

Einführung in die 2D-Spieleprogrammierung mit Pygame

Ralf Adams (TBS1, Bochum)

Version 0.10 vom 22. April 2024

Inhaltsverzeichnis

1	Ziele	4
2	Grundlagen	5
2.1	Das erste Beispiel	5
2.2	Grafikprimitive	11
2.2.1	Grundlagen	11
2.2.2	Beispiel: Fontaine	13
2.3	Bitmaps laden und ausgeben	21
2.4	Bitmaps bewegen	29
2.4.1	Grundlagen	29
2.4.2	Geschwindigkeiten normalisieren (<i>deltatime</i>)	33
2.5	Sprite-Klasse	46
2.6	Tastatur	54
2.7	Textausgabe mit Fonts	62
2.7.1	Default-Font	62
2.7.2	Fontliste	65
2.8	Textausgabe mit Bitmaps	69
2.9	Kollisionserkennung	75
2.10	Zeitsteuerung	83
2.11	Animation	90
2.11.1	Die laufende Katze	90
2.11.2	Der explodierende Felsen	95
2.12	Maus	98
2.13	Soundausgaben	104
2.13.1	Hintergrundmusik und Soundereignisse	104
2.13.2	Stereo	108
2.14	Dirty Sprites	115
2.14.1	Einfaches Beispiel	115
2.14.2	Performancemessungen	121
2.14.3	Fazit	124
2.15	Events	127
2.15.1	Welche Infos stecken in einem Event?	127
2.15.2	Wie kann ich selbst ein Event erzeugen?	128
2.15.3	Wie kann ich periodisch Ereignisse erzeugen lassen?	133

3 Beispielprojekte	135
3.1 Pong	135
3.1.1 Requirement 1: Standards	135
3.1.2 Requirement 2: Die Schläger	137
3.1.3 Requirement 3: Der Ball	140
3.1.4 Requirement 4: Punkte	142
3.1.5 Requirement 5: Tennisschlag	144
3.1.6 Requirement 6: Computerspieler	145
3.1.7 Requirement 7: Sound	148
3.1.8 Requirement 8: Pause und Hilfebildschirm	150
3.2 Bubbles	152
3.2.1 Requirement 1: Standards	152
3.2.2 Requirement 2: Blasen erscheinen	155
3.2.3 Requirement 3: Blasenanzahl	159
3.2.4 Requirement 4: Blasenwachstum	159
3.2.5 Requirement 5: Mauscursor	162
3.2.6 Requirement 6: Blasen zerplatzen	163
3.2.7 Requirement 7: Punktestand	164
3.2.8 Requirement 8: Spielende	167
3.2.9 Requirement 9: Zeitanpassungen	168
3.2.10 Requirement 10: Kollision anzeigen	170
3.2.11 Requirement 11: Pause	173
3.2.12 Requirement 12: Neustart	175
3.2.13 Requirement 13: Sound	177

1 Ziele

Dieses Skript ist eine Einführung in die Programmierung zweidimensionaler Spiele mit Hilfe von [Pygame](#) mit der Programmiersprache [Python](#).

Im ersten Teil werden die wichtigsten Konzepte anhand einfacher Beispiele eingeführt. Im zweiten Teil werden Spielprojekte vollständig durchprogrammiert und Probleme bzw. Techniken der Spieleprogrammierung verdeutlicht.

Es bleibt offen, welche Entwicklungsumgebung verwendet wird; ich verwende Visual Code.

Aber Version 0.9 liegt diesem Skript der Pygame-Fork *Pygame Community Edition* ([Pygame-ce](#)) zu Grunde. Die Quelltexte werden nicht auf Kompatibilität mit dem ursprünglichen Pygame abgeglichen. Der besseren Lesbarkeit halber werde ich aber immer von Pygame sprechen und nicht zwischen den beiden Modulen unterscheiden.

