
72         self.running = True
```

In `update()` and `draw()`, the only thing that happens is the corresponding method call on the sprite group and now the paddles finally show up on screen.

Listing 4.8: Pong (Requirement 2) – `Game.update()` and `Game.draw()`

```
    def update(self):
        self.all_sprites.update(action="move")      # Move

    def draw(self):
        self.background.draw(self.screen)
        self.all_sprites.draw(self.screen)          #
        self.window.flip()
```

And now the keyboard events are handled. Pressing a key triggers a movement (starting at line 101), while releasing the key causes the corresponding paddle to stop (starting at line 109).

In each case, the method `Paddle.update()` is called with an appropriate parameter: for movement with `action="up"` or `action="down"`, and for stopping with `action="halt"`.

Listing 4.9: Pong (Requirement 2) – `Game.watch_for_events()`

```
94      def watch_for_events(self):
95          for event in pygame.event.get():
96              if event.type == pygame.QUIT:
97                  self.running = False
98              elif event.type == pygame.KEYDOWN:
99                  if event.key == pygame.K_ESCAPE:
100                     self.running = False
101                 elif event.key == pygame.K_UP:      # Paddle moves
102                     self.paddle["right"].update(action="up")
103                 elif event.key == pygame.K_DOWN:
104                     self.paddle["right"].update(action="down")
105                 elif event.key == pygame.K_w:
106                     self.paddle["left"].update(action="up")
107                 elif event.key == pygame.K_s:
108                     self.paddle["left"].update(action="down")
```

```
            elif event.type == pygame.KEYUP:        # Paddle stops
                if event.key == pygame.K_UP or event.key == pygame.K_DOWN:
                    self.paddle["right"].update(action="halt")
                if event.key == pygame.K_w or event.key == pygame.K_s:
                    self.paddle["left"].update(action="halt")
```

4.1.3 Requirement 3: The Ball

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/01%20Pong/v03

Requirement 3 Ball

1. *The ball is a circle with a radius of 10 px.*
2. *Its speed is* $\frac{\textit{screen width}}{3}$ *px/s.*
3. *It starts in the center of the screen with a random horizontal and vertical direction.*
4. *It bounces off the top and bottom edges of the screen.*
5. *When it touches the left edge, it is reset to the center. The same happens when it touches the right edge.*
6. *If the right edge is hit, player 1 scores a point; if the left edge is hit, player 2 scores a point.*

Since we need to keep track of the players' scores according to requirement 3.6, a corresponding array is added to `config.py` (line 6).

Listing 4.10: Pong (Requirement 3) – `config.py`

```
1 from pygame import Rect
2
3 WINDOW = Rect(0, 0, 1000, 600)
4 FPS = 60
5 DELTATIME = 1.0 / FPS
6 POINTS = [0, 0]                                   # Score
```

In accordance with requirements 3.1 and 3.2, the size and speed of the ball are defined in line 65 and line 69. Since the ball needs to be restarted frequently, the initialization of its starting position and direction is moved into the separate method `service()` (line 71).

Listing 4.11: Pong (Requirement 3) – Constructor of `Ball`

```
class Ball(pygame.sprite.Sprite):
    def __init__(self, *groups: Tuple[pygame.sprite.Group]) -> None:
        super().__init__(*groups)
        self.rect = pygame.FRect(0, 0, 20, 20)     # Size
        self.image = pygame.Surface(self.rect.size).convert()
        self.image.set_colorkey("black")
        pygame.draw.circle(self.image, "green", self.rect.center, self.rect.width // 2)
        self.speed = cfg.WINDOW.width // 3               # Speed
        self.speedxy = pygame.Vector2()
        self.service()                                   #
```

In `update()`, the responsibilities are distributed.

Listing 4.12: Pong (Requirement 3) – `Ball.update()`

```
    def update(self, *args: Any, **kwargs: Any) -> None:
        if "action" in kwargs.keys():
            if kwargs["action"] == "move":
                self.move()
            elif kwargs["action"] == "service":
                self.service()
        return super().update(*args, **kwargs)
```

Let us now take a closer look at the helper methods, one by one. We start with `move()`. As expected, the position is updated using the velocity values. After that, starting at line 83, it is checked whether the ball has reached any of the four edges of the screen.

If the top or bottom edge is hit (requirement 3.4), the sign of the vertical velocity is inverted by calling `vertical_flip()` (source code 4.15 on the next page). After the flip, the ball is clamped to the top or bottom edge, since it may already have crossed the boundary.

Things are different when the ball reaches the left or right edge. In that case, the ball is served again according to requirement 3.5 (see source code 4.14 on the following page), and – as specified in requirement 3.6 – the appropriate player's score is increased.

Listing 4.13: Pong (Requirement 3) – `Ball.move()`

```
81      def move(self) -> None:
82          self.rect.move_ip(self.speedxy * cfg.DELTATIME)
83          if self.rect.top <= 0:                           #
84              self.vertical_flip()
85              self.rect.top = 0
86          elif self.rect.bottom >= cfg.WINDOW.bottom:
87              self.vertical_flip()
88              self.rect.bottom = cfg.WINDOW.bottom
89          elif self.rect.right < 0:
90              cfg.POINTS[1] += 1
91              self.service()
92          elif self.rect.left > cfg.WINDOW.right:
93              cfg.POINTS[0] += 1
94              self.service()
```

When serving, the center of the ball is set to the center of the screen (requirement 3.3). After that, the signs of the two velocity components are chosen randomly, which determines the direction of movement (left or right, and up or down). Since we do not have a score display yet, a temporary console output is implemented in line 99.

Listing 4.14: Pong (Requirement 3) – `Ball.service()`

```
96      def service(self) -> None:
97          self.rect.center = cfg.WINDOW.center
98          self.speedxy = pygame.Vector2(choice([-1, 1]), choice([-1, 1])) * self.speed
99          print(cfg.POINTS)                               # Ugly
```

The direction change is simply a sign flip. The method `flip_horizontal()` is not used yet, but we will need it later when we want the ball to bounce off the paddle.

Listing 4.15: Pong (Requirement 3) – The flip methods of `Ball`

```
    def horizontal_flip(self) -> None:
        self.speedxy.x *= -1

    def vertical_flip(self) -> None:
        self.speedxy.y *= -1
```

Typical reflection pitfalls when handling the ball

⚠ Sticky edge / multiple flips per frame

If the ball is still inside the wall after a flip (because it has already crossed the boundary), the velocity is inverted again in the next frame. The result is a ball that appears to jitter or stick to the edge.

Fix: After flipping the velocity, clamp the position explicitly (e.g. `rect.top = 1` or `rect.bottom = WINDOW.HEIGHT - 1`).

⚠ Checking the wrong reference (center vs. rect)

A common mistake is to compute movement using the ball center or a position vector, but perform collision checks against `rect.left/right/top/bottom` (or vice versa). This usually leads to off-by-radius errors.

Fix: Be consistent: either check collisions exclusively using `rect.*`, or use `center` together with ± `radius` – but do not mix both approaches.

⚠ Flipping the wrong axis

A classic error: when hitting the top or bottom wall, `speed.x` is inverted instead of `speed.y` (or the other way around).

Fix:

- Top / bottom collision: `speed.y *= -1`
- Left / right collision: `x *= -1` (or trigger a service/reset)

Tunneling at high speed

With large `DELTATIME` values or high velocities, the ball may jump over a wall between two frames and never register a collision.

Fix (simple): After moving the ball, check whether it has crossed a boundary and clamp it back.

Fix (robust): Split the movement into smaller steps (sub-stepping) or use swept collision detection.

Incorrect handling of multiple collisions in one frame

If the ball hits, for example, a corner (top wall and paddle edge at the same time), naively flipping both axes can cancel out the reflection entirely.

Fix: Prioritize collisions (e.g. wall before paddle), or decide based on the smaller penetration depth.

Not pushing the ball out of the paddle after a hit

If the ball remains inside the paddle rectangle after a bounce, it will flip direction again in the next frame and appear to vibrate.

Fix: After a paddle collision, move the ball explicitly in front of the paddle edge (clamp), and only then invert `vx`.

4.1.4 Requirement 4: Scoring

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/01%20Pong/v04

Requirement 4 Scoring

1. *The score is displayed centered at the top of the screen.*

For displaying the score, the class `Score` is used. In the end, it is just another sprite—but one that needs to be recreated from time to time, namely whenever the score changes. Since the current score is now stored in line 118, it can be removed from `config.py`.

Listing 4.16: Pong (Requirement 4) – Constructor of `Score`

```python

    def __init__(self, *groups: Tuple[pygame.sprite.Group]):
        super().__init__(*groups)
        self.font = pygame.font.SysFont(None, 30)
        self.score = {1: 0, 2: 0}                    # Not in Settings anymore!
        self.image: pygame.Surface = None
        self.rect: pygame.Rect = None
        self.render()
```

In this method, the current score is rendered using a font object and then positioned accordingly.

Listing 4.17: Pong (Requirement 4) – `Score.render()`

```python
        self.image = self.font.render(f"{self.score[1]} : {self.score[2]}", True, "white")
        self.rect = self.image.get_rect(centerx=cfg.WINDOW.centerx, top=15)
```

In `update()`, the appropriate score value is updated and `render()` is called.

Listing 4.18: Pong (Requirement 4) – `Score.update()`

```python
        if "player" in kwargs.keys():
            self.score[kwargs["player"]] += 1
            self.render()
        return super().update(*args, **kwargs)
```

What is still missing is a trigger for updating the score display. This is a perfect opportunity to introduce a user-defined event. Starting at line 7, everything required for such a user event is implemented. First, an event ID is defined, followed by the corresponding `pygame.event.Event` object.

Listing 4.19: Pong (Requirement 4) – `MyEvent`

```python
1 import pygame
2
3 WINDOW = pygame.Rect(0, 0, 1000, 600)
4 FPS = 60
5 DELTATIME = 1.0 / FPS
6
7 class MyEvents:                                     # User events
8     POINT_FOR = pygame.USEREVENT
9     MYEVENT = pygame.event.Event(POINT_FOR, player=0)
```

Now the `Ball` class only has to trigger the appropriate event, and `Game` needs to handle it. Here are the required changes in `Ball`. Inside the method `move()`, the relevant code sections are replaced. For example, in line 94 the number of the player who scores the point is packed into the event, and in line 95 the event is dispatched.

Listing 4.20: Pong (Requirement 4) – `Ball.move()`

```python
        self.rect.move_ip(self.speedxy * cfg.DELTATIME)
        if self.rect.top <= 0:
            self.vertical_flip()
            self.rect.top = 0
        elif self.rect.bottom >= cfg.WINDOW.bottom:
            self.vertical_flip()
            self.rect.bottom = cfg.WINDOW.bottom
        elif self.rect.right < 0:
            MyEvents.MYEVENT.player = 2             # Player
            pygame.event.post(MyEvents.MYEVENT)     # Shoot event
            self.service()
        elif self.rect.left > cfg.WINDOW.right:
            MyEvents.MYEVENT.player = 1
            pygame.event.post(MyEvents.MYEVENT)
            self.service()
```

Now all that remains is to catch the user-defined event inside `watch_for_events()` (starting at line **??**).

Listing 4.21: Pong (Requirement 4) – `Ball.watch_for_events()`

```python
                self.score.update(player=event.player)
```

Why is a user-defined event more elegant than direct access?

- **No direct access from the ball to the score:** If the `Ball` were to call `Score.render()` or `Score.update()` directly, it would need to know about the `Score` object – or even the `Game` class. This creates unnecessary dependencies and tightly couples classes that should remain independent.
- **Clear separation of responsibilities:** The `Ball` only knows one thing: *A point was scored by player X.* The `Game`, on the other hand, decides what that means in practice. Update the score data, re-render the score display, maybe play a sound, reset the ball, or start the next serve. Each class focuses on its own responsibility (SRP).
- **A clean extension point:** Later on, additional reactions can easily be attached to the same event – such as sound effects, particle effects, a short pause, a change in serve direction, or logging – without touching the `Ball` code again.
- **Better testability and maintainability:** All scoring-related behavior is handled centrally in `Game.watch_for_events()`, instead of being scattered across multiple classes. This makes the code easier to understand, test, and maintain.

The ball reports what happened and the game decides what to do about it.

4.1.5 Requirement 5: Paddle hit

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/01%20Pong/v05

At first glance, the game already looks finished – but it is still not really playable, because the paddles are not doing anything yet.

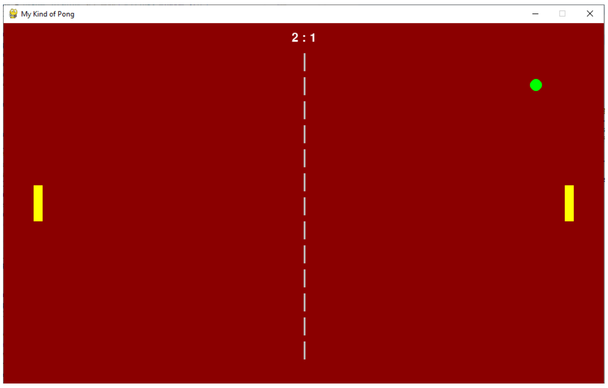

Figure 4.2: Pong: paddles, ball, and score

Requirement 5 Paddle hit

---

1. *When the ball touches a paddle, it bounces off and is returned to the opponent's side of the field.*
2. *Each time the ball hits a paddle, its directional velocities are increased by a small random amount.*

To achieve this, we add the method `check_collision()` to the `Game` class. This method checks whether the ball has hit one of the paddles. A good choice here is the method `pygame.sprite.collide_rect()`.

If a collision is detected, the previously unused method `horizontal_flip()` (see source code 4.15 on page 230) is triggered via `update()`. Afterwards, the positions are adjusted so that the ball and the paddle no longer overlap. In addition, the method `respeed()` is called via `update()` to fulfill requirement 5.2.

Listing 4.22: Pong (Requirement 5) – `Game.check_collision()`

```
    def check_collision(self):
        if pygame.sprite.collide_rect(self.ball, self.paddle["left"]):
            self.ball.update(action="hflip")
```

```
            self.ball.rect.left = self.paddle["left"].rect.right + 1
        elif pygame.sprite.collide_rect(self.ball, self.paddle["right"]):
            self.ball.update(action="hflip")
            self.ball.rect.right = self.paddle["right"].rect.left - 1
```

In `respeed()`, small random values are added to the velocity components. Via the attribute `speed`, this variation is indirectly tied to the screen size.

Listing 4.23: Pong (Requirement 5) – `Ball.respeed()`

```
    def respeed(self) -> None:
        self.speedxy.x += randrange(0, self.speed // 4)
        self.speedxy.y += randrange(0, self.speed // 4)
```

Now the game finally becomes playable.

4.1.6 Requirement 6: Computer-controlled player

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/01%20Pong/v06

Strictly speaking, we would be finished at this point – but I would like to add a computer-controlled player. This allows the game to be played against the computer, or simply to let the computer play against itself for hours.

Requirement 6 Computer player

1. *Pressing* 1 *toggles control of the left paddle between human and computer.*
2. *Pressing* 2 *toggles control of the right paddle between human and computer.*
3. *When control is switched back to manual, the paddle should initially remain stationary.*

In `config.py`, a dictionary of flags is defined in line 6. These flags control, for each player, whether the paddle is controlled manually or by the computer.

Listing 4.24: Pong (Requirement 6) – `config.py`

```
1 import pygame
2
3 WINDOW = pygame.Rect(0, 0, 1000, 600)
4 FPS = 60
```

```python
DELTATIME = 1.0 / FPS
KI = {"left": False, "right": False}             # Flag computer player

class MyEvents:
    POINT_FOR = pygame.USEREVENT
    MYEVENT = pygame.event.Event(POINT_FOR, player=0)
```

In the `update()` method, starting at line 187, the flags are checked to determine whether a paddle is controlled by the computer. If so, a corresponding controller method is called.

Listing 4.25: Pong (Requirement 6) – `Game.update()`

```python
185     def update(self):
186         self.check_collision()
187         for i in cfg.KI.keys():                  # Computer commands
188             if cfg.KI[i]:
189                 self.paddlecontroler(self.paddle[i])
190         self.all_sprites.update(action="move")
```

Let us now take a look at the controller method. The basic idea is simple: the paddle moves upward as long as the center of the ball is above the center of the paddle, and it moves downward as long as the ball's center is below the paddle's center.

There is no need to move all the way to the very top or bottom. The last few pixels can be ignored, since a collision will usually be triggered before that anyway.

Why does this simple computer player work so well?

At first glance, this controller logic looks almost trivial: the paddle simply follows the vertical position of the ball. Surprisingly, this already produces a reasonably strong computer opponent.

The reason is that Pong is a very simple game in terms of physics and decision-making. The ball moves along a straight line between collisions, and its vertical position is the most important piece of information needed to intercept it. By continuously aligning the paddle's center with the ball's center, the computer ensures that the paddle is usually in the right place at the right time.

Another advantage of this approach is that it is stable and predictable. The paddle does not overreact, oscillate wildly, or make unnecessary movements. Since the paddle speed is limited, it also cannot instantly teleport to the ball's position, which keeps the game fair.

Finally, stopping the paddle slightly before reaching the exact ball position is intentional. This avoids jitter and unnecessary micro-movements, and in practice a collision will occur anyway once the ball reaches the paddle.

follow the ball

In short: For simple games like Pong, a straightforward *follow the ball* strategy is often more than sufficient – and a great example of how simple rules can lead to convincing behavior.

Listing 4.26: Pong (Requirement 6) – `Game.paddlecontroler()`

```
    def paddlecontroler(self, paddle: pygame.sprite.Sprite) -> None:
        if paddle.rect.centery > self.ball.rect.centery and paddle.rect.top > 10:
            paddle.update(action="up")
        elif paddle.rect.centery < self.ball.rect.centery and paddle.rect.bottom <
            cfg.WINDOW.bottom - 10:
            paddle.update(action="down")
        else:
            paddle.update(action="halt")
```

In `watch_for_events()`, more extensive changes are required. First, manual control for a paddle must be disabled whenever that paddle is set to computer control. So, before calling the corresponding `update()` method, we first check whether the computer player currently has control. An example can be found in line 205.

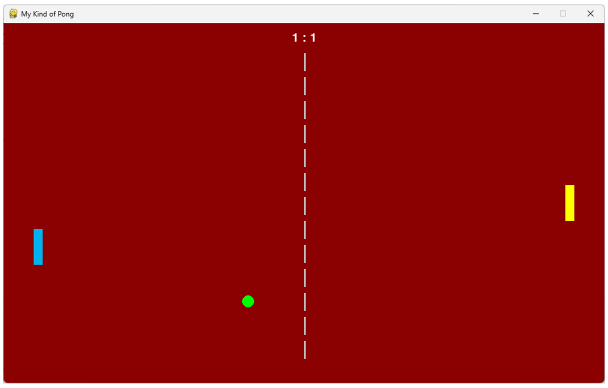

Figure 4.3: Pong: paddle color indicates AI mode (left AI, right manual)

One remaining detail is requirement 6.3. For this, the corresponding flag is checked as shown in line 218, and the paddle is sent a halt signal.

Listing 4.27: Pong (Requirement 6) – `Game.watch_for_events()`

```
197     def watch_for_events(self):
198         for event in pygame.event.get():
199             if event.type == pygame.QUIT:
200                 self.running = False
201             elif event.type == pygame.KEYDOWN:
202                 if event.key == pygame.K_ESCAPE:
203                     self.running = False
204                 elif event.key == pygame.K_UP:
205                     if not cfg.KI["right"]:    #
206                         self.paddle["right"].update(action="up")
207                 elif event.key == pygame.K_DOWN:
208                     if not cfg.KI["right"]:
209                         self.paddle["right"].update(action="down")
210                 elif event.key == pygame.K_w:
211                     if not cfg.KI["left"]:
212                         self.paddle["left"].update(action="up")
213                 elif event.key == pygame.K_s:
214                     if not cfg.KI["left"]:
215                         self.paddle["left"].update(action="down")
216                 elif event.key == pygame.K_1:
```

```
                    cfg.KI["left"] = not cfg.KI["left"]
                    if not cfg.KI["left"]:     # Stop!
                        self.paddle["left"].update(action="halt")
                elif event.key == pygame.K_2:
                    cfg.KI["right"] = not cfg.KI["right"]
                    if not cfg.KI["right"]:
                        self.paddle["right"].update(action="halt")
            elif event.type == pygame.KEYUP:
                if event.key == pygame.K_UP or event.key == pygame.K_DOWN:
                    if not cfg.KI["right"]:
                        self.paddle["right"].update(action="halt")
                elif event.key == pygame.K_w or event.key == pygame.K_s:
                    if not cfg.KI["left"]:
                        self.paddle["left"].update(action="halt")
            elif event.type == MyEvents.POINT_FOR:
                self.score.update(player=event.player)
```

4.1.7 Requirement 7: Sound

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/01%20Pong/v07

A bit of sound would make the game feel much more lively.

Requirement 7 Sound

1. *Hitting the ball with a paddle should be accompanied by an appropriate sound effect.*
2. *Bouncing off the top and bottom edges should also be accompanied by a suitable sound effect.*
3. *Sound should be toggleable on and off using the* F2.

As a first step, we extend `Settings` by adding the flag `SOUNDFLAG` in line 9. This flag controls whether sound should be played or not and provides access to the sound files.

Listing 4.28: Pong (Requirement 7) – `config.py`

```
1  import os
2
3  import pygame
4
5  WINDOW = pygame.Rect(0, 0, 1000, 600)
6  FPS = 60
7  DELTATIME = 1.0 / FPS
8  KI = {"left": False, "right": False}
9  SOUNDFLAG = True                           # Sound flag
10 PATH = {}
11 PATH["file"] = os.path.dirname(os.path.abspath(__file__))
```

```python
PATH["sound"] = os.path.join(PATH["file"], "sounds")

def get_sound(filename: str) -> str:
    return os.path.join(PATH["sound"], filename)

class MyEvents:
    POINT_FOR = pygame.USEREVENT
    MYEVENT = pygame.event.Event(POINT_FOR, player=0)
```

The actual sound playback is implemented in the `Ball` class. In the constructor, starting at line 84, the sound effects are loaded and a channel is selected through which the sounds will be played.

Listing 4.29: Pong (Requirement 7) – Constructor of `Ball`

```python
81 class Ball(pygame.sprite.Sprite):
82     def __init__(self, *groups: Tuple[pygame.sprite.Group]) -> None:
83         super().__init__(*groups)
84         self.sounds: dict[str, pygame.mixer.Sound] = {}  # Sound container
85         self.sounds["left"] = pygame.mixer.Sound(cfg.get_sound("playerl.mp3"))
86         self.sounds["right"] = pygame.mixer.Sound(cfg.get_sound("playerr.mp3"))
87         self.sounds["bounce"] = pygame.mixer.Sound(cfg.get_sound("bounce.mp3"))
88         self.channel = pygame.mixer.find_channel()
89         self.rect = pygame.FRect(0, 0, 20, 20)
90         self.image = pygame.Surface(self.rect.size).convert()
91         self.image.set_colorkey("black")
92         pygame.draw.circle(self.image, "green", self.rect.center, self.rect.width // 2)
93         self.speed = cfg.WINDOW.width // 3
94         self.speedxy = pygame.Vector2()
95         self.service()
```

The first sound effect is implemented for paddle collisions in `horizontal_flip()`. After checking whether sound output is enabled at all, it is determined whether the ball is bouncing off the left or the right paddle. This is done indirectly by checking the current horizontal direction of the ball (line 132). Based on this information (see section 2.11.2.1 on page 160, the volume of the sound is adjusted so that it creates the impression that the bounce happens to the left or right of the listener.

stereo panning

Listing 4.30: Pong (Requirement 7) – `Ball.horizontal_flip()`