Das Skript wird sich nur mit Spielen beschäftigen, die unmittelbar auf dem Desktop laufen. Oder anders herum: Es wird kein Game-Server, Webspiel oder Mobile-Spiel implementiert.

Für eine Rückmeldung bei groben Patzern wäre ich sehr dankbar: adams@tbs1.de.

2 Grundlagen

2.1 Das erste Beispiel

Quelltext 2.1: Mein erstes *Spiel*, Version 1.0

```
1 import os
2
3 import pygame # Pygame-Modul (auch bei Pygame-ce!)
4
5
6 def main():
7     os.environ["SDL_VIDEO_WINDOW_POS"] = "10,50" # Fensterposition
8     pygame.init() # Subsystem starten
9     pygame.display.set_caption("Mein erstes Pygame-Programm") # Fenstertitel
10
11    screen = pygame.display.set_mode((600, 400)) # Fenster erzeugen
12    running = True
13    while running: # Hauptprogrammschleife: start
14        for event in pygame.event.get(): # Ermitteln der Events
15            if event.type == pygame.QUIT: # Fenster X angeklickt?
16                running = False
17            screen.fill((0, 255, 0)) # Spielfläche einfärben
18            pygame.display.flip() # Doublebuffer austauschen
19
20    pygame.quit() # Subsystem beenden
21
22
23 if __name__ == "__main__":
24     main()
```

Wenn Sie jetzt die Anwendung starten, bekommen Sie eine schmucke grüne Spielfläche zu sehen (Abbildung 2.1). Beenden können Sie diese durch das Anklicken des X im Fensterrahmen oben rechts.

Um Pygame verwenden zu können, muss das Modul `pygame` importiert werden (Zeile 3). Danach stehen uns [Konstanten](#), [Funktionen](#) und [Klassen](#) des [Namensraums](#) zur Verfügung.

In Zeile 7 wird die [Umgebungsvariable](#) gesetzt, die erstmal nichts mit Pygame zu tun hat. Vielmehr wird hier die Umgebungsvariable `SDL_VIDEO_WINDOW_POS` des Betriebssystems, genauer der [Simple Direct Media Layer \(SDL\)](#), gesetzt. Diese Umgebungsvariable steuert die linke obere Startposition meines Fensters bezogen auf den ganzen Bildschirm.

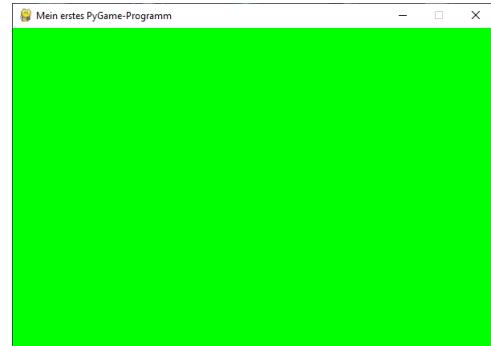


Abb. 2.1: Spielfläche

`SDL_VIDEO_WINDOW_POS`

Pygame ist nicht nur der Aufruf von Funktionen oder die Instantiierung von Klassen, sondern vielmehr wird ein ganzes Subsystem verwendet. Dieses Subsystem muss erst noch gestartet werden. Dabei klinkt sich Pygame in die relevanten Komponenten des Betriebssystems ein, damit diese im Spiel verwendet werden können. In Zeile 8 wird der ganze Pygame-Motor mit `init()` angeworfen. Man könnte auch nur die Komponenten starten, die gerade gebraucht werden wie beispielsweise die Soundunterstützung mit `pygame.mixer.init()`.

Wir brauchen für unsere Spiele eine *Spielfläche*/ein Fenster innerhalb dessen sich alles abspielt. Die Funktion `pygame.display.set_mode()` liefert mir einen solchen Spielfläche. Die Funktion bekommt in Zeile 11 einen(!) Übergabeparameter – nämlich die Breite und die Höhe des Fensters als ein 2-Tupel. Unser Fenster ist also 600 *px* breit und 400 *px* (siehe [Pixel \(px\)](#)) hoch. Als Rückgabe bekomme ich ein `pygame.Surface`-Objekt, was ungefähr sowas wie ein [Bitmap](#) ist.

Diese Rückgabe speichere ich in die Variable `screen`. Das ist aber nicht zwingend nötig. Pygame merkt sich einen Verweis auf diese Spielfläche. Mit `pygame.display.get_surface()` wird mir dieser Verweis zurückgeliefert. Diese Methode kann im ganzen Programm aufgerufen werden und so können Sie sich später oft sperrige Übergaben dieser Information an andere Programmteile Ihres Spiels ersparen.

Dem Fenster kann ich dann noch mit `pygame.display.set_caption()` eine Titelüberschrift verpassen (siehe Zeile 9).

Das Spiel selbst – so wie auch alle zukünftigen Spiele – laufen innerhalb einer [Hauptprogrammschleife](#). Hier startet die Schleife in Zeile 13 und endet in Zeile 20. Innerhalb dieser Schleife werden zukünftig immer drei Dinge passieren:

1. Ereignisse auslesen und verarbeiten: Wie in Zeile 14f. werden Maus-, Tastatur- oder Konsolenereignisse festgestellt und an die Spielemente weitergegeben. In unserem Fall wird lediglich das Anklicken des X im Fenster oben rechts registriert.
2. Zustand der Spielemente aktualisieren: Basierend auf den oben festgestellten Ereignissen und den Zuständen der Spielemente, werden die neuen Zustände ermittelt (Spieler bewegt sich, Geschoss prallt auf, Punkte erhöhen sich etc.). In unserem Fall wird nur das Flag `running` der Hauptprogrammschleife auf `False` gesetzt.
3. Bitmaps der Spielemente malen: Die Spielemente haben eine neue Position oder ein neues Aussehen und müssen deshalb neu gemalt werden. In diesem Minimalbeispiel wird lediglich Zeile 17 der Hintergrund der Spielfläche eingefärbt und anschließend in Zeile 18 der `Doublebuffer` mit `pygame.display.flip()` ausgetauscht.

Pygame schleust durch den Aufruf von `pygame.init()` einen Horchposten in das Betriebssystem. Und zwar horcht Pygame die [Message Queue](#) ab. Dort werden vom Betriebssystem alle Meldungen eingesammelt, die durch Ereignisse ausgelöst werden. Dies können [USB](#)-Anschlussmeldungen, [SSD](#)-Fehlermeldungen, Mausaktionen, Programmstarts bzw. -abstürze usw. sein. Pygame fischt nun aus der Message-Queue mit Hilfe

von `pygame.event.get()` alle Events, die das Spiel betreffen könnten, heraus. Mit Hilfe einer `for`-Schleife iteriere ich nun die Ereignisse durch und picke die für mich interessanten heraus.

Dabei überprüfe ich zuerst, was für ein Ereignistyp (`pygame.event.type`) mir da angeboten wird. Derzeit ist für mich nur der Typ `pygame.QUIT` wichtig. Dieser Typ wird ausgelöst, wenn das Betriebssystem eine *Beenden*-Nachricht an die Anwendung sendet. Falls ich nun eine solche Nachricht empfange, setzte ich das `Flag` `running` auf `False`, so dass die Hauptprogrammschleife beendet wird.

Falls ich dieses Signal nicht empfange, läuft die Hauptprogrammschleife fröhlich weiter und füllt in Zeile 17 die gesamte Spielfläche mit `screen.fill()` mit einer Farbe – hier grün – ein. Bitte beachten Sie, dass ähnlich wie in Zeile 11 die Funktion einen Übergabeparameter – nämlich ein 3-Tupel – erwartet. Dieses 3-Tupel kodiert die Farbe durch `RGB`-Angaben zwischen 0 und 255. Hinweis: Hier können auch vordefinierte Farbnamen wie `green` stehen.