```python
130     def horizontal_flip(self) -> None:
131         if cfg.SOUNDFLAG:
132             if self.speedxy.x < 0:                  # Ball to left?
133                 self.channel.set_volume(0.9, 0.1)
134                 self.channel.play(self.sounds["left"])
135             else:
136                 self.channel.set_volume(0.1, 0.9)
137                 self.channel.play(self.sounds["right"])
138         self.speedxy.x *= -1
139         self.respeed()
```

This sound effect becomes a bit more dynamic in `vertical_flip()`. In line 143, the relative horizontal position of the ball is calculated. If the center of the ball is on the left side, `rel_pos` will be close to 0; if the ball is far to the right, the value will be close to 1.

These values can then be used directly as the left and right volume levels when calling `set_volume()`, creating a simple but effective stereo panning effect.

Listing 4.31: Pong (Requirement 7) – `Ball.vertical_flip()`

```python
    def vertical_flip(self) -> None:
        if cfg.SOUNDFLAG:
            rel_pos = self.rect.centerx / cfg.WINDOW.width  # Where am I?
            self.channel.set_volume(1.0 - rel_pos, rel_pos)
            self.channel.play(self.sounds["bounce"])
        self.speedxy.y *= -1
```

All that remains is toggling sound output on and off inside `watch_for_events()` in line 227 using the function key F2.

Listing 4.32: Pong (Requirement 7) – `Ball.watch_for_events()`

```python
213     def watch_for_events(self):
214         for event in pygame.event.get():
215             if event.type == pygame.QUIT:
216                 self.running = False
217             elif event.type == pygame.KEYDOWN:
218                 if event.key == pygame.K_ESCAPE:
219                     self.running = False
220                 elif event.key == pygame.K_UP:
221                     if not cfg.KI["right"]:
222                         self.paddle["right"].update(action="up")
223                 elif event.key == pygame.K_DOWN:
224                     if not cfg.KI["right"]:
225                         self.paddle["right"].update(action="down")
226                 elif event.key == pygame.K_F2:
227                     cfg.SOUNDFLAG = not cfg.SOUNDFLAG  # Toogle Soundflag
```

4.1.8 Requirement 8: Pause and Help Screen

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/01%20Pong/v08

Requirement 8 Pause and help

1. *Pressing* p *pauses all activity and stops the game. Pressing* p *again resumes the game.*
2. *Pressing* h *pauses the game and displays a help text. Pressing* h *again resumes the game.*

For the pause functionality, we create a separate class—perhaps a bit overengineered, but nicely self-contained. The essential part can be found in line 33. There, a semi-transparent gray overlay is created using a `Surface` object with the same size as the screen. The surface is filled with a gray color whose alpha channel is set to 200, allowing the background to shine through.

Listing 4.33: Pong (Requirement 8) – `Pause`

```
28 class Pause(pygame.sprite.Sprite):
29     def __init__(self, *groups: Tuple[pygame.sprite.Group]) -> None:
30         super().__init__(*groups)
31         self.rect = pygame.Rect(cfg.WINDOW.topleft, cfg.WINDOW.size)
32         self.image = pygame.Surface(self.rect.size).convert_alpha()
33         self.image.fill([120, 120, 120, 200])       # Transparent Grey
```

The help screen is implemented in an analogous way. The only difference is that an additional text is blitted onto the surface. The text is split into a left and a right column to improve readability.

Listing 4.34: Pong (Requirement 8) – `Help`

```
class Help(pygame.sprite.Sprite):
    def __init__(self, *groups: Tuple[pygame.sprite.Group]) -> None:
        super().__init__(*groups)
        self.rect = pygame.Rect(cfg.WINDOW.topleft, cfg.WINDOW.size)
        self.image = pygame.Surface(self.rect.size).convert_alpha()
        self.image.fill([20, 20, 20, 200])          # Transparent Grey
        font = pygame.font.Font(pygame.font.get_default_font(), 20)
        text_l = "h\np\nESC\n\nF2\nk\nl\nr\n\nUP\nDOWN\nw\ns"
        text_r = "- toogle help modus\n- toogle pause modus\n- quit\n\n- toogle sound modus\n"
        text_r += "- toogle both paddles KI modus\n- toogle left paddle KI modus\n- toogle right paddle KI modus\n\n"
        text_r += "- left paddle move up\n- left paddle move down\n- right paddle move up\n- right paddle move down"
        lines = font.render(text_l, True, "white")
        self.image.blit(lines, (10, 10))
        lines = font.render(text_r, True, "white")
        self.image.blit(lines, (10 + 70, 10))
```

In the constructor of `Game`, two flags now need to be created to represent the respective modes (line 213 and line 214). After that, the two overlay objects are created and assigned to a `pygame.Group.Single` object.

Listing 4.35: Pong (Requirement 8) – Constructor of `Game`

```
199 class Game:
200     def __init__(self):
201         pygame.init()
202         self.window = pygame.Window(size=cfg.WINDOW.size, title="My Kind of Pong", position=pygame.WINDOWPOS_CENTERED)
203         self.screen = self.window.get_surface()
204         self.clock = pygame.time.Clock()
205         self.background = pygame.sprite.GroupSingle(Background())
206         self.all_sprites = pygame.sprite.Group()
207         self.paddle = {}
```

```
        self.paddle["left"] = Paddle("left", self.all_sprites)
        self.paddle["right"] = Paddle("right", self.all_sprites)
        self.ball = Ball(self.all_sprites)
        self.score = Score(self.all_sprites)
        self.running = True
        self.pausing = False                        # Pause Flag
        self.helping = False                        # Help Flag
        self.pause = pygame.sprite.GroupSingle(Pause())
        self.help = pygame.sprite.GroupSingle(Help())
```

Once everything is prepared, the `update()` method is modified so that the actual game logic is only executed when neither pause mode nor help mode is active (line 231). If one of these modes is enabled, the game state is effectively frozen: positions, movements, and collisions are no longer updated, while the current screen remains visible.

Listing 4.36: Pong (Requirement 8) – `Game.update()`

```
230     def update(self):
231         if not (self.pausing or self.helping):      #
232             self.check_collision()
233             for i in cfg.KI.keys():
234                 if cfg.KI[i]:
235                     self.paddlecontroler(self.paddle[i])
236             self.all_sprites.update(action="move")
```

In `draw()`, the currently active mode is checked as well. If the game is paused or the help screen is active, the corresponding sprite is rendered on top of the game scene. Otherwise, only the current game state is rendered as usual.

Listing 4.37: Pong (Requirement 8) – `Game.draw()`

```
    def draw(self):
        self.background.draw(self.screen)
        self.all_sprites.draw(self.screen)
        if self.pausing:                            #
            self.pause.draw(self.screen)
        elif self.helping:                          #
            self.help.draw(self.screen)
        self.window.flip()
```

Pause vs. Help – what happens technically?

Both the pause mode and the help screen are based on the same fundamental idea: the game is still rendered visually, but the actual game simulation is stopped.

- **Pause**: All movement and state-changing calculations are suspended. The current game situation is frozen and merely covered by a semi-transparent overlay.
- **Help**: Technically identical to the pause mode, but extended by an additional text overlay. The player receives information about controls and gameplay while the game state itself remains unchanged.

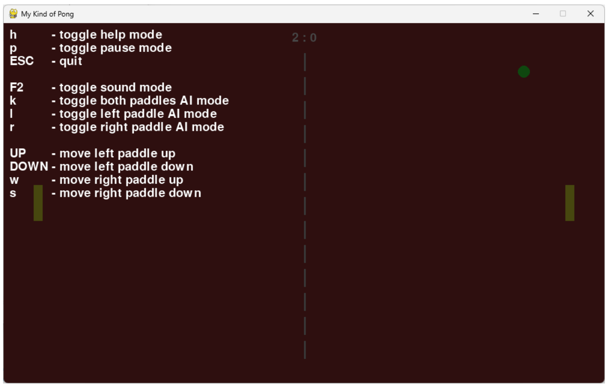

Figure 4.4: Pong: Help screen

The crucial point is that in both modes the method `update()` does not modify any game objects. Rendering continues, which keeps the game visually *alive* while it is logically paused.

This approach has several advantages

- No special-case logic inside individual sprite classes
- A clear separation between *game state* and *presentation*
- Easy to extend (e. g. for menus, dialogs, or settings)

Rule of thumb: Pausing does not mean drawing nothing – it means changing nothing.

4.2 Bubbles

In this chapter, the game *Bubbles* is discussed as an example. We will develop this game step by step in a systematic way. I will assume that the techniques introduced in chapter 1 are already familiar. I will deliberately omit docstring comments in the source code, since everything is explained in the text and the listings would otherwise become unnecessarily long. In the final version, however, these comments are included.

Thank you

I would like to point out right away that the idea for the game did not come from me. A student once presented it as a mobile version at an Information Technology Assistants (ITA) fair. Unfortunately, I can no longer remember the student's name, but I would like to take this opportunity to say a sincere *thank you*.

The game can be extended almost without limits: bubble popping animations, high score lists, and much more. But as is so often the case, the better is the enemy of the good. I hope you enjoy studying this example.

4.2.1 Requirement 1: Standards

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/02%20Bubbles/v01

Requirement 1 Standard functionality

1. *The window has an appropriate size.*
2. *The background is either a suitable bitmap or a solid color.*
3. *The game can be exited using* Esc *or by clicking the red "X".*
4. *All bitmaps are converted and scaled appropriately after loading.*
5. *All bitmaps – except for the background – are transparent.*
6. *All bitmaps are stored in* `pygame.sprite.Group` *or* `pygame.sprite.GroupSingle` *objects.*
7. *The game has a frame-rate-independent execution speed.*

Requirement 1 does not only define specific requirements, but also general ones. For this reason, it will appear again in later implementations.

At this point, the preamble is presented once. I assume that you have sufficient Python knowledge to extend it as needed. The static configuration values of the game are stored, as usual, in the separate `config.py` file.

It is required that the window has an appropriate size. With 1220 px × 1002 px, the window is large enough to distribute the bubbles, yet small enough to allow quick mouse movement. Everything else has already been discussed in detail in previous chapters (e.g. `FPS`, `DELTATIME`, or `PATH`) and will therefore not be explained further here.

Figure 4.5: Bubbles: background image

Listing 4.38: Bubbles (requirement 1.1) – `config.py`

```python
import os
from typing import Dict

import pygame

WINDOW = pygame.Rect(0, 0, 1220, 1002)
FPS = 60
DELTATIME = 1.0 / FPS
PATH: Dict[str, str] = {}
PATH["file"] = os.path.dirname(os.path.abspath(__file__))
PATH["image"] = os.path.join(PATH["file"], "images")
PATH["sound"] = os.path.join(PATH["file"], "sounds")
CAPTION = 'Bubbles'

def get_file(filename: str) -> str:
    return os.path.join(PATH["file"], filename)

def get_image(filename: str) -> str:
    return os.path.join(PATH["image"], filename)

def get_sound(filename: str) -> str:
    return os.path.join(PATH["sound"], filename)
```

The `Background` class is a subclass of `Sprite`. It is only loaded and scaled to the appropriate size. Since the background never changes, there is no need to implement an `update()` method. Creating a dedicated subclass for this is somewhat like using a sledgehammer to crack a nut. We could just as well have implemented it directly as a `Sprite` object. I chose this approach purely for the sake of clarity. The background image can be seen in figure 4.5 on the preceding page.

Listing 4.39: Bubbles (requirement 1.2) – `Background`

```python
class Background(pygame.sprite.Sprite):
    def __init__(self) -> None:
        super().__init__()
        imagename = cfg.get_image("aquarium.png")
        self.image: pygame.Surface = pygame.image.load(imagename).convert()
        self.image = pygame.transform.scale(self.image, cfg.WINDOW.size)
        self.rect: pygame.Rect = self.image.get_rect()
```

In the `Game` class, the usual Pygame suspects are initialized or created in `__init__()`: `init()`, `Window()`, and `clock()`. The flag `running` for the main game loop is also initialized. The methods `run()`, `watch_for_events()`, `update()`, and `draw()` contain only basic functionality and therefore do not need to be explained further at this point.

Listing 4.40: Bubbles (requirement 1) – `Game`

```python
class Game:

    def __init__(self) -> None:
        pygame.init()
        self.window = pygame.Window(size=cfg.WINDOW.size, title=cfg.CAPTION)
        self.screen = self.window.get_surface()
        self.clock = pygame.time.Clock()
        self.background = pygame.sprite.GroupSingle(Background())
        self.running = True

    def watch_for_events(self) -> None:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                self.running = False
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_ESCAPE:
                    self.running = False

    def draw(self) -> None:
        self.background.draw(self.screen)
        self.window.flip()

    def update(self) -> None:
        pass

    def run(self) -> None:
        time_previous = time()
        self.running = True
        while self.running:
            self.watch_for_events()
            self.update()
            self.draw()
            self.clock.tick(cfg.FPS)
            time_current = time()
            cfg.DELTATIME = time_current - time_previous
            time_previous = time_current
        pygame.quit()
```

However, these methods already define the overall flow of the game. All further properties of the game are merely extensions of this flow and no longer change it. Finally, the call is made (see source code 4.41). With this, all subitems of requirement 1 on page 244 that apply here are fulfilled.

Listing 4.41: Bubbles (requirement 1) – invocation

```python
def main():
    Game().run()


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

4.2.2 Requirement 2: Bubbles appear

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/02%20Bubbles/v02

Requirement 2 Bubbles appear

1. *A bubble appears at a random position.*
2. *At the beginning, this happens every half second.*
3. *It has an initial radius of 15 px.*
4. *It keeps a minimum distance of 10 px from the edges.*
5. *It keeps a minimum distance of 10 px from all other bubbles.*

For the bubble, we use the already transparent graphic from figure 4.6. The random position still needs to be restricted. The aquarium does not fill the entire screen (see figure 4.5 on page 245); instead, it sits inside something like a TV frame. So we have to define a playing area (*playground*). The bubbles should only appear inside this area.

Fig. 4.6: Bubble

The playing area is a rectangle with an offset from the left and top edges of the screen – `left` and `top` – and a width (`width`) and height (`height`). The corresponding values are defined in line 16. The distance to the border of the playing area and the minimum distance between bubbles are defined in line 15 as 10 px, in accordance with requirement 2.4. The initial radius – and therefore the minimum radius – is set to 15 px in line 14 because of requirement 2.3. While playing, I noticed that smaller initial radii are simply too hard to see.

Listing 4.42: Bubbles (requirement 2) – additions in `config.py`

```
RADIUS = {"min": 15}                              # Radius to start with
DISTANCE = 50                                     # Border-/Bubbledistance
PLAYGROUND = pygame.Rect(90, 90, 1055, 615)# Rect inside aquarium
```

The `Timer` class is exactly the one described above in chapter 2.9 on page 136; everything is explained there.

Listing 4.43: Bubbles (requirement 2) – `Timer`

```
class Timer:
    def __init__(self, duration: int, with_start: bool = True) -> None:
        self.duration = duration
        if with_start:
            self._next = pygame.time.get_ticks()
        else:
            self._next = pygame.time.get_ticks() + self.duration

    def is_next_stop_reached(self) -> bool:
        if pygame.time.get_ticks() > self._next:
            self._next = pygame.time.get_ticks() + self.duration
            return True
        return False
```

Let us now take a look at the `Bubble` class. The constructor is self-explanatory; it only handles the usual suspects: `image`, `rect`, and `radius`. The `update()` method is currently empty, since no changes are required yet. However, the `randompos()` method is needed because of requirement 2.1. It calculates a new bubble center and assigns it to `rect`. If necessary, this method must be repeated until a free area is found (see requirement 2.4 and requirement 2.5).

Listing 4.44: Bubbles (requirement 2) – `Bubble`

```
class Bubble(pygame.sprite.Sprite):
    def __init__(self) -> None:
        super().__init__()
        self.radius = cfg.RADIUS["min"]
        imagename = cfg.get_image("bubble1.png")
        self.image: pygame.Surface = pygame.image.load(imagename).convert_alpha()
        self.image = pygame.transform.scale(self.image, (cfg.RADIUS["min"],
            cfg.RADIUS["min"]))
        self.rect: pygame.Rect = self.image.get_rect()

    def update(self, *args: Any, **kwargs: Any) -> None:
        pass

    def randompos(self) -> None:
        bubbledistance = cfg.DISTANCE + cfg.RADIUS["min"]
        centerx = randint(cfg.PLAYGROUND.left + bubbledistance, cfg.PLAYGROUND.right -
            bubbledistance)
        centery = randint(cfg.PLAYGROUND.top + bubbledistance, cfg.PLAYGROUND.bottom -
            bubbledistance)
        self.rect.center = (centerx, centery)
```

The `Game` class now has to be extended accordingly. In line 58, the `Background` object is created. line 59 creates a `Timer` object with an interval of 500 ms, where no bubbles are generated during the first interval (see requirement 2.2).

Listing 4.45: Bubbles (requirement 2) – Constructor of `Game`

```
class Game:
    def __init__(self) -> None:
        pygame.init()
        self.window = pygame.Window(size=cfg.WINDOW.size, title=cfg.CAPTION)
        self.screen = self.window.get_surface()
        self.clock = pygame.time.Clock()
        self.background = pygame.sprite.GroupSingle(Background())  #
        self.timer_bubble = Timer(500, False)       # Timer 500ms
        self.all_sprites = pygame.sprite.Group()    # All bubbles
        self.running = True
```

In the `draw()` method, only the `draw()` methods of the sprite groups are called. The `update()` method has also been adjusted; it now calls the `spawn_bubble()` method and thus delegates the task of creating new bubbles.

Listing 4.46: Bubbles (requirement 2) – `draw()` and `update()` of `Game`

```
    def draw(self) -> None:
        self.background.draw(self.screen)
        self.all_sprites.draw(self.screen)
        self.window.flip()

    def update(self) -> None:
        self.spawn_bubble()
```

The basic idea behind `spawn_bubble()` is to keep guessing a position for a new bubble until a free area is found.

Avoid endloss loop

To avoid ending up in an infinity loop, the number of attempts is limited to 100. If no free area is found, the bubble is not added to the sprite group – it is simply discarded.

For this purpose, the radius is temporarily increased (line 84) and then reduced back to its original value after the collision check (line 86).

This is an example showing that a method reference is passed to `pygame.sprite.spritecollide()` – in this case `pygame.sprite.collide_circle()` – and that the usual rectangle-based collision check is therefore not used.

sprite-collide()

collide_-circle()

Listing 4.47: Bubbles (requirement 2) – `spawn_bubble()` of `Game`

```
79      def spawn_bubble(self) -> None:
80          if self.timer_bubble.is_next_stop_reached():
81              b = Bubble()
82              for _ in range(100):
83                  b.randompos()
84                  b.radius += cfg.DISTANCE       # Distance to other bubbles
85                  collided = pygame.sprite.spritecollide(b, self.all_sprites, False,
                        pygame.sprite.collide_circle)
86                  b.radius -= cfg.DISTANCE       # Old radius!
87                  if not collided:
```

```
                    self.all_sprites.add(b)
                    break
```

The result can be seen in figure 4.7. The bubbles are evenly distributed across the playing area, and the required minimum distance to the edges and between the bubbles is maintained.

Figure 4.7: Bubbles: the bubbles have a minimum distance at the start

4.2.3 Requirement 3: Number of bubbles

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/02%20Bubbles/v03

Requirement 3 Number of bubbles

The maximum number of bubbles shall depend on the size of the playing area.

I want to define the maximum number in the Game class. Based on the available area, an upper limit is calculated:

Listing 4.48: Bubbles (requirement 3) – additions in config.py

```
17 MAX_BUBBLES = PLAYGROUND.height * PLAYGROUND.width // 10000  # A guess
```

This upper limit from line 17 is checked in line 81. A new bubble is only created if the maximum number has not yet been reached.

Listing 4.49: Bubbles (requirement 3) – additions in `Game.spawn_bubbles()`

```
    def spawn_bubble(self) -> None:
        if self.timer_bubble.is_next_stop_reached():
            if len(self.all_sprites) <= cfg.MAX_BUBBLES:  # Enough space?
                b = Bubble()
                for _ in range(100):
                    b.randompos()
                    b.radius += cfg.DISTANCE
                    collided = pygame.sprite.spritecollide(b, self.all_sprites, False,
                        pygame.sprite.collide_circle)
                    b.radius -= cfg.DISTANCE
                    if not collided:
                        self.all_sprites.add(b)
                        break
```

The rest of the program remains unchanged.

4.2.4 Requirement 4: Bubble growth

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/02%20Bubbles/v04

Requirement 4 Bubble growth

1. *Bubbles of different sizes are managed in a container.*
2. *The maximum radius of a bubble is 240 px.*

The purpose of requirement 4.1 is to save computing time. During the game, bubbles repeatedly start with a certain radius and then grow. Scaling the bitmap to the required size every single time would waste processing power – after all, the same bubble appears multiple times with identical radii. For this reason, it makes sense to scale the bubble once to all possible radii and store the results in a dictionary. The key used is the respective radius (see line 37).

scale()

The `get()` method then returns the appropriately scaled and ready-to-use image for a given radius. Before that, lines 40 and 41 check whether the radius lies within the valid range. If the radius is too large, the maximum value is used; if it is too small, the minimum value is applied instead.

Listing 4.50: Bubbles (requirement 4.1) – `BubbleContainer`

```
33 class BubbleContainer:
34     def __init__(self) -> None:
35         imagename = cfg.get_image("bubble1.png")
36         image: pygame.Surface = pygame.image.load(imagename).convert_alpha()
```

```
        self.images = {i: pygame.transform.scale(image, (i * 2, i * 2)) for i in
            range(cfg.RADIUS["min"], cfg.RADIUS["max"] + 1)}  #

    def get(self, radius: int) -> pygame.Surface:
        radius = max(cfg.RADIUS["min"], radius)  # Lower limit
        radius = min(cfg.RADIUS["max"], radius)  # Upper limit
        return self.images[radius]
```

So far, only a start value – and thus a lower bound – for the bubble radius has been defined in `Game`. This definition is now extended in line 14 in accordance with requirement 4.2 by adding a maximum radius.

Listing 4.51: Bubbles (requirement 4.2) – extension of `config.py`

```
14 RADIUS = {"min": 15, "max": 240}         #
```

The `BubbleContainer` is passed to the constructor of `Bubble`, allowing this class to retrieve images from it. A direct example of this can be found in line 50. The `image` attribute is set according to the current `radius`.

The `update()` method is no longer empty. Its main purpose is to make the bubble grow. To achieve this, the radius is continuously increased, which results in increasingly larger images being loaded from the `BubbleContainer` and displayed (line 62). The new radius is calculated in line 59. In the same line, this value is compared with the maximum radius from `config.py`, and the minimum of the two is selected. This logic prevents the radius from becoming too large.

Center-based scaling

But what is the purpose of lines 61 and 64? The reference point of an image in a sprite is its top-left corner. If the bubble grows, it would therefore expand to the right and downward; the left and top edges would remain fixed, which looks awkward. To avoid this, we store the old center point, load the new image, create the corresponding `Rect` object, and then move it back to the old center. This way, the bubble visually grows outward from its center in all directions.