Alternativ zu `pygame.display.flip()` kann auch `pygame.display.update()` verwendet werden. Wenn wir die beiden Methoden hier austauschen würden, würden wir keinen Unterschied erkennen. Diese Methode ist aber in der Lage, nicht nur den ganzen Bildschirm neu zu zeichnen, sondern Sie können ihr ein Rechteck oder eine Liste von Rechtecken mitgeben, die neu gezeichnet werden sollen. Dies kann billiger sein, als den ganzen Bildschirm immer wieder neu zu zeichnen. Mehr dazu in Zusammenhang mit *Dirty Sprites* in Abschnitt 2.14 auf Seite 115.

Verbleibt noch Zeile 18: Dort wird die Funktion `pygame.quit()` aufgerufen. Diese Funktion ist quasi das Gegenteil von `pygame.init()` in Zeile 8. Alle reservierten Ressourcen werden wieder freigegeben und die Pygame-Horchposten werden wieder aus dem System entfernt. Rufen Sie diese Funktion unbedingt immer am Ende Ihrer Anwendung auf; beenden Sie nicht einfach das Spiel. Der Unterschied entspricht dem einfachen Herauslaufen aus der Wohnung und dem ordnungsgemäßen Lichtausmachen und Türabschließen beim Verlassen der Wohnung.

Wenn wir uns das Spiel mal im Task-Manager anschauen (siehe Abbildung 2.2), könnten wir leicht überrascht sein: Es werden rund 30% der CPU-Zeit für dieses *IchMacheJaEigentlichGarNichts*-Spiel verbraucht.



Abbildung 2.2: Ressourcenverbrauch ohne Taktung

Wenn wir uns die Hauptprogrammschleife anschauen, sollte es allerdings nicht wirklich verwundern. Da wird ungebremst ein Bitmap auf den Bildschirm gemalt und das ohne Unterbrechung. Besser wäre es, bei jedem Schleifendurchlauf genügend Zeit zur Verfügung zu stellen, um die Ereignisse einzusammeln, die neuen Zustände zu berechnen und erst dann die Bildschirmausgabe zu generieren. Die Bildschirmausgabe selbst sollte

fps

auch nicht beliebig schnell und oft passieren, sondern in der Regel reichen 60 **frames per second (fps)**, um eine Bewegung als flüssig wahrzunehmen.

Quelltext 2.2: Mein erstes *Spiel*, Version 1.1

```

1 import os
2
3 import pygame
4
5
6 def main():
7     os.environ['SDL_VIDEO_WINDOW_POS'] = "10,50"
8     pygame.init()
9     pygame.display.set_caption('Mein erstes Pygame-Programm')
10
11    screen = pygame.display.set_mode((600, 400))
12    clock = pygame.time.Clock() # Clock-Objekt
13
14    running = True
15    while running:
16        for event in pygame.event.get():
17            if event.type == pygame.QUIT:
18                running = False
19            screen.fill((0, 255, 0))
20            pygame.display.flip()
21            clock.tick(60) # Taktung auf 60 fps
22
23    pygame.quit()
24
25
26 if __name__ == '__main__':
27     main()

```

Clock

In Zeile 12 wird zur Taktung ein `pygame.time.Clock`-Objekt erzeugt. Mit Hilfe dieses Objektes können verschiedene zeitbezogene Aufgaben bewältigt werden, wir brauchen das Objekt im Moment nur für die Taktung in Zeile 21. Dort wird `pygame.time.Clock.tick()` mit einer Framerate gemessen in *fps* aufgerufen. Diese Funktion sorgt dafür, dass die Anwendung nun mit maximal 60 *fps* abläuft. Dies ist an dem deutlich reduzierten CPU-Verbrauch in Abbildung 2.3 zu erkennen.

tick()

tick_busy_loop()

Hinweis: In der Pygame-Dokumentation wird darauf verwiesen, dass die Funktion `tick()` zwar sehr ressourcenschonend, aber etwas ungenau sei. Falls Genauigkeit aber bei der Taktung wichtig ist, wird die Funktion `tick_busy_loop()` empfohlen. Deren Nachteil ist, dass sie aber erheblich mehr Rechenzeit als `tick()` verbraucht.



Abbildung 2.3: Ressourcenverbrauch mit Taktung

Was war neu?

Sie müssen folgendes tun, um ein minimales Pygame zu starten:

- Die Pygame-Bibliothek importieren.
- Das Pygame-System starten.
- Einen Fenster/eine Spielfläche erzeugen.
- Eine Hauptprogrammschleife anlegen:
 1. Events abfragen.
 2. Spielobjekte aktualisieren.
 3. Bildschirminhalt ausgeben.
 4. Schleifendurchläufe takten.
- Beim Verlassen das Pygame-System stoppen.

Es wurden folgende Pygame-Elemente eingeführt:

- `import pygame:`
<https://pyga.me/docs/tutorials/en/import-init.html>
- `os.environ['SDL_VIDEO_WINDOW_POS']:`
<https://docs.python.org/3/library/os.html#os.environ>
- `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.display.flip():`
<https://pyga.me/docs/ref/display.html#pygame.display.flip>
- `pygame.display.get_surface():`
https://pyga.me/docs/ref/display.html#pygame.display.get_surface
- `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.display.update():`
<https://pyga.me/docs/ref/display.html#pygame.display.update>
- `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>

2.2 Grafikprimitive

2.2.1 Grundlagen

Unter Grafikprimitiven versteht man gezeichnete einfache grafische Figuren wie Linien, Punkte, Kreise etc. Sie spielen in der Spieleprogrammierung nicht so eine große Rolle, können aber manchmal ganz nützlich sein. Ich will hier deshalb nur einige vorstellen.

Quelltext 2.3: Mein zweites *Spiel*, Version 1.0

```

1 import os
2
3 import pygame
4 import pygame.gfxdraw # Muss sein!
5
6
7 def main():
8     os.environ['SDL_VIDEO_WINDOW_POS'] = "10, 50"
9     pygame.init()
10    pygame.display.set_caption('Mein_zweites_Pygame-Programm')
11    screen = pygame.display.set_mode((530, 530))
12    clock = pygame.time.Clock()
13
14    mygrey = pygame.Color(200, 200, 200) # Eigene Farben
15
16    myrectangle1 = pygame.rect.Rect(10, 10, 20, 30) # Rechteck-Objekt
17    myrectangle2 = pygame.rect.Rect(60, 10, 20, 30)
18    points1 = ((120, 10), (160, 10), (140, 90)) # Punktliste
19    points2 = ((180, 10), (220, 10), (200, 90))
20
21    running = True
22    while running:
23        for event in pygame.event.get():
24            if event.type == pygame.QUIT:
25                running = False
26            screen.fill(mycgrey)
27            pygame.draw.rect(screen, "red", myrectangle1) # Gefülltes Rechteck
28            pygame.draw.rect(screen, "red", myrectangle2, 3, 5) # Rechteck
29            pygame.draw.polygon(screen, "green", points1) # Gefülltes Polygon
30            pygame.draw.polygon(screen, "green", points2, 1) # Polygon
31            pygame.draw.line(screen, "red", (5, 230), (240, 230), 3) # Linie
32            pygame.draw.circle(screen, "blue", (40, 150), 30) # Gefüllter Kreis
33            pygame.draw.circle(screen, "blue", (110, 150), 30, 2) # Kreis
34            pygame.draw.circle(screen, "blue", (180, 150), 30, 5, True) # Kreisbogenschnitt
35            for i in range(255):
36                for j in range(255):
37                    screen.set_at((265+i, 10+j), (255, i, j)) # Punkte Variante 1
38                    screen.fill((i, j, 255), ((10+i, 265+j), (1, 1))) # Variante 2
39                    pygame.gfxdraw.pixel(screen, 265+i, 265+j, (i, 255, j)) # Variante 3
40
41            pygame.display.flip()
42            clock.tick(60)
43
44    pygame.quit()