Listing 4.52: Bubbles (requirement 4) – extension of `Bubble`

```
45 class Bubble(pygame.sprite.Sprite):
46     def __init__(self, bubble_container: BubbleContainer) -> None:
47         super().__init__()
48         self.bubble_container = bubble_container    # Reference to container
49         self.radius = cfg.RADIUS["min"]
50         self.image = self.bubble_container.get(self.radius)  # Get bubble
51         self.rect: pygame.Rect = self.image.get_rect()
52         self.fradius = float(self.radius)
53         self.speed = 100
54 
55     def update(self, *args: Any, **kwargs: Any) -> None:
56         if "action" in kwargs.keys():
57             if kwargs["action"] == "grow":
```

```python
                self.fradius += self.speed * cfg.DELTATIME
                self.fradius = min(self.fradius, cfg.RADIUS["max"])  # New radius
                self.radius = round(self.fradius)
                center = self.rect.center           # Save center pos
                self.image = self.bubble_container.get(self.radius)  # New image
                self.rect = self.image.get_rect()
                self.rect.center = center           # Restore center pos

    def randompos(self) -> None:
        bubbledistance = cfg.DISTANCE + cfg.RADIUS["min"]
        centerx = randint(cfg.PLAYGROUND.left + bubbledistance, cfg.PLAYGROUND.right -
            bubbledistance)
        centery = randint(cfg.PLAYGROUND.top + bubbledistance, cfg.PLAYGROUND.bottom -
            bubbledistance)
        self.rect.center = (centerx, centery)
```

The `update()` method in `Game` only needs to be extended by calling all `update()` methods of the bubbles. This can be done very conveniently using the sprite group mechanism. Just like with `draw()`, `update()` can be called for the entire group in a single step (see line 99).

Listing 4.53: Bubbles (requirement 4) – extension of `Game.update()`

```python
98      def update(self) -> None:
99          self.all_sprites.update(action="grow")      # Bubbles growing
100         self.spawn_bubble()
```

Create the BubbleContainer

Listing 4.54: Bubbles (requirement 4) – extension of the constructor of `Game`

```python
        self.background = pygame.sprite.GroupSingle(Background())
        self.bubble_container = BubbleContainer()
        self.timer_bubble = Timer(500, False)
```

And in the `spawn_bubble()` method, the constructor call of `Bubble` is extended by passing the `BubbleContainer`.

Listing 4.55: Bubbles (requirement 4) – extension of `Game.spawn_bubble()`

```python
    def spawn_bubble(self) -> None:
        if self.timer_bubble.is_next_stop_reached():
            if len(self.all_sprites) <= cfg.MAX_BUBBLES:
                b = Bubble(self.bubble_container)  #
                for _ in range(100):
```

The bubbles now grow outward from their center. The result might then look like the one shown in figure 4.8 on the following page.

Figure 4.8: Bubbles – the bubbles have grown and merged

4.2.5 Requirement 5: Mouse cursor

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/02%20Bubbles/v05

Requirement 5 Mouse cursor

If the mouse is inside a bubble, its appearance should change.

This requirement is intended to provide visual feedback to the player. It allows them to recognize more quickly whether they have already reached a bubble. Pygame itself does not provide a method or function to test whether a point lies inside a circle. However, figure 4.9 on the next page provides a simple approach to solving this problem.

The value d represents the distance in pixels between the center of the circle (x_1, y_1) and the point (x_2, y_2). If $d \leq r$, the point lies inside the circle or touches it. However, we do not actually need the distance itself. Put simply, we only need to know whether the expression on the left side of the inequality is smaller than the one on the right side. We can therefore avoid the expensive square root operation and instead check $(x_2 - x_1)^2 + (y_2 - y_1)^2 \leq r^2$. We therefore extend `Bubble` with an appropriate method.

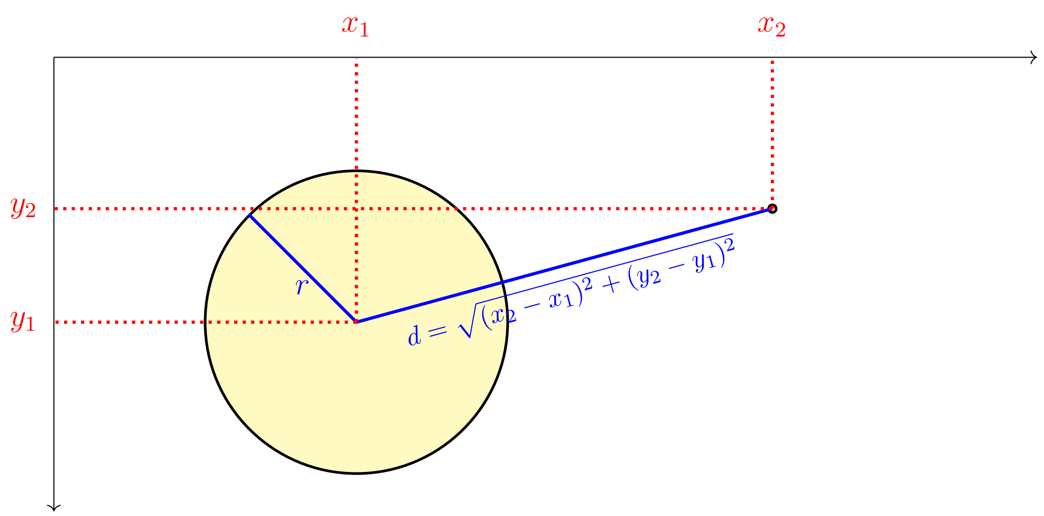

Figure 4.9: collision detection – point inside a circle (Pythagorean theorem)?

Listing 4.56: Bubbles (requirement 5) – `Game.collidepoint()`

```python
    def collidepoint(self, point: Tuple[int, int], sprite: pygame.sprite.Sprite) -> bool:
        if hasattr(sprite, "radius"):
            deltax = point[0] - sprite.rect.centerx
            deltay = point[1] - sprite.rect.centery
            return deltax * deltax + deltay * deltay <= pow(sprite.radius, 2)
        return False
```

With the help of this method, the solution is no longer a problem. The variable `is_over` is a flag that keeps track of whether the mouse coordinates are inside a bubble or not. The normal case is that the mouse is not inside any bubble, so the variable is initialized with `False`.

After that, the current mouse position is obtained using `pygame.mouse.get_pos()`. This mouse position is passed to the `Bubble.collidepoint()` method in line 128. If a bubble is found that collides with the mouse, the flag is set to `True` and the loop is terminated using `break`. This saves some processing time, since not all remaining bubbles have to be checked. Depending on the flag, the mouse cursor is then set accordingly.

Listing 4.57: Bubbles (requirement 5) – `Game.set_mousecursor()`

```python
124     def set_mousecursor(self) -> None:
125         is_over = False
126         pos = pygame.mouse.get_pos()
127         for b in self.all_sprites:
128             if self.collidepoint(pos, b):           # Mouse pos inside?
129                 is_over = True
130                 break
131         if is_over:
132             pygame.mouse.set_cursor(pygame.SYSTEM_CURSOR_CROSSHAIR)
133         else:
134             pygame.mouse.set_cursor(pygame.SYSTEM_CURSOR_HAND)
```

The `update()` method in `Game` still needs to be extended by adding the call to the collision check.

Listing 4.58: Bubbles (requirement 5) – `update()` in `Game`

```
    def update(self) -> None:
        self.all_sprites.update(action="grow")
        self.spawn_bubble()
        self.set_mousecursor()                      #
```

Try running the program. Place the mouse in a lower-left corner outside a bubble and wait until the growing bubble touches the mouse.

4.2.6 Requirement 6: Bubbles burst

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/02%20Bubbles/v06

Requirement 6 Bubbles burst

When a left mouse click occurs inside a bubble, the bubble should burst.

MOUSEBUTTONDOWN
get_pos()

Most of the work required to implement this requirement has already been done with the implementation of the `Bubble.collidepoint()` method. We only need to use this method in a clever way – in fact, only a few remaining steps are necessary. In `watch_for_events()`, the left mouse click is detected first (line 92) and the current mouse position is passed to the newly created `sting()` method (line 94).

`watch_for_events()` as a dispatcher

As a general rule, implement as little logic as possible in `watch_for_events()`. This method acts as a dispatcher; the actual processing should always be delegated to separate methods.

Listing 4.59: Bubbles (requirement 6) – `Game.watch_for_event()`

```
85      def watch_for_events(self) -> None:
86          for event in pygame.event.get():
87              if event.type == pygame.QUIT:
88                  self.running = False
89              elif event.type == pygame.KEYDOWN:
90                  if event.key == pygame.K_ESCAPE:
91                      self.running = False
```

```
            elif event.type == pygame.MOUSEBUTTONDOWN:  # Mouse clicked?
                if event.button == 1:  # left
                    self.sting(pygame.mouse.get_pos())  #
```

The `sting()` method is now very simple. All `Bubble` objects are iterated over and checked to see whether the mouse position lies within their radius (line 140). If the answer is *yes*, the corresponding object is removed from the sprite group using `kill()`.

kill()

Listing 4.60: Bubbles (requirement 6) – `Game.sting()`

```
138     def sting(self, mousepos: Tuple[int, int]) -> None:
139         for bubble in self.all_sprites:
140             if self.collidepoint(mousepos, bubble): # Inside?
141                 bubble.kill()
```

4.2.7 Requirement 7: Score

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/02%20Bubbles/v07

Requirement 7 Score

1. *The game starts with 0 points.*
2. *When a bubble bursts, the score is increased proportionally to its radius.*
3. *The score is displayed in the lower part of the screen.*

Popping bubbles should, of course, be rewarded with points. To do this, the score has to be calculated and displayed. The simplest way to keep track of the score is to use a variable in `config.py` or a global variable. I prefer option 1 (see source code 4.61).

Listing 4.61: Bubbles (requirement 7.1) – extension of `config.py`

```
POINTS = 0                                          # Score
```

Since popping a bubble no longer only makes it disappear but also updates the score, I added a new method to the `Bubble` class. In line 76, the radius of the bubble is simply added to the score.

Listing 4.62: Bubbles (requirement 7.2) – `Bubble.stung()`

```
74      def stung(self):
75          self.kill()
76          cfg.POINTS += self.radius                # Increment points
```

The call to `stung()` is triggered by an adjusted `update()` method.

Listing 4.63: Bubbles (requirement 7.2) – `Bubble.update()`

```
    def update(self, *args: Any, **kwargs: Any) -> None:
        if "action" in kwargs.keys():
            if kwargs["action"] == "grow":
                self.fradius += self.speed * cfg.DELTATIME
                self.fradius = min(self.fradius, cfg.RADIUS["max"])
                self.radius = round(self.fradius)
                center = self.rect.center
                self.image = self.bubble_container.get(self.radius)
                self.rect = self.image.get_rect()
                self.rect.center = center
            elif kwargs["action"] == "sting":
                self.stung()
```

The `sting()` and `update()` methods in `Game` have to be adjusted accordingly (see line 162 and line 123).

Listing 4.64: Bubbles (requirement 7.2) – `Game.sting()`

```
159     def sting(self, mousepos: Tuple[int, int]) -> None:
160         for bubble in self.all_sprites:
161             if self.collidepoint(mousepos, bubble):
162                 bubble.update(action="sting")        # Redesigned
```

Listing 4.65: Bubbles (requirement 7.2) – `Game.update()`

```
122     def update(self) -> None:
123         self.all_sprites.update(action="grow")       #
124         self.spawn_bubble()
125         self.set_mousecursor()
```

Rect

This leaves requirement 7.3. Similar to the playing area, I want to define the dimensions of the lower section as an output box in `config.py`.

Listing 4.66: Bubbles (requirement 7.3) – extension of `config.py`

```
            self._next = pygame.time.get_ticks() + self.duration
```

Font

For displaying the score itself, I once again create a small class that encapsulates this task: `Points`. In the constructor, a `Font` object is created, which is then used in `update()` to render the score. The position of the text output is determined from the values defined in `config.py`. The rest is handled for me by the `Sprite` class.

Listing 4.67: Bubbles (requirement 7.3) – Class `Points`

```
class Points(pygame.sprite.Sprite):
    def __init__(self) -> None:
        super().__init__()
        self.font = pygame.font.Font(pygame.font.get_default_font(), 18)
        self.oldpoints = -1
```

```python

    def update(self, *args: Any, **kwargs: Any) -> None:
        if self.oldpoints != cfg.POINTS:
            self.image = self.font.render(f"Points: {cfg.POINTS}", True, "red")
            self.rect = self.image.get_rect()
            self.rect.left = cfg.BOX.left
            self.rect.top = cfg.BOX.top
```

A few extensions remain in `Game`. In the constructor, the `Points` object is added to the `Group` object.

Listing 4.68: Bubbles (requirement 7.3) – extension of the `Game` constructor

```python
        self.all_sprites = pygame.sprite.Group()
        self.all_sprites.add(Points())
        self.running = True
```

Figure 4.10: Bubbles – score display

In figure 4.10, you can see the score display in the lower part of the screen. This area could later also be used for a list of the top ten scores or other types of output.

4.2.8 Requirement 8: Game over

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/02%20Bubbles/v08

Requirement 8 Game over

1. *If two bubbles touch, the game is lost.*
2. *If a bubble touches the edge, the game is lost.*

Note: To make the game playable, I set the growth speed of a bubble to 10.

Listing 4.69: Bubbles (requirement 8) – `Bubble.speed`

```
        self.speed = 10
```

The basic structure of our game makes it fairly easy to implement this requirement by extending the `update()` method in `Game`.

Listing 4.70: Bubbles (requirement 8) – extension of `Game.update()`

```
125     def update(self) -> None:
126         if self.check_bubblecollision():            # Game over?
127             self.running = False
128         else:
129             self.all_sprites.update(action="grow")
130             self.spawn_bubble()
131         self.set_mousecursor()
```

In the new method `check_bubblecollision()`, it is checked whether bubbles touch each other or whether a bubble collides with the edge. This method is simply used as a decision maker (line 126) to determine whether the game should end. If the answer is *yes*, the flag of the main game loop is set; if the answer is *no*, the remaining game logic is executed as usual. The two nested `for`-loops starting at line 171 iterate over the bubble group twice and avoid two problems:

sprites()

- A bubble must not be compared with itself. Therefore, the index of the inner loop always starts one position after the current index of the outer loop, and the outer loop index ends before the last element of the bubble group.
- If bubble 1 has already been compared with bubble 2, bubble 2 should not be compared again with bubble 1. This is also achieved by the shifted index.

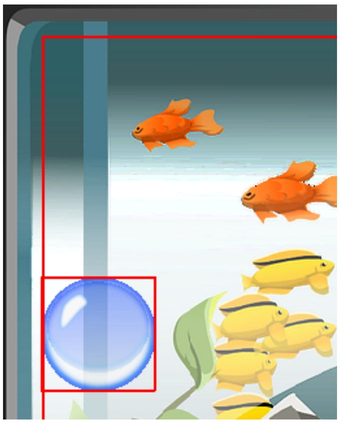

Figure 4.11: Bubbles – collision with the edge

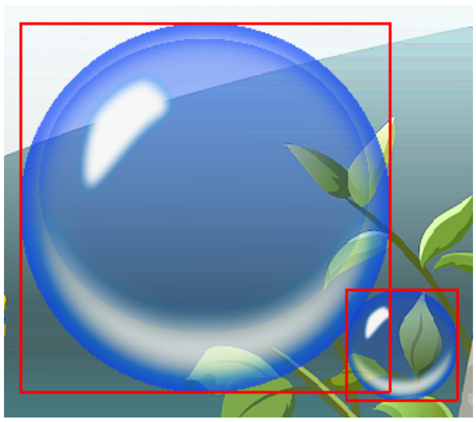

Figure 4.12: Bubbles – bubble collision

In line 176, requirement 8.1 is checked. For this purpose, circle-based collision detection using `collide_circle()` is applied. In line 178 and line 180, requirement 8.2 is implemented. This makes use of the fact that the playing area is a rectangle and that the ontains()

sprite itself also has a rectangular shape. The method `pygame.Rect.contains()` checks whether one rectangle is completely contained within another. If this is not the case – meaning the bubble leaves the playing area – a collision is detected.

Listing 4.71: Bubbles (requirement 8) – `Game.check_bubblecollision()`

```python
    def check_bubblecollision(self) -> bool:
        for index1 in range(0, len(self.all_sprites) - 1):  # Check bubbles
            for index2 in range(index1 + 1, len(self.all_sprites)):
                bubble1 = self.all_sprites.sprites()[index1]
                bubble2 = self.all_sprites.sprites()[index2]
                if type(bubble1).__name__ == "Bubble" and type(bubble2).__name__ == "Bubble":
                    if pygame.sprite.collide_circle(bubble1, bubble2):  # Bubbles?
                        return True
                    if not cfg.PLAYGROUND.contains(bubble1):  # Bubble1 touches border
                        return True
                    if not cfg.PLAYGROUND.contains(bubble2):  # Bubble2 touches border
                        return True
        return False
```

In figure 4.11 on the facing page, the collision of a bubble with the edge is shown. To make this easier to see, helper lines have been drawn. You can clearly see that the rectangle of the bubble is no longer contained within the rectangle of the playing area. figure 4.12 on the preceding page shows the collision of two bubbles. Here as well, helper lines are drawn. These helper lines are displayed when you remove the three comment characters in `Game.draw()`.

Listing 4.72: Bubbles (requirement 8) – helper lines in `Game`

```python
    def draw(self) -> None:
        self.background.draw(self.screen)
        self.all_sprites.draw(self.screen)
        # pygame.draw.rect(self._screen, "red", Settings.PLAYGROUND, 2)
        # for b in self._all_sprites:
        #     pygame.draw.rect(self._screen, "red", b.rect, 2)
        self.window.flip()
```

4.2.9 Requirement 9: Time-based adjustments

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/02%20Bubbles/v09

Requirement 9 Time-based adjustments

The bubbles should grow faster over time.

Since the bubbles are supposed to grow faster over time, I want to pass the growth speed to them as a constructor parameter. In line 54, this parameter is stored in an attribute.

Listing 4.73: Bubbles (requirement 9) – `Bubble`

```
46 class Bubble(pygame.sprite.Sprite):
47     def __init__(self, bubble_container: BubbleContainer, speed: int) -> None:
48         super().__init__()
49         self.bubble_container = bubble_container
50         self.radius = cfg.RADIUS["min"]
51         self.image = self.bubble_container.get(self.radius)
52         self.rect: pygame.Rect = self.image.get_rect()
53         self.fradius = float(self.radius)
54         self.speed = speed                          # Rate of growing
```

Timer

These are all the required changes in `Bubble`; everything else happens in `Game`. In line 103, a timer is created that emits a signal every 1000 ms. Below that, the initial growth speed of the bubbles is set to 10 px.

Listing 4.74: Bubbles (requirement 9) – adjustment of the `Game` constructor

```
101         self.bubble_container = BubbleContainer()
102         self.timer_bubble = Timer(500, False)
103         self.timer_bubble_speed = Timer(1000, False)  #
104         self.bubble_speed = 10
105         self.all_sprites = pygame.sprite.Group()
```

In `spawn_bubble()`, the timer is checked and, if necessary, the bubble growth speed is increased (line 134). The maximum growth speed is limited to 100 px/s; anything faster does not seem playable to me. Each timer signal increases the speed by 5 px/s. This is done in this method so that the new speed is available for newly created bubbles.

Listing 4.75: Bubbles (requirement 9) – `Game.spawn_bubble()`

```
133     def spawn_bubble(self) -> None:
134         if self.timer_bubble_speed.is_next_stop_reached():  #
135             if self.bubble_speed < 100:
136                 self.bubble_speed += 5
137         if self.timer_bubble.is_next_stop_reached():
138             if len(self.all_sprites) <= cfg.MAX_BUBBLES:
139                 b = Bubble(self.bubble_container, self.bubble_speed)
140                 for _ in range(100):
141                     b.randompos()
142                     b.radius += cfg.DISTANCE
143                     collided = pygame.sprite.spritecollide(b, self.all_sprites, False,
                            pygame.sprite.collide_circle)
144                     b.radius -= cfg.DISTANCE
145                     if not collided:
146                         self.all_sprites.add(b)
147                         break
```

If you now try out the game, you will notice an easy start and a moderate increase in game difficulty.

4.2.10 Requirement 10: Display collision

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/02%20Bubbles/v10

Requirement 10 Display collision

If bubbles collide with the edge or with each other, they should change color and remain visible for 5 s before the application terminates.

So far, the game ends so quickly that I cannot really check whether I actually lost for a valid reason or whether the program is misbehaving. With this requirement, I want to be able to see the two colliding bubbles, or the bubble that touches the edge, in a different color. For this purpose, I colored the bubble red (see figure 4.13).

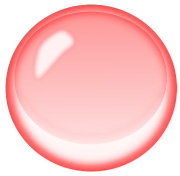

Fig. 4.13: Bubble 2

To achieve this, a second `BubbleContainer` with scaled red bubbles is required. To make access easier, these containers are stored in `Game` as a static dictionary.

In line 102, such a dictionary is created. Under a key, I can now store arbitrary `BubbleContainer` objects.

Listing 4.76: Bubbles (requirement 10) – extension of `Game`

```
101 class Game:
102     BUBBLE_CONTAINER: Dict[str, BubbleContainer] = {}  # Now mor than one
```

The constructor of `BubbleContainer` now receives a filename as a parameter, allowing different graphics to be used as a basis.

Listing 4.77: Bubbles (requirement 10) – change to the constructor of `BubbleContainer`

```
class BubbleContainer:
    def __init__(self, filename: str) -> None:      # Now with filename
        imagename = cfg.get_image(filename)
        image: pygame.Surface = pygame.image.load(imagename).convert_alpha()
        self.images = {i: pygame.transform.scale(image, (i * 2, i * 2)) for i in
            range(cfg.RADIUS["min"], cfg.RADIUS["max"] + 1)}
```

The constructor of `Game` now populates the static dictionary `BUBBLE_CONTAINER` (line 109 and line 110).

Listing 4.78: Bubbles (requirement 10) – change to the constructor of `Game`

```
        self.clock = pygame.time.Clock()
        Game.BUBBLE_CONTAINER["blue"] = BubbleContainer("bubble1.png")   # blue
        Game.BUBBLE_CONTAINER["red"] = BubbleContainer("bubble2.png")    # red
        self.background = pygame.sprite.GroupSingle(Background())
```

Several changes are now required in `Bubble`. The new attribute `mode` (line 49) determines the color of the bubble. Whenever an image is loaded from the `BubbleContainer`, this attribute controls which of the two `BubbleContainer` instances is used as the data source. As an example, line 63 in `update()` can be mentioned here.

Listing 4.79: Bubbles (requirement 10) – constructor of `Bubble` and `update()`

```
46 class Bubble(pygame.sprite.Sprite):
47     def __init__(self, speed: int) -> None:
48         super().__init__()
49         self.mode = "blue"                          # Color mode
50         self.radius = cfg.RADIUS["min"]
51         self.image = Game.BUBBLE_CONTAINER[self.mode].get(self.radius)
52         self.rect: pygame.Rect = self.image.get_rect()
53         self.fradius = float(self.radius)
54         self.speed = speed
55 
56     def update(self, *args: Any, **kwargs: Any) -> None:
57         if "action" in kwargs.keys():
58             if kwargs["action"] == "grow":
59                 self.fradius += self.speed * cfg.DELTATIME
60                 self.fradius = min(self.fradius, cfg.RADIUS["max"])
61                 self.radius = round(self.fradius)
62                 center = self.rect.center
63                 self.image = Game.BUBBLE_CONTAINER[self.mode].get(self.radius)  #
64                 self.rect = self.image.get_rect()
65                 self.rect.center = center
66             elif kwargs["action"] == "sting":
67                 self.stung()
68         elif "mode" in kwargs.keys():
69             self.set_mode(kwargs["mode"])
```

If the mode changes, the alternative color has to be reloaded. This is handled by the `set_mode()` method in `Bubble`.

Listing 4.80: Bubbles (requirement 10) – `Bubble.set_mode()`

```
    def set_mode(self, mode: str) -> None:
        if mode != self.mode:
            self.mode = mode
            self.image = Game.BUBBLE_CONTAINER[self.mode].get(self.radius)
```

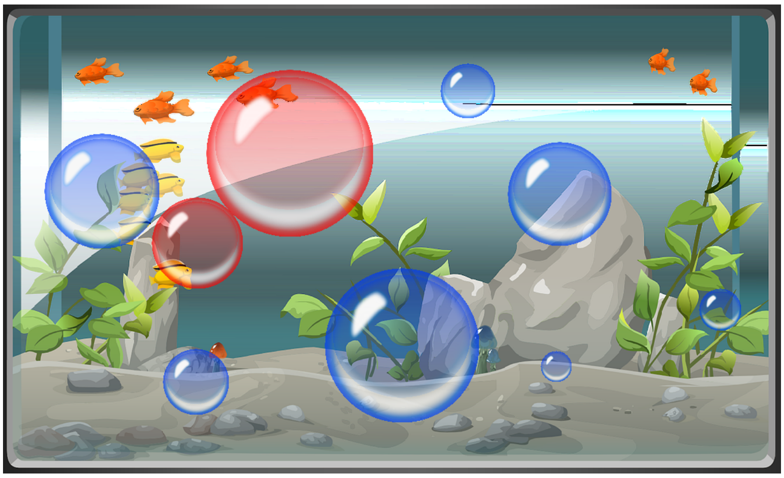

Fig. 4.14: Bubbles – displaying a collision

Now, in the case of a collision – that is, when the game ends – the mode simply needs to be changed.

In figure 4.14, you can see how the two colliding bubbles appear in red. An example of how this is implemented can be found in line 190. In a nested `for`-loop, all sprites are iterated over.

A naive approach would cause problems if both loops compared all sprites from start to finish. First, because due to symmetry you would check $B = A$ even though you have already checked $A = B$. Second, because you would end up comparing each sprite with itself, which would lead to a false collision detection.

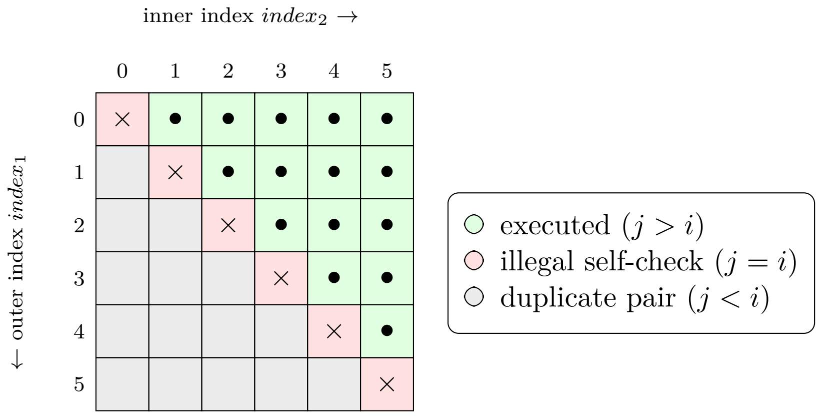

Figure 4.15: Why does the inner loop start at $index_1 + 1$?

In figure 4.15, the algorithm used is illustrated. For each outer index `i=`$index_1$, the inner loop runs `j=`$index_2$ from $i + 1$ to $N - 1$: `for` $index_2$ `in range(`$index_1 + 1$`, N)`

- Starting at $i + 1$ skips the diagonal ($j = i$), so a bubble is never compared with itself.
- It also skips everything left of the diagonal ($j < i$), which would repeat comparisons (e.g. $(1, 2)$ and later $(2, 1)$).
- The outer loop stops at $N - 2$ (i.e. `range(0, N-1)`), because when $i = N - 1$ there is no valid $j > i$ left to compare.

After that, it is checked whether both sprites are of type `Bubble`. Only then is a collision check performed — before that, it would not be worthwhile. If a collision is detected, both bubbles are colored red and the function is exited.

But what if the two bubbles do not touch each other? In that case, there is no need to check whether either of them touches the edge. If one of them does, that bubble is also colored red and the function is exited.

Listing 4.81: Bubbles (requirement 10) – `Game.check_bubblecollision()`

```
    def check_bubblecollision(self) -> bool:
        for index1 in range(0, len(self.all_sprites) - 1):
            for index2 in range(index1 + 1, len(self.all_sprites)):
                bubble1 = self.all_sprites.sprites()[index1]
                bubble2 = self.all_sprites.sprites()[index2]
                if type(bubble1).__name__ == "Bubble" and type(bubble2).__name__ == "Bubble":
                    if pygame.sprite.collide_circle(bubble1, bubble2):
                        bubble1.update(mode="red")  #
                        bubble2.update(mode="red")
                        return True
                    if not cfg.PLAYGROUND.contains(bubble1):
                        bubble1.update(mode="red")
                        return True
                    if not cfg.PLAYGROUND.contains(bubble2):
                        bubble2.update(mode="red")
                        return True
        return False
```

wait()

To give me enough time to see the collision, I want to wait for 2 s at the end. The method `pygame.time.wait()` pauses the application for the specified duration (line 212).

Listing 4.82: Bubbles (requirement 10) – waiting time in `run()`

```
201     def run(self) -> None:
202         time_previous = time()
203         self.running = True
204         while self.running:
205             self.watch_for_events()
206             self.update()
207             self.draw()
208             self.clock.tick(cfg.FPS)
209             time_current = time()
210             cfg.DELTATIME = time_current - time_previous
211             time_previous = time_current
212         pygame.time.wait(2000)                   # Wait a moment
213         pygame.quit()
```

4.2.11 Requirement 11: Pause

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/02%20Bubbles/v11

Requirement 11 Pause

The game enters or leaves pause mode by pressing the right mouse button or p*. The current game state is frozen and displayed in a* grayed-out *form.*

The idea behind this requirement is that a necessary interruption should not automatically mean that the player loses the game. In figure 4.16, you can see what the pause screen should look like.

Figure 4.16: Bubbles – pause screen

In the constructor of `Game`, the flag `pausing` is defined. It later controls whether the game is currently in pause mode or not.

Listing 4.83: Bubbles (requirement 11) – constructor in `Game`

```
        self.pausing = False                        #
```

In `watch_for_events()`, it is now checked whether the `P` key (line 135) or the right mouse button (line 140) has been pressed. In both cases, the new `setpause()` method is called.

Listing 4.84: Bubbles (requirement 11) – `Game.watch_for_events()`

```
128     def watch_for_events(self) -> None:
129         for event in pygame.event.get():
130             if event.type == pygame.QUIT:
131                 self.running = False
132             elif event.type == pygame.KEYDOWN:
133                 if event.key == pygame.K_ESCAPE:
134                     self.running = False
135                 elif event.key == pygame.K_p:       #
136                     self.setpause()
137             elif event.type == pygame.MOUSEBUTTONDOWN:
138                 if event.button == 1:               # left
139                     self.sting(pygame.mouse.get_pos())
140                 elif event.button == 3:               # right
141                     self.setpause()
```

Für die Darstellung der Pause, habe ich die – vielleicht etwas überflüssige – Klasse `Pause` implementiert.

Listing 4.85: Bubbles (requirement 11) – Class `Pause`

```
class Pause(pygame.sprite.Sprite):
    def __init__(self) -> None:
        super().__init__()
        imagename = cfg.get_image("pause.png")
        self.image: pygame.Surface = pygame.image.load(imagename).convert_alpha()
        self.rect = self.image.get_rect()
```

In the constructor of `Game`, an object of the `Pause` class is created so that it can be used in `draw()`.

Listing 4.86: Bubbles (requirement 11) – constructor in `Game`

```
        self.msgpause = Pause()
```

However, the `setpause()` method still needs to be explained. This method either adds the `Pause` object to the list of sprites or removes it again, depending on whether the game is currently in pause mode or not. Afterwards, the boolean value of the flag is negated (toggling).

Listing 4.87: Bubbles (requirement 11) – `Game.setpause()`

```
    def setpause(self):
        if not self.pausing:
            self.all_sprites.add(self.msgpause)
```

```
        else:
            self.msgpause.kill()
        self.pausing = not self.pausing
```

Nothing more is required, since the rest is handled by the usual `update()` and `draw()` mechanisms.

4.2.12 Requirement 12: Restart

So ... where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/02%20Bubbles/v12

Requirement 12 Restart

At the end of the game, the player should be asked whether they want to restart the game or not.

flag

The basic idea of the implementation is to define the state of the game using two flags. As with the pause feature, we need a flag that controls whether the semi-transparent foreground is placed over the game (`restarting`). This is always the case when the bubble collision check detects a collision.

The other flag – `do_start` – indicates whether the player wants to restart. At the relevant points in `update()` and `draw()`, these flags are then evaluated.

The task of displaying a confirmation dialog in the foreground is essentially already solved by the `Pause` class. I can therefore generalize this class by renaming it to `Message` and passing the filename to the constructor as a string parameter (line 46).

Listing 4.88: Bubbles (requirement 12) – from `Pause` to `Message`

```
45 class Message(pygame.sprite.Sprite):
46     def __init__(self, filename: str) -> None:      #
47         super().__init__()
48         imagename = cfg.get_image(filename)
49         self.image: pygame.Surface = pygame.image.load(imagename).convert_alpha()
50         self.rect = self.image.get_rect()
```

In `Game`, starting at line 122, the adjustments required for restarting the game are implemented in the constructor. Essentially, two `Message` objects are created for pause and restart, and all attributes that need to be reset on a start or restart are handled in the new `restart()` method.

Figure 4.17: Bubbles – restart screen

Listing 4.89: Bubbles (requirement 12) – restructuring of the `Game` constructor

```
        self.pausing = False
        self.msg_pause = Message("pause.png")       #
        self.msg_restart = Message("restart.png")
        self.restart()
```

The score is reset, the sprite group containing the bubbles is cleared, the timers are reinitialized, the bubble growth speed is reset to its initial value, and the two flags described above are set to `False`.

Listing 4.90: Bubbles (requirement 12) – `Game.restart()`

```
    def restart(self):
        cfg.POINTS = 0
        self.all_sprites.empty()
        self.all_sprites.add(Points())
        self.bubble_speed = 10
        self.timer_bubble = Timer(500, False)
        self.timer_bubble_speed = Timer(10000, False)
        self.do_start = False
        self.restarting = False
```

The method is called in `update()` when the corresponding flag `do_start` is set. In addition, `update()` inserts the restart screen into the sprite group and sets the flag `restarting` to `True` when a collision is detected.

Listing 4.91: Bubbles (requirement 12) – `Game.update()`

```
    def update(self) -> None:
        if self.do_start:                              # Restart?
            self.restart()
        if not self.pausing and self.running:
            if self.check_bubblecollision():
                if not self.restarting:
                    self.all_sprites.add(self.msg_restart)
                    self.restarting = True
            else:
                self.all_sprites.update(action="grow")
                self.spawn_bubble()
            self.set_mousecursor()
```

The response to the restart screen is queried in `watch_for_events()` and translated into the corresponding flag values. If the player responds with `y` (line 135), the game has to be restarted, so `do_start` is set to `True`. If the player presses *N*, the game should end, which is why the flag of the main game loop is set to `False` (line 137).

Listing 4.92: Bubbles (requirement 12) – extension of `watch_for_events()`

```
135                 elif event.key == pygame.K_j:       #
136                     self.do_start = True
137                 elif event.key == pygame.K_n:       #
138                     self.running = False
```

Since we now display a semi-transparent foreground at the end of the game, there is no longer any need for a two-second pause to inspect the colliding bubbles (see figure 4.17 on the preceding page).

4.2.13 Requirement 13: Sound

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/02%20Bubbles/v13

Requirement 13 Sound

1. *The appearance of bubbles is accompanied by a sound.*
2. *Popping a bubble is accompanied by a sound.*
3. *Touching a bubble is accompanied by a sound.*

Finally, a small sound accompaniment is added. Similar to the bubble sprites, I do not want to lose performance by repeatedly loading sound files. Therefore, the sounds are stored in a static dictionary (line 110).

Listing 4.93: Bubbles (requirement 13) – `SOUND_CONTAINER`

```
    BUBBLE_CONTAINER: Dict[str, BubbleContainer] = {}
    SOUND_CONTAINER: Dict[str, pygame.mixer.Sound] = {}  #
```

Sound

In the constructor of `Game`, the dictionary is populated with objects of the `Sound` class. The class used for sound effects is `pygame.mixer.Sound` (see line 119 ff.).

Listing 4.94: Bubbles (requirement 13) – populating `SOUND_CONTAINER`

```
113         pygame.init()
114         self.window = pygame.Window(size=cfg.WINDOW.size, title=cfg.CAPTION)
115         self.screen = self.window.get_surface()
116         self.clock = pygame.time.Clock()
117         Game.BUBBLE_CONTAINER["blue"] = BubbleContainer("bubble1.png")
118         Game.BUBBLE_CONTAINER["red"] = BubbleContainer("bubble2.png")
119         Game.SOUND_CONTAINER["bubble"] = pygame.mixer.Sound(cfg.get_sound("plopp1.mp3"))  #
120         Game.SOUND_CONTAINER["burst"] = pygame.mixer.Sound(cfg.get_sound("burst.mp3"))
121         Game.SOUND_CONTAINER["clash"] = pygame.mixer.Sound(cfg.get_sound("glas.wav"))
122         self.background = pygame.sprite.GroupSingle(Background())
```

play()

Now the sounds only need to be played at the appropriate places using `pygame.mixer-.Sound.play()`. First, the sound that is played when a new bubble appears: in `spawn_-bubble()` in line 200.

Listing 4.95: Bubbles (requirement 13.1) – `spawn_bubble()`

```
198                     if not collided:
199                         self.all_sprites.add(b)
200                         Game.SOUND_CONTAINER["bubble"].play()  #
201                         break
```

Then, when a bubble bursts in `sting()` (line 225):

Listing 4.96: Bubbles (requirement 13.2) – `sting()`

```
222     def sting(self, mousepos: Tuple[int, int]) -> None:
223         for bubble in self.all_sprites:
224             if self.collidepoint(mousepos, bubble):
225                 Game.SOUND_CONTAINER["burst"].play()  #
226                 bubble.update(action="sting")
```

Finally, the sound is played when a collision with other bubbles or with the edge occurs in `update()`. Here, it must also be taken into account whether the game is currently displaying the restart prompt. If the answer is *yes*, the sound must not be played again; otherwise, the touch sound would be played continuously.

Listing 4.97: Bubbles (requirement 13.3) – `update()`

```
    def update(self) -> None:
        if self.do_start:
            self.restart()
        if not self.pausing and self.running:
            if self.check_bubblecollision():
```

```
                if not self.restarting:
                    Game.SOUND_CONTAINER["clash"].play()  #
                    self.all_sprites.add(self.msgrestart)
                    self.restarting = True
            else:
                self.all_sprites.update(action="grow")
                self.spawn_bubble()
            self.set_mousecursor()
```

And that's it :-)

4.2.14 Or maybe not?

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/02%20Bubbles/v14

Pause, restart, and game over are currently still implemented rather sloppily; for example, the end of the game is delayed by playing a sound instead of using a properly programmed delay. In one version (see source code 4.82 on page 266), I even froze the entire program for several seconds using `pygame.time.wait()` – brrr :-(

It is far more advantageous to control the game using states. What exactly this means should become clear when looking at the solution.

As a first step, let us think about which states the game can actually have:

- `PLAYING`: I am happily popping bubbles and collecting points.
- `PAUSED`: p has been pressed and the game should stop temporarily, but not terminate.
- `WAITING`: The game has ended, but remains active for a few more seconds, for example to display a farewell message.
- `GAME_OVER`: The game is exited. In our example, little or nothing happens here, but one might still want to close files or sockets.

Let us simply add an enumeration for this:

Listing 4.98: Bubbles game state – `GameState(Enum)`

```
    PLAYING = "playing"
    PAUSED = "paused"
    GAME_OVER = "game_over"
    WAITING = "waiting"
```

Since we now have to manage more than just a single state indicating whether the game is paused or not, the attribute `self.state` is created in the constructor of `Game`. In addition, two attributes are introduced that will later be used to measure whether the game remains in a waiting state for 10 s before it finally terminates.

Listing 4.99: Bubbles game state – Constructor of `Game`

```
        self.msgpause = Message("pause.png")
        self.msgrestart = Message("restart.png")
        self.wait_start_time = None  # Zeitstempel für die Wartezeit
        self.wait_duration = 5000  # 5 Sekunden in Millisekunden
```

A real refactoring is now required for `watch_for_events()`. It no longer only checks for events, but also takes into account the current state of the game.

Consider pressing `p` as an example: if the game state is currently `PLAYING`, the game switches to pause mode by calling `set_pause()`. If the game is in the `PAUSED` state, calling `resume()` switches the game back to running mode.

Listing 4.100: Bubbles game state – `Game.watch_for_events()`

```
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                self.running = False
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_ESCAPE:
                    self.running = False
                elif event.key == pygame.K_p:
                    if self.state == GameState.PLAYING:
                        self.set_pause()
                    elif self.state == GameState.PAUSED:
                        self.set_resume()
                elif event.key == pygame.K_j:
                    if self.state in [GameState.GAME_OVER, GameState.WAITING]:
                        self.restart()
                elif event.key == pygame.K_n:
                    self.set_game_over()
            elif event.type == pygame.MOUSEBUTTONDOWN:
                if event.button == 1:  # left
                    if self.state == GameState.PLAYING:
                        self.sting(pygame.mouse.get_pos())
                elif event.button == 3:  # right
                    if self.state == GameState.PLAYING:
                        self.set_pause()
                    elif self.state == GameState.PAUSED:
                        self.set_resume()
```

Let us now take a closer look at the three helper methods, each of which is tailored to a specific state: `set_pause()`, `set_resume()`, and `set_game_over()`. All three methods consist of two parts: in the first part, the state change itself is performed, and in the second part, an action required by this state change is executed. In some cases, a message screen is added to the sprite group; in others, it is removed again. In the third method, the time at which the 10 seconds waiting period begins is stored in `wait_start_time`.

Listing 4.101: Bubbles game state – `set_pause()`, `set_resume()`, and `set_game_over()` of `Game`

```
    def set_pause(self) -> None:
        self.state = GameState.PAUSED
        self.all_sprites.add(self.msgpause)

    def set_resume(self) -> None:
```

```
        self.state = GameState.PLAYING
        self.msgpause.kill()

    def set_game_over(self) -> None:
        self.state = GameState.WAITING
        self.wait_start_time = pygame.time.get_ticks()
```

Just as the game states are taken into account in `watch_for_events()`, they must also be considered in `update()`.

Listing 4.102: Bubbles game state – `Game.update()`

```
        if self.state == GameState.PLAYING:
            if self.check_bubblecollision():
                if not self.restarting:
                    Game.SOUND_CONTAINER["clash"].play()
                    self.all_sprites.add(self.msgrestart)
                    self.restarting = True
                    self.set_game_over()
            else:
                self.all_sprites.update(action="grow")
                self.spawn_bubble()
            self.set_mousecursor()
        elif self.state == GameState.WAITING:
            self.check_waiting_timeout()
        elif self.state == GameState.GAME_OVER:
                self.running = False
```

What remains is the helper method that checks whether the 10 seconds have elapsed: `check_waiting_timeout()`. In this method, the elapsed time is compared with `wait_duration`. If the time has elapsed, the game state is set to `GAME_OVER`, so that the game can terminate cleanly.

Listing 4.103: Bubbles game state – `Game.check_waiting_timeout()`

```
        if self.wait_start_time is not None:
            elapsed_time = pygame.time.get_ticks() - self.wait_start_time
            if elapsed_time >= self.wait_duration:
                self.state = GameState.GAME_OVER
                self.wait_start_time = None
```

In figure 4.18 on the next page, we can once again visually trace these state transitions.

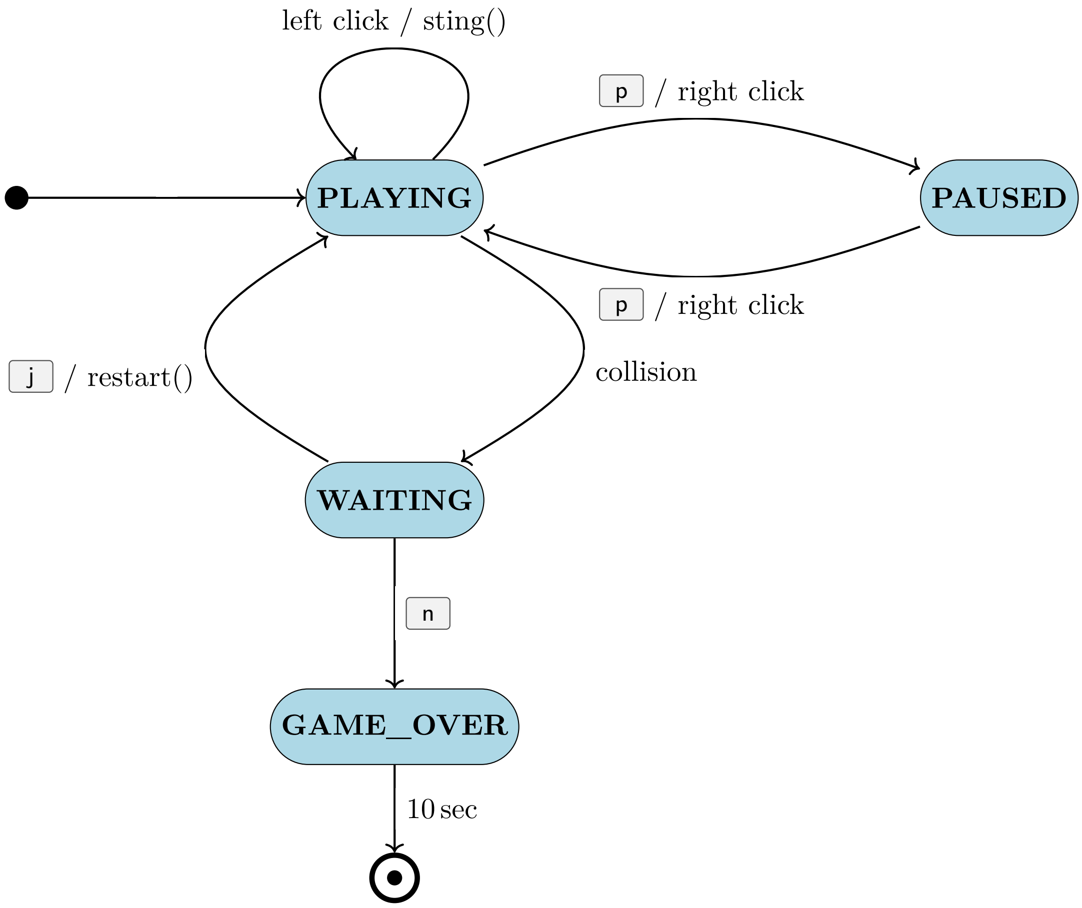

Figure 4.18: Bubbles – state diagram

4.3 Moonlander

In this chapter, we will build a Moon Lander game. For this project, I want to avoid pre-made sprites and create all graphics entirely from drawing primitives (see section 2.2 on page 28).