```

Der Grundaufbau ist der gleiche wie in Quelltext 2.2 auf Seite 8. Die Unterschiede beginnen in Zeile 14. Die Klasse `pygame.Color` kann Farbinformationen in verschiedenen Formaten inklusive eines [Alpha-Kanals](#) (Transparenz) kodieren; mehr dazu später in

[Color](#)

Abschnitt 2.3 auf Seite 21. Ich verwende hier eine RGB-Kodierung mit Farbkanalwerten zwischen 0 und 255.

Farbnamen

Für die meisten Fälle brauche ich mir aber keine eigenen Farben zu definieren. Pygame stellt mir eine wirklich umfangreiche Liste von 664 vordefinierten Farbnamen zur Verfügung. Überall dort, wo Farbwerte erwartet werden, kann ich entweder eine `Color`-Objekt, einen Zahlencode oder einen Farbnamen als String übergeben.

Rect

Gehen wir der Reihe nach die einzelnen Figuren durch und fangen mit dem Rechteck an. Es gibt mehrere Möglichkeiten, ein Rechteck in Pygame zu bestimmen. Da wir es später auch sehr oft brauchen, möchte ich hier schonmal die Klasse `pygame.Rect` einführen. Sie wird durch vier Parameter bestimmt: die linke obere Ecke, seine Breite und seine Höhe. In Zeile 16 wird also ein Rechteck an der Position (10, 10) mit der Breite von 20 px und einer Höhe von 30 px definiert.

Hinweis: Die Klasse `Rect` ist kein gezeichnetes Rechteck, sondern lediglich ein Container für Informationen, die für ein Rechteck interessant sind.

rect()

In Zeile 27 zeichnet `pygame.draw.rect()` ein gefülltes Rechteck. Die [Semantik](#) der Parameter sollte selbsterklärend sein. Anders der Aufruf von Zeile 28. Der erste Parameter hinter dem Rechteck – hier 3 – legt die Dicke der Linie fest. Ist dieser Parameter angegeben und größer 0, so wird das Rechteck nicht mehr ausgefüllt. Der Wert 10 legt die Rundung der Ecken fest. Dort kann ein Wert von 0 bis $\min(\text{width}, \text{height})/2$ stehen, entspricht er doch dem Radius der Eckenrundung.

polygon()

Allgemeiner als ein Rechteck ist ein [Polygon](#). Ein Polygon ist ein geschlossener Linienzug, der in Pygame durch seine Punkte (Ecken) definiert wird. Ähnlich wie bei den Rechtecken, gibt es gefüllte (Zeile 29) und ungefüllte (Zeile 30) Varianten. Beide werden mit Hilfe von `pygame.draw.polygon()` gezeichnet. Vorsicht bei der Liniendicke: Diese wachsen nach außen, so dass bald hässliche Versatzstücke an den Ecken erkennbar werden. Probieren Sie es aus, indem Sie den Wert 2 in 5 ändern.

line() lines()

Für einzelne Linien gibt es `pygame.draw.line()` bzw. für einen – hier ohne Beispiel – [Linienzug](#) `pygame.draw.lines()`. Ein Beispiel finden Sie in Zeile 31.

circle()

Ein Kreis wird durch zwei Angaben definiert: Mittelpunkt und Radius. In Zeile 32 wird mit `pygame.draw.circle()` ein gefüllter Kreis mit dem Mittelpunkt (40, 150) und einem Radius von 30 px gezeichnet. Wie bei Rechtecken und Polygonen gibt es auch nicht gefüllte Varianten (Zeile 33). Interessant ist der Kreisbogenausschnitt in Zeile 34. Hier