We will develop this game systematically, step by step, assuming that the techniques from chapter 2 are already familiar. I will omit docstring comments in the source code, since everything is explained in the text and the listings would otherwise become unnecessarily long. In the final version, they are included.

4.3.1 Requirement 1: Standards

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/03%20Moonlander/v01

Requirement 1 Standard functionality

1. *The window has a size of* 600 × 800 px.
2. *The background is divided into a black sky, a blue Earth in the upper-right corner, and the lunar surface.*
3. *The game can be exited using* Esc *or by clicking the red X.*
4. *Pressing* R *triggers a restart.*
5. *The game runs at a speed independent of the frame rate.*

Requirement 1.1 is already defined in the preamble. In addition, `FPS` and the associated `DELTATIME` are defined in `config.py`. The constant `HORIZONT` specifies where the lunar surface ends and the black night sky begins.

Listing 4.104: Moonlander (requirement 1.1) – `config.py`

```python
import pygame

WINDOW = pygame.Rect(0, 0, 600, 800)
FPS = 60
DELTATIME = 1.0 / FPS
HORIZONT = 50
```

I implement requirement 1.2 using three classes: `Sky`, `Moon`, and `Earth`. Let us start with the `Sky` class. It has a fairly simple basic structure. In the constructor, a reference to the window is passed in and the size of the sky is stored as a `Rect` object. Space is left at the bottom for the lunar surface. The `draw()` method then draws a black rectangle at the appropriate position.

Listing 4.105: Moonlander (requirement 1.2) – Class `Sky`

```
class Sky:
    def __init__(self, screen:pygame.Surface) -> None:
        top = 0
        left = 0
        width = cfg.WINDOW.width
        height = cfg.WINDOW.height - cfg.HORIZONT
        self.rect = pygame.Rect(top, left, width, height)
        self.screen = screen

    def draw(self) -> None:
        pygame.draw.rect(self.screen, "black", self.rect)
```

The `Moon` class works in exactly the same way (see source code 4.106). The only differences are the different position and the different color – gray in this case.

Listing 4.106: Moonlander (requirement 1.2) – Class `Moon`

```
class Moon:
    def __init__(self, screen:pygame.Surface) -> None:
        top = 0
        left = cfg.WINDOW.height - cfg.HORIZONT
        width = cfg.WINDOW.width
        height = cfg.HORIZONT
        self.rect = pygame.Rect(top, left, width, height)
        self.screen = screen


    def draw(self) -> None:
        pygame.draw.rect(self.screen, "gray", self.rect)
```

The `Earth` class draws a blue sphere in the upper-right corner of the screen (see source code 4.107).

Listing 4.107: Moonlander (requirement 1.2) – Class `Earth`

```
class Earth:
    def __init__(self, screen:pygame.Surface) -> None:
        self.radius = 80
        left = cfg.WINDOW.right - 2*self.radius - 30
        top = cfg.WINDOW.top + 15
        width = 2*self.radius
        height = 2*self.radius
        self.rect = pygame.Rect(left, top, width, height)
        self.screen = screen


    def draw(self) -> None:
        pygame.draw.circle(self.screen, "blue", self.rect.center, self.radius)
```

As usual, the game is encapsulated in its own class: `Game`. The three objects only need to be integrated into the standard structure of `Game`. This is also where quitting and restarting the game are implemented.

In source code 4.107 on the preceding page, Pygame is initialized in the constructor of `Game`, a window is created, the window's screen surface is obtained, and a `Clock` object is created for the delta-time logic (see section 2.4.3.1 on page 64).

Listing 4.108: Moonlander (requirement 1) – Constructor of `Game`

```python
class Game:
    def __init__(self) -> None:
        pygame.init()
        self.window = pygame.Window(size=cfg.WINDOW.size, title="MyMoonlander",
            position=pygame.WINDOWPOS_CENTERED)
        self.screen = self.window.get_surface()
        self.clock = pygame.time.Clock()
```

The structure of the `run()` method follows the examples shown above. Its core consists of calling the event handler, updating the game objects, and drawing the game objects; in addition, the delta-time logic is applied.

Listing 4.109: Moonlander (requirement 1) – `Game.run()`

```python
    def run(self) -> None:
        self.restart()
        time_previous = time()
        while self.running:
            self.watch_for_events()
            self.update()
            self.draw()
            self.clock.tick(cfg.FPS)
            time_current = time()
            cfg.DELTATIME = time_current - time_previous
            time_previous = time_current
        pygame.quit()
```

The event handler should no longer come as a surprise. Using `QUIT` or Esc ends the game, and pressing r triggers a restart.

Listing 4.110: Moonlander (requirement 1) – `Game.watch_for_events()`

```python
    def watch_for_events(self) -> None:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                self.running = False
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_ESCAPE:
                    self.running = False
                elif event.key == pygame.K_r:
                    self.restart()
```

At the moment, the `update()` method serves only as a placeholder for functionality that will be added later.

Listing 4.111: Moonlander (requirement 1) – `Game.update()`

```python
    def update(self) -> None:
        pass
```

In `draw()`, the drawing methods of the game objects are called, and the window buffer is flipped.

Listing 4.112: Moonlander (requirement 1) – `Game.draw()`

```
    def draw(self) -> None:
        self.background.draw()
        self.moon.draw()
        self.earth.draw()
        self.window.flip()
```

The restart does not reset the state of the individual game objects. Instead, the objects are recreated entirely. This is the simplest way to implement a restart, but it is not suitable for every type of game.

Listing 4.113: Moonlander (requirement 1) – `Game.restart()`

```
    def restart(self) -> None:
        self.background = Sky(self.screen)
        self.moon = Moon(self.screen)
        self.earth = Earth(self.screen)
        self.running = True
```

All that remains is the actual call that starts the game.

Listing 4.114: Moonlander (requirement 1) – `main()`

```
def main():
    Game().run()

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

After starting the program, a scene like the one shown in figure 4.19 appears.

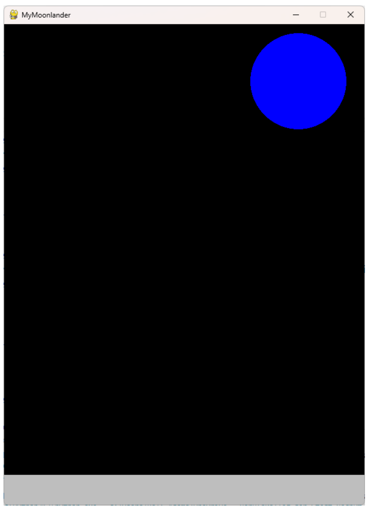

Fig. 4.19: Moonlander (1)

4.3.2 Requirement 2: Lunar surface

So … where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/03%20Moonlander/v02

So far, the lunar surface is just a gray rectangle. However, I want a gray mountainous landscape to enhance the visual appeal.

Requirement 2 Lunar surface

The lunar surface consists of consecutively arranged mountain ranges.

As a first step, I extend the constructor of `Moon` by adding the number of mountain ranges. Each mountain range is initially represented by a gray rectangle. The variations in height will be added later.

The actual lunar surface (landing area in line 25) remains a rectangle with a height of `HORIZONT`. In `self.layers`, the information for each mountain range is stored as a list.

Starting at line 28, the mountain ranges (the default is 5) are created. First, the color of each mountain range is defined (line 29). Starting from a base color value of 180, an amount depending on the layer index is subtracted. The larger the layer index, the more is subtracted from 180. In terms of color, this means that the mountain range becomes darker. The farther away a mountain range (the layer) is, the darker it appears.

The height of a mountain range (`y`) is calculated by moving upward at least 10 px from the upper edge of the landing area. This value is then increased by a random number between 5 and 30, so that the heights of the mountain ranges are not always the same. To ensure that the mountain ranges in the background always stand out nicely, this value is additionally multiplied by the layer index. Finally, `draw()` is adjusted (source code 4.116 on the following page) so that the mountain ranges are drawn as rectangles. The lunar surface should look roughly like the one shown in figure 4.20.

Fig. 4.20: Moonlander (2)

Listing 4.115: Moonlander (requirement 2) – Constructor of `Moon`

```
20 class Moon:
21     def __init__(self, screen: pygame.Surface, layer_count: int=5):
22         self.screen = screen
23         top = cfg.WINDOW.height - cfg.HORIZONT
24         self.rect = pygame.Rect(0, top,
25                                 cfg.WINDOW.width, cfg.HORIZONT)   # Landing area
26
27         self.layers = []
```

```
28         for layer_index in range(layer_count):  # Mountain range
29             mycolor = 180 - layer_index * 20    # Foreground darker, background lighter
30             self.layers.append({"y":self.rect.top - 10 - randint(5, 30)*layer_index,
31                                 "color":(mycolor, mycolor, mycolor)})
```

Listing 4.116: Moonlander (requirement 2) – `Moon.draw()`

```
33     def draw(self):
34         pygame.draw.rect(self.screen, (230, 230, 230), self.rect)
35 
36         for layer in reversed(self.layers):
37             r = pygame.Rect()
38             r.top = layer["y"]
39             r.left = self.rect.left
40             r.width = self.rect.width
41             r.height = self.rect.top - r.top
42             pygame.draw.rect(self.screen, layer["color"], r)
```

Now it is time to add the mountain peaks. The basic idea is to generate random height variations around the upper edge of each mountain range and subtract them from the height of that upper edge.

As a first step, the constructor of `Moon` is extended by the parameter `peaks`. In line 28, the distance between two height variations is calculated and stored in `dist`. This value could also be randomized further, but for some reason I did not feel like doing that here.

Fig. 4.21: Moonlander (3)

Now, within the loop, a peak or a valley is generated for each mountain range. The determination of the color (the shade of gray) remains unchanged. In `lofPeaks`, the peaks are stored as a list of points. The first point is always located at the far left on the upper edge of the landing area (line 33). This point serves as the starting point of our closed polygon.

After that, the list of peaks is extended with additional random points using a loop. In line 35, a height variation between -5 px and 10 px is chosen at random and subtracted from the upper edge. In the following line, the next peak is shifted to the right by adding `dist`. Once this inner `for`-loop has finished, the list of height points is complete and can be added to the corresponding layer in line 38. Before doing so, however, the final point of the polygon chain must still be generated and added.

Listing 4.117: Moonlander (requirement 2) – constructor of `Moon` with peaks

```
20 class Moon:
21     def __init__(self, screen: pygame.Surface, layer_count:int=5, peaks: int=35):
22         self.screen = screen
23         top = cfg.WINDOW.height - cfg.HORIZONT
24         self.rect = pygame.Rect(0, top,
25                                  cfg.WINDOW.width, cfg.HORIZONT) # Landing area
26 
27         self._layers = []
28         dist = self.rect.width // peaks        # Distance between height differences
29         for layer_index in range(layer_count): # Build mountain
30             mycolor = 180 - layer_index * 20   # Foreground darker, background lighter
```

```
            y = self.rect.top - 10 - randint(5, 30)*layer_index # Random starting height
            x = self.rect.left                  # First peak starts at the left
            lofPeaks = [(x, top)]               # The first peak as a point
            for i in range(peaks):              # The other peaks of the layer are generated.
                lofPeaks.append((x, y + randint(-5, 10))) # Random height difference
                x += dist                       # The next peak is further to the right
            lofPeaks.append((self.rect.right, y))   # Last peak is at the right
            lofPeaks.append((self.rect.right, top)) # Base of the mountain range
            self._layers.append({"color":(mycolor, mycolor, mycolor),
                                 "peaks":lofPeaks})
```

The `draw()` method has now become pleasantly simple. For each mountain range, `draw.polygon()` is called; the actual work is done in the constructor. The result can be admired in figure 4.21 on the facing page.

Listing 4.118: Moonlander (requirement 2) – `Moon.draw()` with peaks

```
    def draw(self):
        pygame.draw.rect(self.screen, (230, 230, 230), self.rect)
        for layer in reversed(self._layers):
            pygame.draw.polygon(
                self.screen,
                layer["color"],
                layer["peaks"]
            )
```

Finally, I want to give the mountains a bit more contour. To do this, the single polygon is split into many smaller ones, where each polygon spans from one peak to the next. First, it becomes apparent that the number of peaks now varies for each mountain range (line 29); this makes the ranges appear less like a checkerboard. Of course, the distance between the peaks then also has to be recalculated (line 30).

Fig. 4.22: Moonlander (4)

To make the source code easier to understand, I separated the generation of the peaks from the calculation of the corresponding polygons. Something genuinely new only happens starting at line 41. For each peak, four points are now determined: the starting peak, the peak to its right, the point directly below it down to the surface, and finally a point on the surface back to the left underneath the starting peak. In addition, a subtle shade of gray is chosen at random. These four points are stored as a polygon together with the corresponding color in the list `layers`.

Listing 4.119: Moonlander (requirement 2) – constructor of `Moon` with contour

```
20 class Moon:
21     def __init__(self, screen: pygame.Surface, layer_count:int=5, peaks: int=35):
22         self.screen = screen
23         top = cfg.WINDOW.height - cfg.HORIZONT
24         self.rect = pygame.Rect(0, top,
25                                 cfg.WINDOW.width, cfg.HORIZONT)
26 
27         self._layers = []
28         for layer_index in range(layer_count):
29             mypeaks = randint(peaks//2, peaks)      # Number varies
30             dist = self.rect.width // mypeaks       # Distance between height differences
```

```
            mycolor = 180 - layer_index * 20
            y = self.rect.top - 10 - randint(5, 30)*layer_index
            x = self.rect.left
            lofPeaks = [(x, top)]
            for i in range(mypeaks):
                lofPeaks.append((x, y + randint(-5, 20)))
                x += dist
            lofPeaks.append((self.rect.right, y))
            lofPeaks.append((self.rect.right, top))

            poly = []                                       # A polygon path
            for index in range(len(lofPeaks)-1):
                p1 = lofPeaks[index]
                p2 = lofPeaks[index+1]
                p3 = (lofPeaks[index+1][0], self.rect.top)
                p4 = (lofPeaks[index][0], self.rect.top)
                r = randint(-5,5)
                c =  [mc +  r for mc in (mycolor, mycolor, mycolor)]
                poly.append({"points":(p1, p2, p3, p4), "color":c})
            self._layers.append(poly)
```

After a few more minor and easy-to-understand changes in `draw()`, the Moon is complete (figure 4.22 on the previous page).

Listing 4.120: Moonlander (requirement 2) – `Moon.draw()` with contour

```
    def draw(self):
        pygame.draw.rect(self.screen, (230, 230, 230), self.rect)
        for layer in reversed(self._layers):
            for poly in layer:
                pygame.draw.polygon(
                    self.screen,
                    poly["color"],
                    poly["points"])
```

Redrawing the mountain ranges from scratch every time is certainly a huge waste of computing time. A common technique this is to draw the image once onto a bitmap (`pygame.surface.Surface`) and then simply blit this bitmap each frame.

Note that `rect` is now needed for the entire bitmap and has therefore first been renamed to `landingarea`. It is also no longer necessary to keep `layers` and `landingarea` as attributes of the class, since this information is no longer required after the bitmap has been created.

Listing 4.121: Moonlander (requirement 2) – `Moon` as bitmap

```
class Moon:
    def __init__(self, screen: pygame.Surface, layer_count:int=5, peaks: int=35):
        self.screen = screen
        self.surface = pygame.Surface((cfg.WINDOW.width,
                                        cfg.HORIZONT + layer_count*30),
                                        pygame.SRCALPHA)
        self.rect = self.surface.get_rect()
        self.rect.left = cfg.WINDOW.left
        self.rect.bottom = cfg.WINDOW.bottom
        landingarea = pygame.Rect(0, self.rect.height - cfg.HORIZONT,
                                  cfg.WINDOW.width, cfg.HORIZONT)

        layers = []
```

```python
        for layer_index in range(layer_count):
            mypeaks = randint(peaks//2, peaks)
            dist = landingarea.width // mypeaks
            mycolor = 180 - layer_index * 20
            y = landingarea.top - 10 - randint(5, 30)*layer_index
            x = landingarea.left
            lofPeaks = [(x, landingarea.top)]
            for i in range(mypeaks):
                lofPeaks.append((x, y + randint(-5, 20)))
                x += dist
            lofPeaks.append((landingarea.right, y))
            lofPeaks.append((landingarea.right, landingarea.top))

            poly = []
            for index in range(len(lofPeaks)-1):
                p1 = lofPeaks[index]
                p2 = lofPeaks[index+1]
                p3 = (lofPeaks[index+1][0], landingarea.top)
                p4 = (lofPeaks[index][0], landingarea.top)
                r = randint(-5,5)
                c =  [mc +  r for mc in (mycolor, mycolor, mycolor)]
                poly.append({"points":(p1, p2, p3, p4), "color":c})
            layers.append(poly)

        pygame.draw.rect(self.surface, (230, 230, 230), landingarea)
        for layer in reversed(layers):
            for poly in layer:
                pygame.draw.polygon(
                    self.surface,
                    poly["color"],
                    poly["points"])

    def draw(self):
        self.screen.blit(self.surface, self.rect.topleft)
```

4.3.3 Requirement 3: Earth

So … where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/03%20Moonlander/v02

The Earth as a simple blue spot? That would be far too unattractive!

Requirement 3 Earth

1. *The Earth should have an atmospheric glow.*
2. *Landmasses should be visible on the Earth.*

First, the Earth is also converted into a bitmap in order to improve performance. The procedure is analogous to source code 4.121 on the facing page.

Listing 4.122: Moonlander (requirement 3) – `Earth` as bitmap

```python
class Earth:
    def __init__(self, screen:pygame.Surface) -> None:
        self.radius = 80
        self.surface = pygame.Surface(
            (2*self.radius, 2*self.radius),
            pygame.SRCALPHA)
        self.rect = self.surface.get_rect()
        self.rect.left = cfg.WINDOW.right - 200
        self.rect.top = cfg.WINDOW.top + 50
        self.screen = screen

        pygame.draw.circle(self.surface,
                           (30, 144, 255),
                           (self.radius, self.radius),
                           self.radius)

    def draw(self) -> None:
        self.screen.blit(self.surface, self.rect.topleft)
```

Next, the atmospheric glow is created. The basic idea is to draw circles from the inside to the outside with increasing transparency. To achieve this, a loop counts down from 20 to 1. This counter is multiplied by 10 in line 81 and subtracted from 210, resulting in a sequence like $(10, 20, \ldots, 200)$. Correspondingly, the radius of these circles increases steadily in line 83. Finally, the slightly reduced Earth itself is drawn (see figure 4.23 on the next page.

Listing 4.123: Moonlander (requirement 3.1) – `Earth` with atmospheric glow

```python
79          for a in range(20, 1, -1):
80              pygame.draw.circle(self.surface,
81                                 (135, 206, 250, 210-a*10),  # Steigende Transparanz
82                                 (self.radius, self.radius),
83                                 self.radius-20+a)           # Wachsender Radius
84          pygame.draw.circle(self.surface, (30, 144, 255, 255),
85                             (self.radius, self.radius), self.radius-20)
```

I had the polygon data for the landmasses generated by ChatGPT (what a blessing!). For the sake of readability, this data has been moved to an external file (`continent_polygons.py`). It consists of a list of lists of points. The inner lists represent the landmasses as closed polygon paths. First, the polygon data is imported as a module:

Listing 4.124: Moonlander (requirement 3.2) – importing the polygon data

```python
from continent_polygons import continent_polygons
```

The actual drawing is then fairly straightforward. The coordinates only need to be aligned with the center of the Earth and scaled to half the size so that they fit inside the circle (see figure 4.24 on the facing page).

Listing 4.125: Moonlander (requirement 3.2) – `Earth` with continents

```
        for continent in continent_polygons:
            poly = [(self.radius + (0.5*x), self.radius + (0.5*y)) for (x, y) in continent]
            pygame.draw.polygon(self.surface, (181, 150, 116), poly)
```

That will be enough for the Earth. On to the next effect.

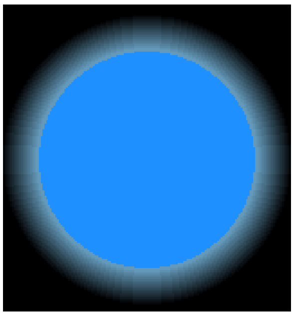

Figure 4.23: Moonlander (5) glowing

Figure 4.24: Moonlander (6) continent

4.3.4 Requirement 4: Stars

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/03%20Moonlander/v02

Outer space is neither black nor empty.

Requirement 4 Stars

1. *Stars of different sizes should be visible in the background.*
2. *The stars should change in brightness and size. This should create a kind of twinkling effect.*

First, the constructor of `Sky` is extended by a parameter specifying the number of stars; the default value is 200 stars. In line 18, a list for the stars is created. Afterwards, the loop fills this list with the corresponding number of entries. Position, size, and color are determined randomly.

Listing 4.126: Moonlander (requirement 4.1) – Constructor of `Sky`

```
9  class Sky:
10     def __init__(self, screen:pygame.Surface, star_count: int=200) -> None:
11         top = 0
```

```
12        left = 0
13        width = cfg.WINDOW.width
14        height = cfg.WINDOW.height - cfg.HORIZONT
15        self.rect = pygame.Rect(top, left, width, height)
16        self.screen = screen
17
18        self.stars = []                    # Sternenliste
19        for _ in range(star_count):
20            self.stars.append({"pos":(randint(2, self.rect.right-1),
21                                      randint(2, self.rect.right-1)),
22                               "size":randint(1, 3),
23                               "color":randint(10, 255)})
```

In `draw()`, the entries of the list are used to render the stars.

Listing 4.127: Moonlander (requirement 4.1) – `Sky.draw()`

```
25    def draw(self) -> None:
26        pygame.draw.rect(self.screen, "black", self.rect)
27        for star in self.stars:
28            pygame.draw.circle(self.screen, (255,255,star["color"]), star["pos"],
                  star["size"])
```

Creating the twinkling effect is a bit more interesting. As preparation, each star is assigned a random value in line 23 that specifies after how many frames a change in brightness should occur. At 60 fps, this corresponds to roughly 3.3 to 10 s.

Listing 4.128: Moonlander (requirement 4.2) – twinkling stars (1)

```
18        self.stars = []
19        for _ in range(star_count):
20            self.stars.append({"pos":(randint(2, self.rect.right-1),
21                                      randint(2, self.rect.right-1)),
22                          "size":randint(1, 3),
23                          "duration": randint(200, 600), # Time in frames
24                          "counter":0,
25                          "color":randint(10, 255)})
```

Since the state of the game object now changes over time, the `update()` method is required. This method recalculates the brightness and size of the stars. In line 29, a counter is increased by 1 in every frame (that is, on each call). The value is then processed using the modulo operator. In this way, the counter always stays within a fixed range, preventing an overflow — that is, exceeding the valid range of an integer.

Within the loop, all stars are now processed. If the value of `counter` modulo `duration` is 0, exactly `duration` frames have passed and the color and size must be updated. The `draw()` method remains unchanged.

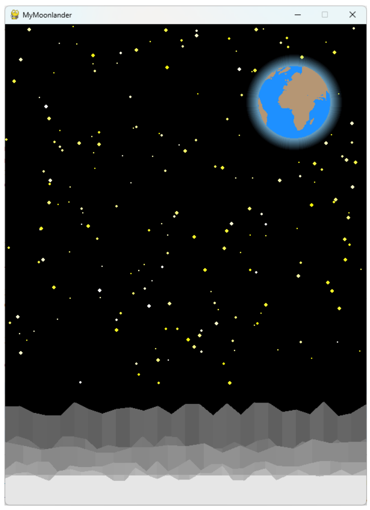

Fig. 4.25: Moonlander (7)

Listing 4.129: Moonlander (requirement 4.2) – twinkling stars (2)

```
    def update(self) -> None:
        for star in self.stars:
            star["counter"] = (star["counter"] + 1) % (star["duration"] + 1) # Counter
            if star["counter"] == 0:
                star["color"] = (star["color"] + randint(0, 70)) % 256
                star["size"] = (star["size"] + 1) % 4
```

Finally, the previously unused `update()` method in `Game` is extended with the corresponding method call, and everything should work as expected (see figure 4.25 on the preceding page).

Listing 4.130: Moonlander (requirement 4.2) – twinkling stars (3) `Game.update()`

```
    def update(self) -> None:
        self.background.update()
```

4.3.5 Requirement 5: Lander

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/03%20Moonlander/v03

Requirement 5 Lander

1. *The lander consists of an antenna, a crew module, a base with connectors to the crew module, and landing legs with pads.*
2. *Pressing the* Space *key displays a thrust exhaust.*
3. *The lander starts roughly in the middle and fairly high up, but not directly at the top edge.*

The first thing to notice is that I do not use just a single `Surface` object, but two. The idea behind this is to create one sprite for the lander with thrust and one without thrust. In `draw()`, the attribute `thrusting` (line 126) is then used to control which of the two sprites is blitted to the screen. For the sake of clarity, the drawing of the lander is encapsulated in the method `create_lander()` (line 125).

Listing 4.131: Moonlander (requirement 5.1) – Constructor of `Lander`

```
117 class Lander:
118     def __init__(self, screen: pygame.Surface) -> None:
119         self.screen = screen
120         self.surface = pygame.Surface((90,81), pygame.SRCALPHA)
```

```
121         self.surface_thrusting = pygame.Surface((90,81), pygame.SRCALPHA)
122         self.rect = self.surface.get_frect()
123         self.rect.centerx = cfg.WINDOW.centerx      # horizontal start position
124         self.rect.top = self.rect.height            # vertical start position
125         self.create_lander()                        # Drawing is delegated
126         self.thrusting = False                      # Flag whether acceleration is active
```

Explaining each individual drawing step would certainly be somewhat tedious and would not provide much additional learning value. The easiest way to understand the source code is to change individual details and observe the effect. Nevertheless, I would like to address one specific aspect.

In the first step, all drawing operations are performed on the surface `surface`. This results in a lander without thrust. Starting at line 126, the surface with thrust is created. For this purpose, the lander without thrust is first copied onto `surface_thrusting` using `blit()`. After that, an additional thrust flame is drawn onto `surface_thrusting`. As a result, two `Surface` objects are available for rendering the lander. Both can be seen in figure 4.26 and figure 4.27 on the facing page.

Listing 4.132: Moonlander (requirement 5.1) – `Lander.create_lander()`

```
    def create_lander(self) -> None:
        # A few abbreviations
        cx = self.rect.width // 2
        cy = self.rect.height //2
        s = self.rect.width // 2
        sur = self.surface

        # Antenna
        pygame.draw.line(sur, (220, 220, 220), (cx, cy - s//2), (cx, cy - s//1.2), 2)
        pygame.draw.circle(sur, (255, 255, 255), (cx, cy - s//1.2), 3)

        # Upper crew module (narrower)
        pygame.draw.polygon(
            sur, (160, 160, 160),
            [(cx - s//4, cy - s//2),
             (cx - s//6, cy - s//3),
             (cx + s//6, cy - s//3),
             (cx + s//4, cy - s//2)]
        )

        # Connector between base and crew module
        conn_color = (160, 160, 160)
        pygame.draw.line(sur, conn_color, (cx - s//3, cy), (cx - s//6, cy - s//3), 2)
        pygame.draw.line(sur, conn_color, (cx, cy), (cx, cy - s//3), 2)
        pygame.draw.line(sur, conn_color, (cx + s//3, cy), (cx + s//6, cy - s//3), 2)

        # Module base (central capsule, light gray)
        pygame.draw.polygon(
            sur, (200, 200, 200),
            [(cx - s//3, cy),
             (cx - s//2, cy + s//2),
             (cx + s//2, cy + s//2),
             (cx + s//3, cy)]
        )

        # Windows in module
        r = 5
        window_color = (50, 50, 50)
        pygame.draw.circle(sur, window_color, (cx, cy+(s//4)), r)
```

```python
        pygame.draw.circle(sur, window_color, (cx-(s//4), cy+(s//4)), r)
        pygame.draw.circle(sur, window_color, (cx+(s//4), cy+(s//4)), r)

        # Landing legs
        leg_color = (100, 100, 100)
        pygame.draw.line(sur, leg_color, (cx - s//2, cy + s//2), (cx - s, cy + s), 3)
        pygame.draw.line(sur, leg_color, (cx + s//2, cy + s//2), (cx + s, cy + s), 3)
        pygame.draw.line(sur, leg_color, (cx - s//4, cy + s//2), (cx - s//3, cy + s), 3)
        pygame.draw.line(sur, leg_color, (cx + s//4, cy + s//2), (cx + s//3, cy + s), 3)

        # Feet
        feet_color = (150, 150, 150)
        pygame.draw.circle(sur, feet_color, (cx - s + (r+2), cy + s - (r+2)), r-1)
        pygame.draw.circle(sur, feet_color, (cx + s - (r+2), cy + s - (r+2)), r-1)
        pygame.draw.circle(sur, feet_color, (cx - s//3 + (r-4), cy + s - (r+2)), r-1)
        pygame.draw.circle(sur, feet_color, (cx + s//3 - (r-4), cy + s - (r+2)), r-1)

        # Thruster exhaust
        self.surface_thrusting.blit(sur, (0,0))
        pygame.draw.polygon(self.surface_thrusting, (255, 140, 0), [
            (cx - 5, cy + s//2),
            (cx + 5, cy + s//2),
            (cx, cy + s//2 + 20)
        ])
```

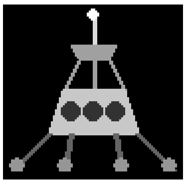

Figure 4.26: Moonlander (8) – lander without thrust

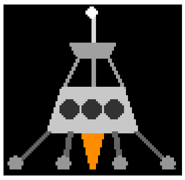

Figure 4.27: Moonlander (9) – lander with thrust

In `update()`, the `thrusting` flag is controlled and set from outside the class.

Listing 4.133: Moonlander (requirement 5.1) – `Lander.update()`

```python
    def update(self, *args: Any, **kwargs: Any) -> None:
        if "action" in kwargs.keys():
            if kwargs["action"] == "thrust":
                self.thrusting = True
            elif kwargs["action"] == "unthrust":
                self.thrusting = False
```

In `Game`, `watch_for_events()` must be adapted to handle the thrust control. Pressing Space activates the lander's *thrust* mode, and releasing the key deactivates it again.

Listing 4.134: Moonlander (requirement 5.2) – `Game.watch_for_events()`

```python
    def watch_for_events(self) -> None:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                self.running = False
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_ESCAPE:
```

```
                    self.running = False
                elif event.key == pygame.K_r:
                    self.restart()
                elif event.key == pygame.K_SPACE:
                    self.lander.update(action="thrust")
            elif event.type == pygame.KEYUP:
                if event.key == pygame.K_SPACE:
                    self.lander.update(action="unthrust")
```

In `Lander.draw()`, the output switches between the two surfaces depending on the current *thrust* mode.

Listing 4.135: Moonlander (requirement 5.2) – `Lander.draw()`

```
    def draw(self) -> None:
        if self.thrusting:
            self.screen.blit(self.surface_thrusting, self.rect.topleft)
        else:
            self.screen.blit(self.surface, self.rect.topleft)
```

All that remains is to define the starting position, which is fairly simple. In the constructor of `Lander`, the appropriate position is determined (see line 123 and line 124). The result should look like the one shown in figure 4.28.

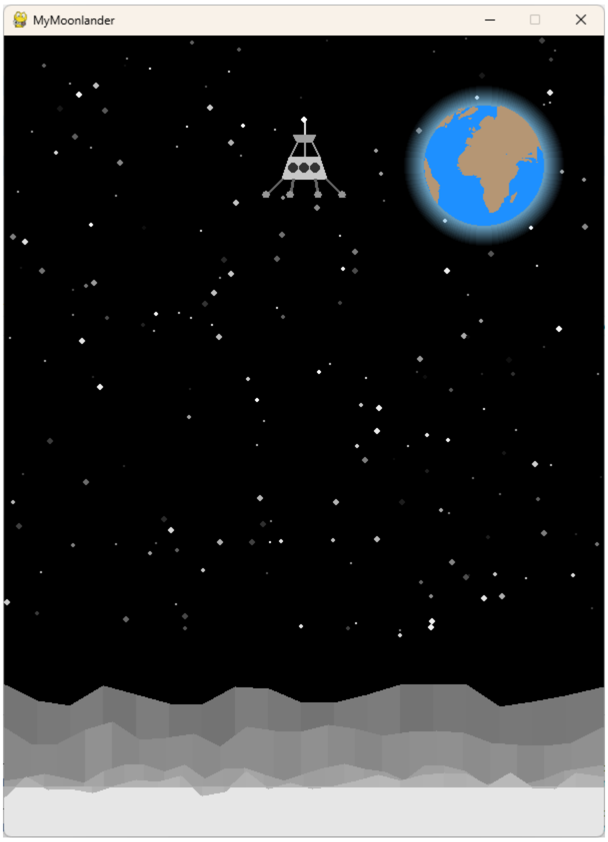

Figure 4.28: Moonlander (10) – the lander

4.3.6 Requirement 6: Gravitation and landing

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/03%20Moonlander/v04

Requirement 6 Gravitation and landing

1. *The lander is accelerated by the Moon's gravity at* $1.62\,\mathrm{m/s^2}$.
2. *When the pads of the landing legs touch the lunar surface, the lander comes to a stop.*

For this purpose, several parameters are defined in `config.py`, starting at line 7. I included both lunar and Earth gravity. Of course, you are completely free to let the lander touch down on Venus or even Jupiter instead.

Listing 4.136: Moonlander (requirement 6.1) – physical constants

```
3  WINDOW = pygame.Rect(0, 0, 600, 800)
4  FPS = 60
5  DELTATIME = 1.0 / FPS
6  HORIZONT = 50
7  # Physical constants (Moon conditions)
8  MOON_GRAVITY = 1.62                          # m/s²
9  EARTH_GRAVITY = 9.81                         # m/s²
10 PIXELS_PER_METER = 10                        # Scaling 1m = 10px
11 GRAVITY = MOON_GRAVITY * PIXELS_PER_METER    # = 16.2 px/s²
```

In the constructor of the lander, its vertical velocity is defined in line **??**. At the start of the game, this value is always set to 0. This is admittedly unrealistic, since the lander is already in the middle of its descent – but let's not worry about that for now.

Listing 4.137: Moonlander (requirement 6.1) – extension of the `Lander` constructor

```
        self.velocity = 0
```

The `update()` method of `Lander` is extended by adding the action `move`. The actual calculation of the new position is encapsulated in the method `move()`.

Listing 4.138: Moonlander (requirement 6.1) – extension of `Lander.update()`

```
            elif kwargs["action"] == "move":
                self.move()
```

First, the new velocity is calculated based on gravity. After that, the change in position is computed using this velocity. If the lower boundary is crossed (the pads of the landing legs), the lander is aligned with the lunar surface and then remains there.

Listing 4.139: Moonlander (requirement 6.2) – `Lander.move()`

```
    def move(self) -> None:
        self.velocity += cfg.GRAVITY * cfg.DELTATIME
        self.rect.top += self.velocity * cfg.DELTATIME
        if self.rect.bottom >= cfg.WINDOW.bottom - cfg.HORIZONT:
            self.rect.bottom = cfg.WINDOW.bottom -cfg.HORIZONT
```

Finally, `Game.update()` still needs to be adapted.

Listing 4.140: Moonlander (requirement 6) – `Game.update()`

```
    def update(self) -> None:
        self.background.update()
        self.lander.update(action="move")      # Should move!
```

4.3.7 Requirement 7: Counter-thrust

So ... where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/03%20Moonlander/v05

Requirement 7 Counter-thrust

If counter-thrust is activated using Space*, this thrust should affect the lander's descent speed. The counter-thrust should be* $-3\,\mathrm{m/s^2}$.

The counter-thrust is rather arbitrarily set to $-3\,\mathrm{m/s^2}$. The negative sign is used because this thrust acts in the exact opposite direction of the Moon's gravity.

Listing 4.141: Moonlander (requirement 7) – magnitude of the counter-thrust

```
WINDOW = pygame.Rect(0, 0, 600, 800)
FPS = 60
DELTATIME = 1.0 / FPS
HORIZONT = 50
# Physical constants (Moon conditions)
MOON_GRAVITY = 1.62                         # m/s²
EARTH_GRAVITY = 9.81                        # m/s²
PIXELS_PER_METER = 10                       # Scaling 1m = 10px
GRAVITY = MOON_GRAVITY * PIXELS_PER_METER   # = 16.2 px/s²
THRUST = -3 * PIXELS_PER_METER              # = 30.0 px/s²
```

In `move()`, the counter-thrust is now included in the velocity calculation. To do this, it is first checked whether counter-thrust has been activated by pressing Space.

Listing 4.142: Moonlander (requirement 7) – adjustment of `Lander.move()`

```
    def move(self) -> None:
        if self.thrusting:
            self.velocity += cfg.THRUST * cfg.DELTATIME
        self.velocity += cfg.GRAVITY * cfg.DELTATIME
        self.rect.top += self.velocity * cfg.DELTATIME
        if self.rect.bottom >= cfg.WINDOW.bottom - cfg.HORIZONT:
            self.rect.bottom = cfg.WINDOW.bottom -cfg.HORIZONT
```

With this in place, the player can now influence the descent speed of the lander using counter-thrust.

4.3.8 Requirement 8: Fuel

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/03%20Moonlander/v06

Requirement 8 Fuel

1. *The lander has a limited fuel supply.*
2. *Depending on the difficulty level, different fuel amounts are available.*
3. *Fuel consumption is 20 units per second.*
4. *When the fuel supply is empty, no counter-thrust can be generated.*

First, the game constants are defined in `config.py`. `THRUST` represents the counter-thrust, but not in the unit m/s^2; instead, it is given in px/s^2. The possible fuel supplies for requirement 8.2 are stored in the dictionary `LEVEL` in line 15.

Listing 4.143: Moonlander (requirement 8) – preparations in `config.py`

```
15 LEVEL = {"easy":sys.maxsize, "fair":500, "hard":450, "ai":380} #
```

In the constructor of `Lander`, the initial fuel supply is defined, and the current fuel level is stored in the attribute `fuel`, initialized with this starting value.

Listing 4.144: Moonlander (requirement 8) – adjustment in the constructor of `Lander`

```
        self.fuel_initial = cfg.LEVEL["fair"]        # Starttreibstoff
        self.fuel = self.fuel_initial                # Aktueller Treibstoff
        self.fuel_consumption = 20                   # Treibstoffverbrauch pro Sekunde
```

In line 219, it is now checked before calculating the counter-thrust whether there is still any fuel left in the tank, and in line 221 the consumed fuel is subtracted from the tank. If the tank is empty, the counter-thrust mode must be switched off and, to prevent a negative fuel value, the fuel level must be clamped to 0.

Listing 4.145: Moonlander (requirement 8) – `Lander.move()`

```
218     def move(self) -> None:
219         if self.thrusting and self.fuel > 0:                # Fuel remaining?
220             self.velocity += cfg.THRUST * cfg.DELTATIME
221             self.fuel -= self.fuel_consumption * cfg.DELTATIME   # Fuel consumption
222             if self.fuel < 0:
223                 self.thrusting = False
224                 self.fuel = 0
225         self.velocity += cfg.GRAVITY * cfg.DELTATIME
226         self.rect.top += self.velocity * cfg.DELTATIME
227         if self.rect.bottom >= cfg.WINDOW.bottom - cfg.HORIZONT:
228             self.rect.bottom = cfg.WINDOW.bottom -cfg.HORIZONT
```

To verify that the fuel supply starts correctly, is reduced properly when counter-thrust is applied, and that thrust is disabled once the tank is empty, I added a `print()` statement to `Lander.draw()` in line 216.

Listing 4.146: Moonlander (requirement 8) – `Lander.draw()`

```
211     def draw(self) -> None:
212         if self.thrusting:
213             self.screen.blit(self.surface_thrusting, self.rect.topleft)
214         else:
215             self.screen.blit(self.surface, self.rect.topleft)
216         print(self.fuel)                             # Temporary only
```

Give it a try – it should work!

4.3.9 Requirement 9: Status display

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/03%20Moonlander/v07

Requirement 9 Status display

1. *A separate status display is required for the lander.*
2. *Velocity and altitude are displayed as text including their units.*
3. *If counter-thrust is active, a colored bar is shown.*
4. *The fuel supply is displayed as a progress bar.*

All essential changes related to this feature take place in the `Lander` class. Since I want the position of the separate status display to depend on the position of the main window, the constructor signature has to be changed. Instead of passing a `Surface` object, a `Window` object is now passed in line 118.

Listing 4.147: Moonlander (requirement 9) – `Lander.draw()`

```
117 class Lander:
118     def __init__(self, window: pygame.window.Window) -> None: #
119         self.screen = window.get_surface()
120         self.surface = pygame.Surface((90,81), pygame.SRCALPHA)
121         self.surface_thrusting = pygame.Surface((90,81), pygame.SRCALPHA)
122         self.rect = self.surface.get_frect()
123         self.rect.centerx = cfg.WINDOW.centerx
124         self.rect.top = self.rect.height
125         self.create_lander()
126         self.thrusting = False
127         self.velocity = 0
128         self.fuel_initial = cfg.LEVEL["fair"]
129         self.fuel = self.fuel_initial
130         self.fuel_consumption = 20
131         self.create_status_window(window)
```

The separate window is created in `create_status_window()`. First, a window of appropriate size is created and the corresponding `Surface` object is obtained. I want the status window to be positioned to the right of the main window and aligned with its top edge. To achieve this, I take the top edge of the main window and assign this value to the top edge of the status window. Then I take the left edge of the main window, add the width of the main window to obtain its right edge, and finally add an additional 10 px of spacing.

Listing 4.148: Moonlander (requirement 9) – `Lander.create_status_window()`

```
    def create_status_window(self, window: pygame.window.Window) -> None:
        self.status_window = pygame.Window(size=(300, 100), title="Status")
        self.status_screen = self.status_window.get_surface()
        top = window.position[1]
        left = window.position[0] + cfg.WINDOW.width + 10
        self.status_window.position = (left, top)
```

In the final line, the `draw()` method is extended by a call to `draw_status()`. As a result, each time `draw()` is executed, not only the lander in the main window is redrawn, but the status window is updated as well.

Listing 4.149: Moonlander (requirement 9) – `Lander.draw()`

```
    def draw(self) -> None:
        if self.thrusting:
            self.screen.blit(self.surface_thrusting, self.rect.topleft)
        else:
            self.screen.blit(self.surface, self.rect.topleft)
        self.draw_status()
```

In `draw_status()`, the window is first filled with a black background. Starting at line 229, the status display for altitude and fuel is rendered as text. From line 240 onward, two bars are drawn. The first bar is only shown when the lander is currently applying counter-thrust. The second bar consists of two rectangles: a gray bar is drawn across the full width of the window, and a green bar is drawn from the left, proportionally scaled according to the remaining fuel supply.

Listing 4.150: Moonlander (requirement 9) – `Lander.draw_status()`

```
    def draw_status(self) -> None:
        self.status_screen.fill("black")

        # Text output
        font = pygame.font.SysFont("Consolas", 14)
        labels = "Velocity (px/s):"
        labels += "\nHeight (px):"
        values = f"{self.velocity:>7.0f}"
        values += f"\n{-1*(self.rect.bottom - (cfg.WINDOW.bottom - cfg.HORIZONT)):>7.0f}"
        text_labels = font.render(labels, True, "white")
        text_values = font.render(values, True, "white")
        self.status_screen.blit(text_labels, (5, 10))
        self.status_screen.blit(text_values, (230, 10))

        # Bar display
        if self.thrusting:
            pygame.draw.rect(self.status_screen, (255, 140, 0), (5, 46, 290, 20))
        pygame.draw.rect(self.status_screen, "grey", (5, 70, 290, 20))
        ratio = int(290 * self.fuel / self.fuel_initial)
        pygame.draw.rect(self.status_screen, "green", (5, 70, ratio, 20))

        self.status_window.flip()
```

A few adjustments in `Game` are still required due to rendering output in multiple windows. One of these concerns event handling. When multiple windows are open in Pygame, the event `pygame.WINDOWCLOSE` must be processed (line 287). In this case, the flag of the main game loop has to be set to `False`, and the window associated with the event must be explicitly destroyed using `destroy()`.

Listing 4.151: Moonlander (requirement 9) – `Game.watch_for_events()`

```
283     def watch_for_events(self) -> None:
284         for event in pygame.event.get():
285             if event.type == pygame.QUIT:
286                 self.running = False
287             elif event.type == pygame.WINDOWCLOSE:  # New due to 2 windows!
288                 self.running = False
289                 event.window.destroy()
290             elif event.type == pygame.KEYDOWN:
291                 if event.key == pygame.K_ESCAPE:
292                     self.running = False
293                 elif event.key == pygame.K_r:
294                     self.restart()
295                 elif event.key == pygame.K_SPACE:
296                     self.lander.update(action="thrust")
297             elif event.type == pygame.KEYUP:
298                 if event.key == pygame.K_SPACE:
299                     self.lander.update(action="unthrust")
```

In `restart()`, the call to the constructor is also adjusted in line 316.

Listing 4.152: Moonlander (requirement 9) – `Game.restart()`

```python
    def restart(self) -> None:
        self.background = Sky(self.screen)
        self.moon = Moon(self.screen)
        self.earth = Earth(self.screen)
        self.lander = Lander(self.window)      # due to status window
        self.running = True
```

The result then looks like the one shown in figure 4.29.

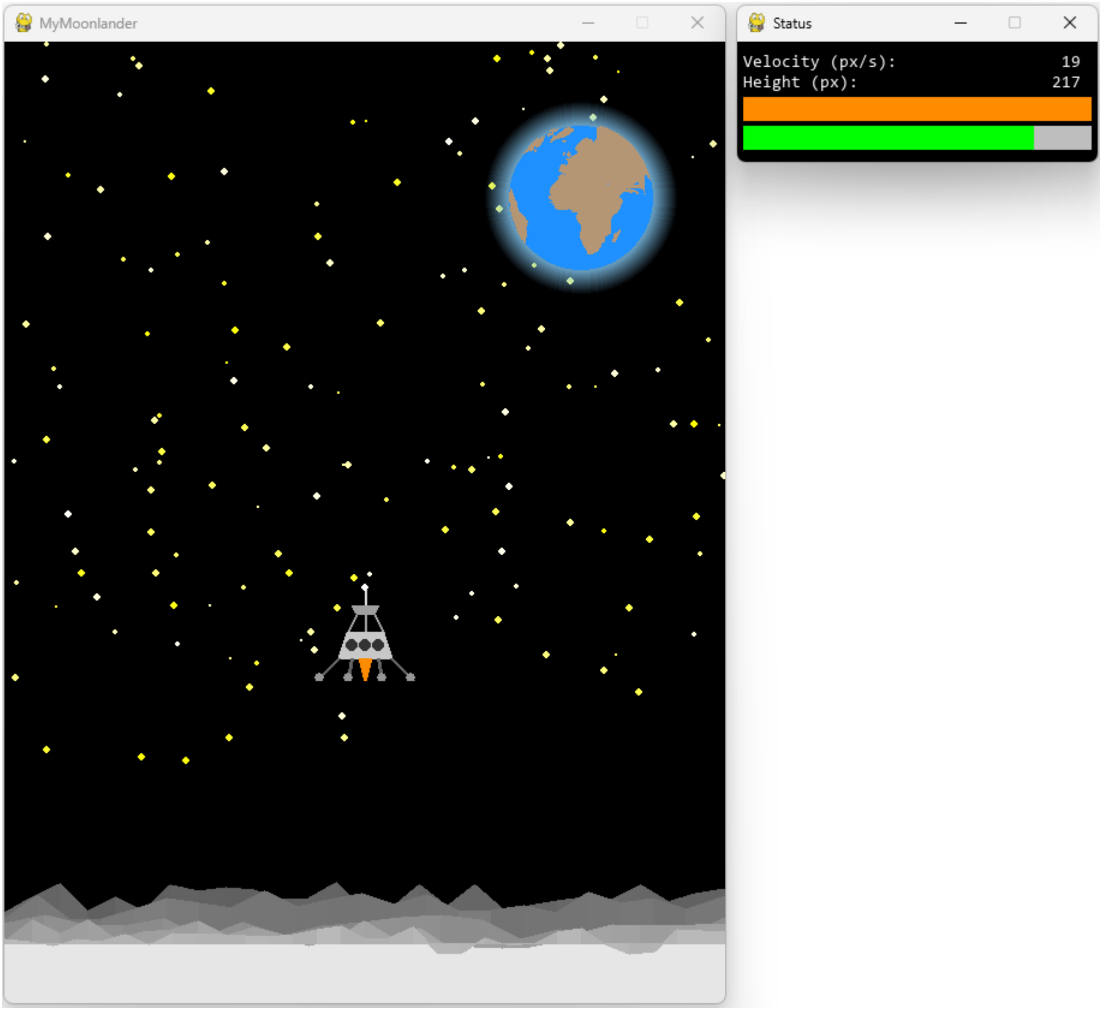

Figure 4.29: Moonlander (11) – now with status window

4.3.10 Requirement 10: Game over and restart

So . . . where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/03%20Moonlander/v08

Requirement 10 Game over and restart

1. *If the lunar module lands with a velocity of* $< 5\,\text{px/s}$*, the landing is considered safe.*
2. *If it lands at a higher velocity, it is considered destroyed.*
3. *The user is asked whether they want to quit the game with* `q` *or restart it with* `r`.

We prepare requirement 10.3 by encapsulating the display of the prompt in a simple class called `Question`. A `Surface` object containing the appropriate text is created and positioned accordingly. In `draw()`, this `Surface` object is then simply rendered on top of the lunar surface at the bottom of the screen.

Listing 4.153: Moonlander (requirement 10) – `question`

```
class Question:
    def __init__(self, screen:pygame.Surface) -> None:
        self.font = pygame.font.Font(None, 24)
        self.screen = screen
        self.surface = self.font.render("(Q)uit or (R)estart?", True, "red")
        self.rect = self.surface.get_rect()
        self.rect.centerx = cfg.WINDOW.centerx
        self.rect.bottom = cfg.WINDOW.bottom - 10

    def draw(self) -> None:
        self.screen.blit(self.surface, self.rect.topleft)
```

How are quitting or restarting the game actually triggered? There are many possible approaches. In this case, I decided to use *events* created with `pygame.event.Event()`. The basic idea is that touching the lunar surface triggers an event: `LANDED` if the descent speed is low enough, otherwise `CRASHED`.

Listing 4.154: Moonlander (requirement 10) – `MyEvents`

```
    LANDED = pygame.USEREVENT + 1
    CRASHED = pygame.USEREVENT + 2
```

This requires `watch_for_events()` to be rewritten. In line 343 and line 346, the two events are intercepted. In both cases, the new flag `landing` is set to `False`. This allows me to determine, for example, whether thrust may still be activated at all or whether the prompt for quitting or restarting the game should be displayed. In addition, an `update()` call is forwarded to the `Lander` so that it, too, is informed about its new state—for instance, to display an appropriate message in the status window.

For this reason, line 350 first checks whether the lander is still in the landing phase before allowing thrust to be activated.

The responses to the prompt are handled starting at line 354 and line 356. If `q` is pressed, the flag of the main program loop is simply set to `False`. If `r` is pressed, a restart is triggered by calling `restart()`.

Listing 4.155: Moonlander (requirement 10) – `Game.watch_for_events()`

```
336     def watch_for_events(self) -> None:
337         for event in pygame.event.get():
338             if event.type == pygame.QUIT:
339                 self.running = False
340             elif event.type == pygame.WINDOWCLOSE:
341                 self.running = False
342                 event.window.destroy()
343             elif event.type == MyEvents.LANDED:          #
344                 self.landing = False
345                 self.lander.update(mode="landed", velocity=event.volocity)
346             elif event.type == MyEvents.CRASHED:         #
347                 self.landing = False
348                 self.lander.update(mode="crashed", velocity=event.volocity)
349             elif event.type == pygame.KEYDOWN:
350                 if self.landing:                        #
351                     if event.key == pygame.K_SPACE:
352                         self.lander.update(action="thrust")
353                 else:
354                     if event.key == pygame.K_q:          #
355                         self.running = False
356                     elif event.key == pygame.K_r:        #
357                         self.restart()
358                 if event.key == pygame.K_ESCAPE:
359                     self.running = False
360             elif event.type == pygame.KEYUP:
361                 if event.key == pygame.K_SPACE:
362                     self.lander.update(action="unthrust")
```

In `Game`, the attribute `landing` is added to record whether the lander is still in the landing phase or has already touched the lunar surface.

Listing 4.156: Moonlander (requirement 10) – `Game.landing`

```
        self.landing = True
```

Finally, `restart()` is extended in line 134 to reset the `landing` flag.

Listing 4.157: Moonlander (requirement 10) – `Game.restart()`

```
    def restart(self) -> None:
        self.landing = True                         #
        self.background = Sky(self.screen)
        self.moon = Moon(self.screen)
```

```
        self.earth = Earth(self.screen)
        self.lander = Lander(self.window)
        self.question = Question(self.screen)
        self.running = True
```

Requirement 10.1 and requirement 10.2 are implemented in the new method `check_landing()` in `Lander`. When the lander reaches the lunar surface, its velocity is checked. If the descent speed is too high, the event `CRASHED` is triggered; otherwise, the event `LANDED` is emitted. The handling of these events itself has already been discussed above in `watch_for_events()` (source code 4.155 on the previous page).

Listing 4.158: Moonlander (requirement 10) – `Lander.check_landing()`

```
    def check_landing(self) -> None:
        if self.rect.bottom >= cfg.WINDOW.bottom - cfg.HORIZONT:
            if self.velocity > 5:
                evt = pygame.event.Event(MyEvents.CRASHED, volocity=self.velocity)
                pygame.event.post(evt)
            else:
                evt = pygame.event.Event(MyEvents.LANDED, volocity=self.velocity)
                pygame.event.post(evt)
```

Finally, `Game.update()` must be extended to include a call to `check_landing()`.

Listing 4.159: Moonlander (requirement 10) – `Game.update()`

```
    def update(self) -> None:
        self.background.update()
        self.lander.update(action="move")
        self.lander.check_landing()
```

A few adjustments in `Lander` are still required. First, in line 134, the attribute `mode` is introduced. It keeps track of which of the three states the lunar module is currently in: `landing`, `landed`, or `crashed`.

Listing 4.160: Moonlander (requirement 10) – `Lander.mode`

```
124 class Lander:
125     def __init__(self, window: pygame.window.Window) -> None: #
126         self.screen = window.get_surface()
127         self.surface = pygame.Surface((90,81), pygame.SRCALPHA)
128         self.surface_thrusting = pygame.Surface((90,81), pygame.SRCALPHA)
129         self.rect = self.surface.get_frect()
130         self.rect.centerx = cfg.WINDOW.centerx
131         self.rect.top = self.rect.height
132         self.create_lander()
133         self.create_lander_thrusting()
134         self.mode = "landing"                       # "landing", "landed" or "crashed"
135         self.thrusting = False
136         self.velocity = 0
137         self.fuel_initial = cfg.LEVEL["fair"]
138         self.fuel = self.fuel_initial
139         self.fuel_consumption = 20
140         self.create_status_window(window)
```

This attribute is set or updated in `update()`, starting at line 229. Once the ground has been touched – i.e. when the state is either `landed` or `crashed` – the thrust is switched off.

Listing 4.161: Moonlander (requirement 10) – `Lander.update()`

```
220     def update(self, *args: Any, **kwargs: Any) -> None:
221         if "action" in kwargs.keys():
222             if kwargs["action"] == "thrust":
223                 self.thrust(True)
224             elif kwargs["action"] == "unthrust":
225                 self.thrust(False)
226             elif kwargs["action"] == "move":
227                 if self.mode == "landing":
228                     self.move()
229         if "mode" in kwargs.keys():                 #
230             self.mode = kwargs["mode"]
231             if self.mode in ("landed", "crashed"):
232                 self.thrust(False)
```

The status display is now extended to show the game-over state as well. Its appearance can be examined in figure 4.30 on the following page.

Listing 4.162: Moonlander (requirement 10) – `Lander.draw_status()`

```
    def draw_status(self) -> None:
        self.status_screen.fill("black")

        # Text output
        font = pygame.font.SysFont("Consolas", 14, bold=True)
        labels = "Velocity (px/s):"
        labels += "\nHeight (px):"
        values = f"{self.velocity:>7.0f}"
        values += f"\n{-1*(self.rect.bottom - (cfg.WINDOW.bottom - cfg.HORIZONT)):>7.0f}"
        text_labels = font.render(labels, True, "white")
        text_values = font.render(values, True, "white")
        if self.mode == "landed":
            text_mode = font.render(f"Status: {self.mode}", True, "green")
        elif self.mode == "crashed":
            text_mode = font.render(f"Status: {self.mode}", True, "red")
        else:
            text_mode = font.render(f"Status: {self.mode}", True, "white")
        text_mode_rect = text_mode.get_rect(top=100)
        text_mode_rect.left = self.status_screen.get_rect().centerx - text_mode_rect.centerx
        self.status_screen.blit(text_labels, (5, 10))
        self.status_screen.blit(text_values, (230, 10))
        self.status_screen.blit(text_mode, text_mode_rect.topleft)

        # Bar display
        if self.thrusting:
            pygame.draw.rect(self.status_screen, (255, 140, 0), (5, 46, 290, 20))
        pygame.draw.rect(self.status_screen, "grey", (5, 70, 290, 20))
        ratio = int(290 * self.fuel / self.fuel_initial)
        pygame.draw.rect(self.status_screen, "green", (5, 70, ratio, 20))
        self.status_window.flip()
```

Figure 4.30: Moonlander (12) – quit or restart?

4.3.11 Requirement 11: Autopilot

So … where are the files?

https://github.com/adamsralf/pygame_book/tree/main/src/02%20Examples/03%20Moonlander/v09

Requirement 11 Autopilot

The autopilot can be switched on or off using `h`.

First, something that actually has nothing to do with requirement 11. I want the physical values to be a bit closer to real-world figures. The thrust is set to $-2.1\,\mathrm{m/s^2}$ and the safe landing speed to 2.5 m/s. According to NASA documentation, Apollo 11 touched down at 0.7 m/s. The NASA target value was 1 m/s, and the acceptable range was between 0.5 and 2.5 m/s. The structural limit was reached at 3 m/s. A value below 0.5 m/s would have resulted in unnecessary fuel consumption.

Listing 4.163: Moonlander (requirement 11) – some constants

```
THRUST = -2.1 * PIXELS_PER_METER            # = 21.0 px/s²
SAVE_SPEED_LANDING = 2.5 * PIXELS_PER_METER # Safe landing velocity in px/s
```

Now back to requirement 11: In `watch_for_events()`, the key press `h` is detected and forwarded to the `Lander`.

Listing 4.164: Moonlander (requirement 11) – extension of `watch_for_events()`

```
                if self.landing:
                    if event.key == pygame.K_SPACE:
                        self.lander.update(action="thrust")
                    elif event.key == pygame.K_h:
                        self.lander.update(action="toggle_ai")
```

In the constructor of `Lander`, the flag `ai` is introduced and initialized with `False` – although the term *ai* is admittedly a bit ambitious here ;-)

Listing 4.165: Moonlander (requirement 11) – extension of `Lander.__init__()`

```
        self.fuel_consumption = 20
        self.ai = False                         # AI flag
        self.create_status_window(window)
```

The `update()` method is extended as well. Starting at line 224, the `ai` flag is toggled on or off. When it is switched off, any thrust that may have been triggered by the autopilot must be stopped. In line 230, it is then checked whether the autopilot is active; if so, control is handed over to the autopilot.

Listing 4.166: Moonlander (requirement 11) – extension of `Lander.update()`

```
        elif kwargs["action"] == "toggle_ai":        # AI on/off
            self.ai = not self.ai
            if not self.ai:
                self.thrust(False)
        elif kwargs["action"] == "move":
            if self.mode == "landing":
                if self.ai > 0:                      # AI active?
                    self.controller()
                self.move()
```

Before diving into the actual implementation of the control logic, we first need to play around a bit with some physical formulas.

The formula for the final velocity in free fall is:

$$v = \sqrt{2 \cdot g \cdot h} \tag{4.1}$$

This equation gives us the final velocity v for a given gravitational acceleration g and a fall height h, assuming the initial velocity was $0\,\mathrm{m/s}$. However, we are not actually interested in the final velocity. What we really care about is the height h: from which height do we have to start applying counter-thrust in order to reach our target velocity?

So we rearrange equation 4.1 to solve for h:

$$\begin{aligned} v &= \sqrt{2 \cdot g \cdot h} && \| x^2 \\ v^2 &= 2 \cdot g \cdot h && \| : (2 \cdot g) \\ \frac{v^2}{2 \cdot g} &= h \end{aligned} \tag{4.2}$$

However, we are no longer dealing with lunar gravity alone; the counter-thrust of the lander also comes into play. In this case, the following applies:

$$acc = g_{Moon} + acc_{Lander} \tag{4.3}$$

Note that the sign of acc_{Lander} is opposite to that of the Moon's gravity g_{Moon} – that is, it is negative. We now substitute equation 4.3 into equation 4.2:

$$\begin{aligned} h &= \frac{v^2}{2 \cdot acc} && \| \leftarrow 4.3 \\ h &= \frac{v^2}{2 \cdot (g_{Moon} + acc_{Lander})} \end{aligned} \tag{4.4}$$

And equation 4.4 can already serve as the basis for our implementation. First, starting at line 239, we check whether the lander is still on its final approach. If not, all thrust is turned off and we are done, because there is nothing left to do.

In line 243, the net acceleration from equation 4.3 on the preceding page is computed, and then the target velocity is defined. This value is chosen to be far enough away from the maximum structural limit by using 50 %. In line 245, we then check whether the current velocity is already below this safe velocity. If it is, there is nothing to do—except that thrust must be switched off.

Next, following equation 4.4 on the facing page, the distance to the ground at which counter-thrust must begin is calculated. In line 251, counter-thrust is then activated or deactivated accordingly. All clear?

With this, the lander is fully implemented for the purposes of this script. If the autopilot performs a safe landing, it should look like the scene shown in figure 4.31 on the next page.

Listing 4.167: Moonlander (requirement 11) – `Lander.controller()`

```
238     def controller(self):
239         if self.mode in ("landed", "crashed"):          # Landing finished?
240             self.thrust(False)
241             return
242
243         acc = -1 * (cfg.THRUST + cfg.GRAVITY)           # Net acceleration
244         v_save = cfg.SAVE_SPEED_LANDING * 0.5           # 50% buffer
245         if self.velocity <= v_save:                     # Already slow?
246             self.thrust(False)
247             return
248
249         brake_distance = self.velocity ** 2 / (2 * acc)
250         ground_distance = (cfg.WINDOW.height - 50) - self.rect.bottom
251         self.thrust(ground_distance <= brake_distance)  # Required braking distance >=
                remaining height?
```

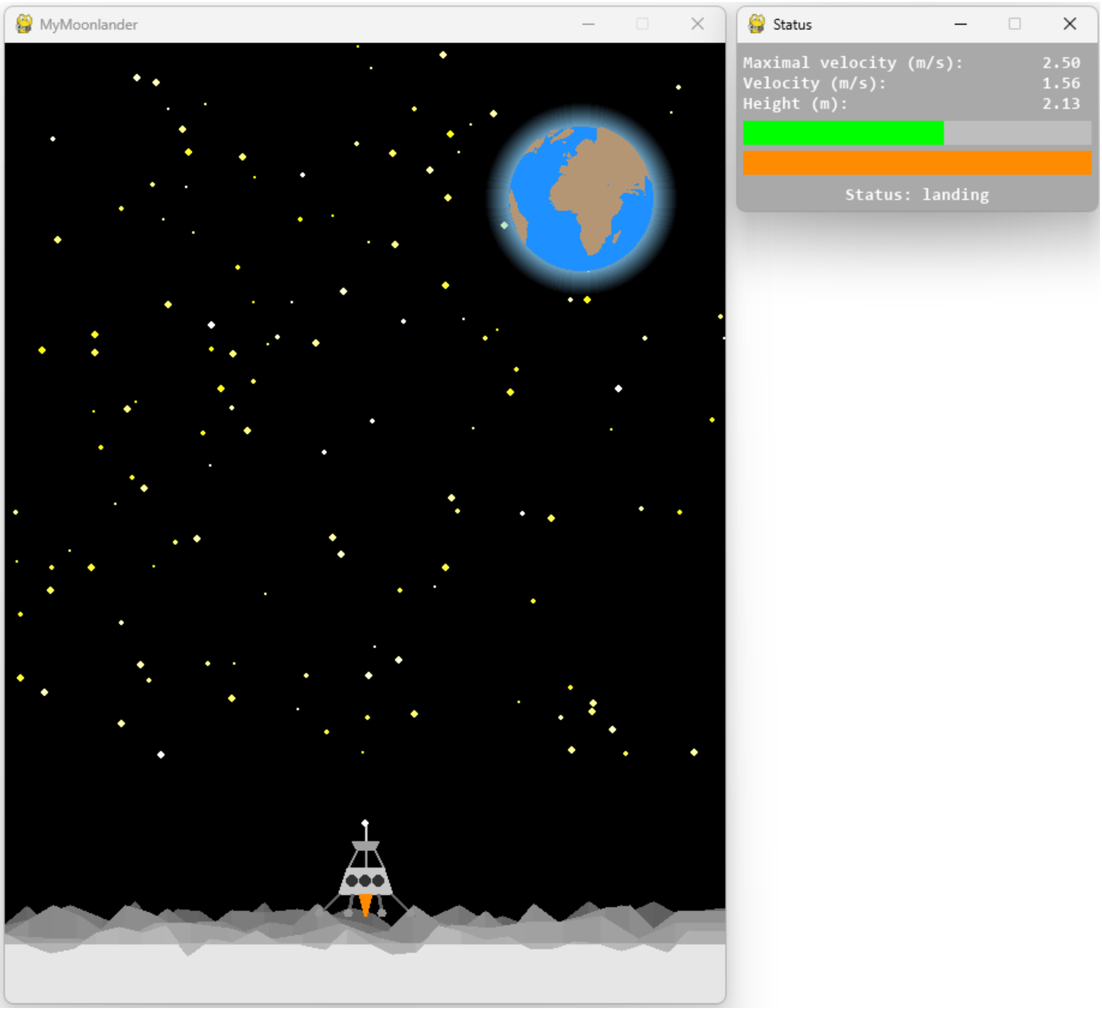

Figure 4.31: Moonlander (13) – autopilot

List of Figures

List of Tables

Glossary

Alpha Blending A technique used in computer graphics for transparency and color blending. In this process, the color values of a foreground pixel are combined with those of a background pixel based on an alpha value (α). The alpha value specifies how opaque a pixel is: $\alpha = 1$ means fully visible (opaque), $\alpha = 0$ means fully transparent. Formula:

$$C_{\text{new}} = \alpha \cdot C_{\text{foreground}} + (1 - \alpha) \cdot C_{\text{background}}$$

Pygame supports alpha blending through surfaces with an alpha channel (RGBA). This makes semi-transparent effects such as shadows, glow, or light effects possible. The quality and performance of alpha blending depend on the hardware and the SDL version. In the pygame-ce version, extended alpha operations and faster blit methods are available. 29

Alpha Channel For each pixel of an image, color information is usually stored in the RGB format: the red channel, the green channel, and the blue channel. Using an additional piece of information, it is also possible to specify how transparent the pixel should be. This additional information is called the alpha channel. 29

Array A data structure that stores values under a unique index (usually a non-negative integer). In the strict sense, arrays contain only elements of the same data type. This restriction does not apply to languages such as PHP or Python. 119

Binary AND Operation The binary AND operation combines two integer values by comparing their bits. For each bit position, the result is `1` if and only if both of the corresponding bits are `1`; otherwise, it is `0`. In Python, the binary AND operation is performed using the operator `&`. It is commonly used with bit masks, for example to test or combine modifier keys such as ⇧, Ctrl, or Alt in keyboard events. 92

Bitmap The term bitmap has two levels of meaning in this context. In general, it refers to the color and transparency information of an image stored in a file. Typical examples are files in the formats Joint Photographic Experts Group (JPEG), PNG, or Windows Bitmap Format (BMP). More specifically, it can also refer to the bitmap file format used for image storage (Windows Bitmap, BMP). 18

Boss Key A boss key is a special key or key combination that immediately exits, hides, or pauses a game without asking for confirmation. Historically, it was used to make a game disappear instantly when a boss, teacher, or other authority figure approached. The feature became popular in early PC games and is often mentioned with a touch of humor, but it can also serve as a practical shortcut for emergency exits or quick interruptions. 89

Clamp In programming, clamp (to *clamp* or *limit*) refers to a function that keeps a numerical value within a predefined range. If the value is below the minimum, the minimum is returned; if it is above the maximum, the maximum is returned. In games, clamping is commonly used to restrict camera positions, movements, or physical values (e.g. speed or zoom factor) to reasonable bounds—for example, to prevent the camera from scrolling outside the game world. 214

Class A class describes the properties (attributes) and the methods (functions) of a logically self-contained programming unit. In practice, there are many different kinds of classes, but in principle a class defines which information belongs to it (for example, the brand, color, and year of manufacture of a car) and what can be done with an object of that class (for example, accelerating, buying, or refueling a car). The information is called *attributes*, and the possible actions are called *methods* or *member functions*. In Python, classes are defined using the keyword `class` and are initialized with `__init__()`. 17

Collision Detection The process of checking whether two bitmaps *touch* each other in any way. In Pygame, we use three types of collision detection: checking whether the bounding rectangles of the bitmaps intersect, checking whether the inner circles of the bitmaps intersect, and checking whether non-transparent pixels of the bitmaps share the same coordinates. 57

Constant A constant is a value that cannot be changed while a program is running. In many programming languages, variables can be declared as constants – that is, as unchangeable values – using keywords such as `const`. Direct numeric or string values written in the source code are also constants. Python does not have true constants at the language level; instead, a common convention is used to write constants in CAPITAL LETTERS (for example, `PI = 3.14159`). Although the value is technically still changeable, this naming style signals to other developers that the variable is intended to remain unchanged. 17

CSV File A simple, text-based file format (*Comma-Separated Values*) used for storing tabular data. Values are separated by delimiters such as commas or semicolons. CSV files can be easily processed by programs such as spreadsheet applications, databases, or scripts. 191

Degree (°) A unit for measuring angles. A full circle has $2\pi°$. 149

Dictionary A data structure that stores values under a unique key. Other common names are: lookup table, associative array, hash table. 118

Don't ask – tell A design principle of object-oriented programming which states that an object should not ask another object about its internal state and then make decisions based on that information. Instead, an object should tell another object *what* to do, while the *how* remains fully encapsulated within the object itself. This improves encapsulation, substitutability, and maintainability, and reduces coupling between classes. 84

Double Buffer This is a second memory area (back buffer) that has the same size as the screen memory (front buffer). When something is drawn onto the playfield, it is first drawn into the back buffer. Only after all game elements have been drawn in their new appearance is the front buffer swapped with the back buffer in a single step. With certain hardware or graphics configurations, it can happen that the screen memory is redrawn even though the game has not yet finished updating all states. This can lead to ugly artifacts or flickering. Double buffering is used to prevent this effect. 18

DTP point Unit of measurement used in desktop publishing (DTP) and digital typography. A DTP point (also called a PostScript point) is defined as *one seventy-second of an inch* and therefore corresponds exactly to $\mathbf{1\,pt = 0.013\,888\,888\,888\,888\,89\,inch \approx 0.3528\,mm}$. This definition was originally introduced by Adobe with the PostScript standard and has become the industry standard for layout, printing, and graphics software. In contrast, traditional typography sometimes uses slightly different point sizes (e. g. 1 pica point = 0.3515 mm). The DTP point is used to specify font sizes, spacing, and page margins in layout programs such as InDesign, Scribus, or LaTeX. Many graphics libraries (e. g. Pygame-ce) and GUI frameworks also use DTP points indirectly via pixel conversions for precise on-screen rendering. 327

Equidistant The spacing between elements is always the same. For elements of equal size, this means that the space between them is always identical. For elements of different sizes, a reference point is required. For example, should the centers of the elements always have the same distance, or should the right edge of one element always have the same distance to the left edge of the next one? A distinction is also made between horizontal and vertical equidistance. 48

Error A message that indicates a serious problem that prevents a program or part of a program from continuing correctly. Errors usually require immediate action, such as fixing the code, correcting input data, or restarting the program. In many cases, an error causes the program to terminate. 53

Event In software engineering, an event is used to control the flow of a program. The program is not executed in a strictly linear way; instead, specific event-handling routines (such as listeners, observers, or event handlers) are executed whenever a particular event occurs. Event-driven programming is considered part of parallel programming techniques and therefore shares their advantages and disadvantages (source: Wikipedia). 17, 170

Fade Derived from the English verb *to fade*. In music and graphics, a distinction is made between *fade-in* and *fade-out*. During a fade-in, an image appears gradually or the volume is increased from zero to the target level. A fade-out does the opposite. 156

Flag A variable that stores only a Boolean value (`True` or `False`) and is used to control the flow of a program. Flags are often used to represent states such as *program running*, *game finished*, or *input allowed*. By setting or resetting the flag, the behavior of a program can be controlled in a targeted way. Example in Python: `running = True`. 18

Floating-point Number A floating-point number represents values as sums of powers of two, where the exponent can also be negative. Example: $6.75 = 2^2 + 2^1 + 2^{-1} + 2^{-2}$.Because storage space is limited, or because some numbers do not have a finite representation, this sum must be truncated at some point. The omitted terms lead to rounding errors. 69

Font Refers to a complete, digitally stored set of characters that contains information about the graphical representation of letters, digits, and symbols. A font defines typographic properties such as style, stroke weight, spacing, and size. In Pygame, a font is represented by a Font object, which is used to render text into a bitmap or Surface. Fonts can be installed system-wide or loaded from separate files (e. g. `.ttf`, `.otf`). 108

Frames Per Second Maximum number of images/frames per second. 19, 327

Framework In computer science, a framework refers to a working environment that provides predefined structures and functionality. This can include individual classes, function libraries, or even complete development environments such as an Integrated Development Environment (IDE). 78

Function In programming, a function is a block of instructions that has a name. It can take one or more parameters and can return results using `return`. In most cases, the principle applies that all values inside a function are local. In Python, a function definition starts with the keyword `def`. 17

Function Pointer A function pointer refers to a variable or reference that points to a function and allows it to be treated like a value. Function pointers make it possible to pass functions as parameters, store them in data structures, or select them dynamically. In languages such as C or C++, function pointers are used explicitly, while in Python functions are first-class objects and can therefore be used as function pointers without any special syntax. In Python, this is often referred to as a callable. 132

Gadget An interactive or usable game element that provides a player character with special abilities, advantages, or tools. Gadgets may appear as equipment, aids, or technical devices and often influence gameplay, puzzle solving, or combat mechanics in a video game. 191

Garbage Collector The garbage collector is a memory management component responsible for automatically freeing memory that is no longer in use. In Python, it detects objects that are no longer referenced by any part of the program and releases the memory they occupy. This helps prevent memory leaks and allows developers to focus on program logic rather than manual memory management. Python primarily uses reference counting, complemented by a garbage collection mechanism to detect and clean up cyclic references. 86

Generative Pre-trained Transformer ChatGPT is a prototype of a chatbot—that is, a text-based dialog system used as a user interface—based on machine learning. The chatbot was developed by the U.S.-based company OpenAI and was released in November 2022. (Source: Wikipedia). 327

Gravity A force that produces a constant acceleration acting on objects with mass. In game development, gravity is commonly implemented as a fixed acceleration vector applied each frame, rather than a full physical simulation. 32

Infinite loop In computer science, an infinite loop is a sequence of instructions that repeats endlessly and has no defined termination condition. In most cases, infinite loops are unintended and therefore represent an error in an application. They often arise from incorrect loop conditions. However, infinite loops are sometimes used intentionally, for example: `while True:`. 249

Infix An infix is a morphological element that is inserted inside a word stem to modify its grammatical or semantic meaning. In contrast to prefixes, which appear before the stem, and suffixes, which appear after the stem, an infix is placed within the stem itself. Infixes are common in many languages (for example Tagalog or Indonesian) but are rare in English and German. In English, infixes are mostly found in informal or expressive language, such as the emphatic insertion *-bloody-* in *abso**bloody**lutely*. 99

Information A message that provides additional details or context about the current state of a program. Information messages do not indicate a problem and do not require immediate action. They are often used to inform the user about successful operations, current settings, or progress. 53

Information Technology Assistants A state-recognized educational program at vocational colleges in North Rhine-Westphalia, Germany, providing school-based vocational training in the field of information technology. The program combines theoretical foundations with practical content from computer science, programming, network technology, databases, software development, and IT systems. Depending on the specific program, students may also obtain the advanced technical college entrance qualification or the general higher education entrance qualification. Graduates are qualified for employment in IT-related professional fields or well prepared for further academic studies. 244, 327

Integer An integer represents numbers as sums of powers of two with non-negative exponents. Example: $17 = 2^4 + 2^0$. The range of representable values is determined by the amount of memory allocated to the integer. With n bits available, unsigned integers can represent values in the range $[0, 2^n - 1]$, while signed integers can represent values in the range $[-2^{n-1}, 2^{n-1} - 1]$. 69

Integrated Development Environment An integrated development environment. It is called *integrated* because it does not only include a compiler and linker, but also tools such as a code editor, debugger, profiler, and other utilities that support the entire development process. 318, 327

Joint Photographic Experts Group A widely used raster image format for the compressed storage of digital images. JPEG uses lossy compression, in which fine details and color differences are partially removed or smoothed in order to significantly reduce file size. The compression level can be adjusted: Higher compression results in smaller files, but also in visible artifacts.

- Supports up to 24-bit color depth (true color)
- No transparency channel (alpha channel) as in PNG
- Ideal for photos, textures, and realistic graphics
- Less suitable for pixel art or UI elements with sharp edges

JPG files are commonly used in games, web applications, and desktop publishing (DTP) projects when storage space or loading time is more important than perfect image fidelity. In graphics libraries such as Pygame-ce, JPG files can be loaded directly and used as *surfaces*. 315, 327

Liskov Substitution Principle A principle of object-oriented programming stating that objects of a derived class must be usable anywhere an object of the base class is expected, without altering the correct behavior of the program. Formulated by Barbara Liskov in 1987. The LSP ensures that inheritance does not introduce unexpected side effects and that class hierarchies remain consistent. 84, 327

Main Loop Every non-trivial program must decide whether it should continue running or whether processing can be finished. If processing cannot or should not be finished yet, the program must continue with user interaction or other program functions, and it must do so until the program can or should be terminated. This behavior is usually controlled by a main loop. Examples: An operating system runs until it is shut down. A Windows application runs until ALT+F4 is pressed. 18

Margin Denotes the outer spacing of an element relative to other objects or to the boundary of a surrounding container. In *graphics* and *UI programming*, *margin* is used to specify how much empty space should exist outside a frame or surface. In contrast, padding describes the inner spacing between content and its border. In applications such as `CameraView`, *margin* can be used to define an outer safety area of the visible game region, determining how close objects or the camera are allowed to approach the edge. 321

Mask A mask is a bitmap that makes it possible to distinguish important pixels from unimportant pixels of a sprite. For sprites with transparency, the mask can easily be created by treating all transparent pixels as unimportant. To save memory and computing time, masks are often not stored in common bitmap formats, but bit by bit. One byte can therefore encode the mask information for 8 pixels. 125

Message Queue A queue provided by the operating system in which events and messages that are directed to applications or processes are stored. Such messages can include, for example, keyboard and mouse input, window events, or system signals. Programs regularly read from the message queue in order to react to current events. In event-driven applications such as graphical user interfaces or games, reading the message queue is a central part of the main program loop. 18

Milliseconds One thousandth (1/1000) of a second. 140, 327

Modulo A mathematical operator that returns the remainder of an integer division. The expression $a \bmod b$ yields the remainder that results when the number a is divided by b. The modulo operator is frequently used in computer science, for example for periodic processes, index calculations, or checking divisibility. 194

MP3 Short for *ISO MPEG Audio Layer 3*. An audio encoding and compression method for sound and music, developed largely by the German electrical engineer and mathematician Karlheinz Brandenburg. 158

Namespace A namespace is a structural area in which identifiers such as variable names, function names, or class names are defined. Namespaces prevent name collisions by allowing a clear and unique mapping of names to objects. In Python, there are, among others, local, global, and module-level namespaces. Example: A variable in a module can be accessed using `modulename.variable`. 17

Object Oriented The analysis, the design or the implementation based on objects – software entities that encapsulates data and functions. 327

OGG An audio file encoding format. The name comes from the English verb *to ogg*. The goal was to provide a license-free, simple, and efficient audio encoding format. 158

Operating System An operating system (OS) is the basic software that controls a computer and manages its hardware and software resources. It acts as an interface between the user, application programs, and the computer hardware. The operating system is responsible for tasks such as process management, memory management, file access, and input/output control. Examples of operating systems include Windows, Linux, and macOS. 18, 327

Padding Refers to additional spacing between an inner area and an outer boundary. In *graphics programming* and *UI layout*, *padding* specifies how much empty space exists inside a frame (e. g. a rectangle or surface) between the actual content and its border. In contrast, Margin denotes the outer spacing between an element and other objects. In games or camera implementations (e. g. in `CameraView`), *padding* can be used to define a safety zone around the player character or the visible area before a camera movement is triggered. 217, 320

Pixel The smallest addressable unit of a digital image or screen display. A pixel typically represents a color that is composed of individual color channels (for example, red, green, and blue). The combination of many pixels forms a complete image. The more pixels a screen or image has, the higher the possible resolution and level of detail. 18, 327

Polygon A closed polyline. It is usually defined by a sequence of points, where the last point is connected to the first one. 29, 322

Polyline A sequence of connected lines. It is usually defined by a sequence of points. A closed polyline is called a polygon. 30, 322

Portable Network Graphics A widely used raster image format that supports lossless compression and optionally an alpha channel for transparent pixels. PNG is especially well suited for graphics with sharp edges, text, or transparency effects and is often used in games and user interfaces. 46, 327

Pygame-ce A free and open-source library written in Python for developing 2D games and multimedia applications. Pygame-ce is based on the C library SDL and provides functions for graphics rendering, event handling, audio playback, and input control via keyboard, mouse, and game controllers. Pygame was originally created by Pete Shinners in the year 2000 and for many years was the standard framework for Python-based game development. However, since the original project did not receive updates for a long period of time, a fork called Pygame Community Edition (pygame-ce) was created in 2023. This version is actively developed by a community in order to support modern Python versions, improved graphics features (for example, alpha blending and better transformations), and higher performance. 9, 317, 320

Pylance Pylance is the default Python extension for Visual Studio Code that supports Python development. Its main features are type checking and code completion. Pylance helps detect errors early by analyzing code statically. 148

Pythagorean theorem In a right-angled triangle, the sum of the squares of the legs is equal to the square of the hypotenuse: $c^2 = a^2 + b^2$. The theorem is named after the mathematician *Pythagoras of Samos* (ca. 570 BC to ca. 510 BC). 255

Python Python is a high-level interpreted programming language that supports both procedural and object-oriented paradigms. It was created in 1991 by Guido van Rossum and is currently one of the most popular programming languages. 9

Radian (rad) A unit for measuring angles. A full circle has 2πrad. 149

Random Access Memory RAM (Random Access Memory) is the main working memory of a computer. It stores data and program code that are currently being used so that the processor can access them quickly. The contents of RAM are volatile, meaning they are lost when the computer is powered of. 93, 327

Red Green Blue An additive color model in which colors are represented by combining the three primary colors red, green, and blue. Each color channel typically has a value range from 0 to 255. By mixing different intensity levels, $256^3 = 16{,}777{,}216$ representable colors are created. The RGB model is used in digital displays, graphics programming, and image processing. 327

Rendering Refers to the process of creating a concrete, displayable image from abstract data (e. g. text, vectors, sprites, or 3D models). In computer graphics and in frameworks such as Pygame, rendering typically means that an object computes its visual representation (a *Surface* or *Bitmap*) and prepares it for display on the screen. Rendering is often performed only when the content or state of an object has changed, in order to save computational resources. 106

Rounding Error A rounding error occurs when a floating-point number cannot be represented exactly due to limited precision. Floating-point numbers are stored as sums of powers of two, and many decimal values (such as $\frac{1}{10}$ or $\frac{1}{60}$) do not have an exact finite representation in this system. As a result, values are stored as close approximations. When such approximations are used repeatedly in calculations, the small errors can accumulate and lead to noticeable deviations, especially in simulations, animations, and game loops. 69

Semantics The meaning of a statement or specification. It is usually used in contrast to the Syntax of a statement. 29, 325

Signature The signature of a function or method describes its formal properties that are visible from the outside. These include its visibility, return type, name, and parameters. The signature defines how a function or method can be called and how it interacts with other parts of the program. 84

Simple Direct Media Layer SDL is a cross-platform multimedia library written in C. It provides low-level access to graphics, sound, input devices (keyboard, mouse, game controllers), and window management. SDL is widely used in game development and serves as the underlying technology for many frameworks and engines. In the context of Pygame-ce, SDL handles the communication with the operating system, while Pygame provides a more convenient Python interface on top of it. In short: SDL does the heavy lifting in the background, and Pygame makes it easy to use from Python. 327

Sine function (sin) A trigonometric function that produces a smooth, periodic wave. For an angle x (usually in radians), it is defined as $y = \sin(x)$. The output is always between -1 and 1, and the function repeats every 2π. In game programming, sin is often used for natural-looking oscillations such as floating, bobbing, and smooth back-and-forth motion. **Pygame tip:** A common pattern is `y = base_y + amplitude * math.sin(t * speed)` to animate smooth vertical bobbing. 41

Single Responsibility Principle Each class or function should have exactly one responsibility. It should focus on that single task and do it well. A solution to a specific problem should be encapsulated in one class or one method, and changes to that responsibility should affect only that part of the code. 82, 327

Singleton A design pattern that ensures that there is exactly one instance of a class. This instance is usually provided in a (semi-)public way. Due to its conceptual similarity to global variables, the Singleton pattern is considered controversial. 163

Slicing A technique that allows convenient extraction of subsets from strings or arrays. 119

Socket In computer science, a socket is an endpoint for communication between two programs over a network. It provides a standardized interface for sending and receiving data using protocols such as TCP or UDP. A socket is typically identified by an IP address and a port number and is commonly used in client–server architectures. 273

Solid-State Drive A mass storage technology that is not based on magnetic principles but on semiconductor technology. 327

Sprite A single two-dimensional graphical element that typically represents a game character, an object, or an animation. Sprites are rendered independently of the background and can be moved, rotated, or scaled. Many game engines and libraries such as Pygame provide special sprite classes for efficient management and updating of game objects. Other names include *movable object (MOB)* or *blitter object (BOB)*. 46, 324

SpriteLib A freely available collection of 2D game sprites, often used for prototyping games and for educational purposes. SpriteLib provides graphics such as characters, objects, and animations that can be used in game engines or frameworks like Pygame. 115, 191

Stereo Refers to a two-channel audio technique in which a sound signal is played back separately through a left and a right channel. Different volume levels or signals on the two channels create a spatial listening impression. In game development, stereo is often used to convey the position of a sound source in space, for example through stereo panning, where the volume of a sound is adjusted depending on the horizontal position of an object. 160

Stereo Panning An audio technique that places a sound in space by changing its volume on the left and right speakers. With stereo panning, the sound of a source is placed more strongly on the left or right channel depending on its horizontal position in space. In game development, for example with Pygame, stereo panning is used to make the position of an object audible, typically by controlling the left and right audio channels of a sound separately. 160, 163

Strategy Pattern A **design pattern** from the category of **behavioral patterns**. It is used to make a specific behavior of an object **interchangeable** without modifying the object's class. The Strategy Pattern encapsulates a family of algorithms in separate classes that all implement a common interface. The context object delegates the execution of a particular task to the currently selected strategy. This allows the behavior to be changed dynamically at runtime. 220

Superclass In object-oriented programming, a superclass is a class from which another class (the subclass) inherits attributes and methods. The superclass defines common behavior and interfaces that can be reused and extended by subclasses. In Python, the superclass can be accessed using the `super()` function, which allows a subclass to call methods of its superclass, such as the constructor `__init__()`. 85

Syntax The form or grammar of a statement or specification. It is usually used in contrast to the Semantics of a statement. 323

Tiled A free, cross-platform tile map editor for creating 2D tile-based maps for games. Tiled supports various map formats (e. g. orthogonal, isometric, hexagonal) as well as flexible tileset structures. The created maps can be exported in numerous formats and integrated into game frameworks such as Pygame, Godot, or Unity. More information can be found at `https://www.mapeditor.org/`. 191

Toggling In computer science, this means that the value of a boolean variable switches from `True` to `False` or from `False` to `True`: *to toggle = to switch*. 268

Trade-off A situation in which an advantage is gained at the expense of a disadvantage. In algorithmic and software-related contexts, a trade-off requires evaluating – based on the available data—whether the overall benefits outweigh the associated costs. For example, using indexes can dramatically speed up access to database contents (benefit). To achieve this speed-up, additional files must be created and maintained, which makes inserting, updating, and deleting data slower (cost). In software development, a trade-off often describes the deliberate balancing between competing factors such as accuracy and computational effort, memory usage and execution speed, or flexibility and complexity. Trade-offs are unavoidable and require decisions that depend on the specific situation. 133

TrueType Font The font information is not stored in bitmap form, but in a vector-based format. This allows text to be rendered at *arbitrary* font sizes without losing quality. TrueType fonts are therefore well suited for scaling, high-resolution displays, and dynamic text rendering in applications and games. 327

Unicode A way of coding characters and symbols. Most popular implementations are UTF-8, UTF-16, and UTF-32. 120

Universal Serial Bus A bit-serial data transmission protocol used to connect peripheral devices to a computer. 327

Variable A named reference to an object in memory. In Python, a variable is not a container that stores a fixed data type, but a flexible name that can refer to arbitrary objects. Unlike a constant, the value of a variable can be changed at any time by assigning a new object using assignment (`=`). Examples: `x = 10` or `name = "Alice"`. 18

Visual Studio Code A free, cross-platform source code editor developed by Microsoft. It supports numerous programming languages and provides an integrated extension manager, debugging features, and a customizable interface. Due to its lightweight architecture, Git integration, and wide range of extensions, Visual Studio Code is suitable for both beginners and professional software development. 10

Warning A message that indicates a potential problem or an unusual situation. A warning does not stop the program, but it signals that something may lead to errors or unexpected behavior if it is not addressed. Warnings are meant to draw attention to issues before they become serious problems. 53

Wayland Wayland is a modern display server protocol for Linux and other Unix-like systems. It defines how graphical applications (clients) communicate with the display server (the compositor) to draw windows, handle input devices like keyboard and mouse, and manage screen updates. Unlike the older X11 system, Wayland is designed to be simpler, more secure, and more efficient. Many tasks that were previously handled by a separate window manager are integrated directly into the compositor, reducing complexity and latency. In practice, Wayland often results in smoother graphics, better support for modern hardware, and improved security, since applications cannot freely access or manipulate each other's windows. Popular desktop environments such as GNOME and KDE increasingly use Wayland as their default display system. 23

Windows Bitmap Format Image information stored in the Windows Bitmap format. 315, 327

Acronyms

BMP Windows Bitmap Format. 315, *Glossar:* Windows Bitmap Format

ChatGPT Generative Pre-trained Transformer. 223, *Glossar:* Generative Pre-trained Transformer

FPS Frames Per Second. 19, *Glossar:* Frames Per Second

IDE Integrated Development Environment. 318, *Glossar:* Integrated Development Environment

ITA Information Technology Assistants. 244, *Glossar:* Information Technology Assistants

JPEG Joint Photographic Experts Group. 315, *Glossar:* Joint Photographic Experts Group

LSP Liskov Substitution Principle. 84, *Glossar:* Liskov Substitution Principle

ms Milliseconds. 140, *Glossar:* Milliseconds

OO Object Oriented. 108, *Glossar:* Object Oriented

OS Operating System. 18, *Glossar:* Operating System

PNG Portable Network Graphics. 46, 315, 320, *Glossar:* Portable Network Graphics

pt DTP point. 108, *Glossar:* DTP point

px Pixel. 18, *Glossar:* Pixel

RAM Random Access Memory. 93, *Glossar:* Random Access Memory

RGB Red Green Blue. 19, 315, *Glossar:* Red Green Blue

SDL Simple Direct Media Layer. *Glossar:* Simple Direct Media Layer

SRP Single Responsibility Principle. 82, *Glossar:* Single Responsibility Principle

SSD Solid-State Drive. 18, *Glossar:* Solid-State Drive

TTF TrueType Font. 112, *Glossar:* TrueType Font

USB Universal Serial Bus. 18, *Glossar:* Universal Serial Bus

Index

Index of the pygame Namespace