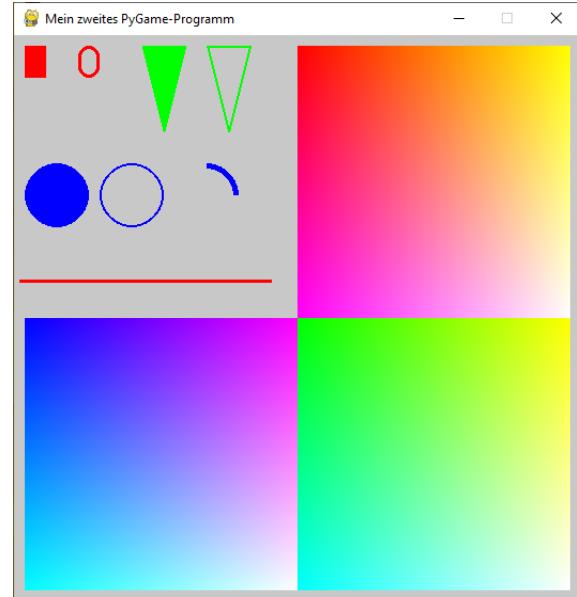


Abb. 2.4: Einige Grafikprimitive

wird über boolsche Variablen gesteuert, welcher Abschnitt des Kreisbogens gezeichnet wird (Näheres in der Pygame-Referenz).

Zum Schluss noch einen kleinen Farbenspielerei. Seltsamerweise gibt es in Pygame keine eigene Funktion zum Zeichnen eines einzelnen Punktes/Pixel. Ich habe hier mal drei Workarounds programmiert, die ich gefunden habe. Man könnte sich noch weitere überlegen: Eine Linie mit `start = ende`, ein Kreis mit dem Radius 1 usw.

In Zeile 37 wird der Punkt durch das Setzen eines einzelnen Farbwertes an einer Position mit `pygame.Surface.set_at()` gezeichnet. Man könnte auch die schon oben verwendete Surface-Funktion `fill()` mit einer Ausdehnung von nur einem Pixel Breite und Höhe verwenden (Zeile 38). Ein Möglichkeit einen Pixel über eine Grafikbibliothek zu setzen, ist die experimentelle `gfxdraw`-Umgebung. In Zeile 39 wird mit `pygame.gfxdraw.pixel()` ein einzelnes Pixel gesetzt. Die `gfxdraw`-Umgebung wird nicht automatisch durch `import pygame` importiert (siehe Zeile 4).

`set_at()`

`pixel()`

2.2.2 Beispiel: Fontaine

Man kann mit Grafikprimitiven dynamische Effekte einbauen, wie beispielsweise Partikelschwärme. Ich will hier mal ein super einfaches Beispiel für eine mausgesteuerte Fontaine aus Kreisen vorstellen.

Bauen wir zuerst ein kleines Programm, dass an der Mausposition einen Kreis zeichnet. Die Klasse `Circle` (siehe Zeile 7) enthält alle Informationen, die ich für das Zeichnen von Kreisen brauche: Position, Radius und Farbe. Die Position wird per Übergabeparameter bestimmt. In der Methode `draw()` wird die Bildschirmausgabe gekapselt.

`main()` enthält nun sehr viel bekanntes, aber auch ein paar Neuigkeiten. In Zeile 7 wird die Bildschirmgröße in einer Liste vorgehalten, da wir die Info noch an anderer Stelle als in Zeile 24 brauchen. Darunter wird in Zeile 26 eine Liste für die Aufnahme der Kreise definiert.

In der Hauptprogrammschleife wird in Zeile 34 abgefragt, ob die linke Maustaste gedrückt wurde. Wenn ja, wird ein Kreis an der Mausposition erzeugt. Anschließend wird der Bildschirm mit weißer Farbe aufgefüllt und die Kreise des Kontainers werden gemalt.

Das Ergebnis ist noch wenig berauschend (siehe Abbildung 2.5) und erinnert mehr an ein Malprogramm wie Paint.

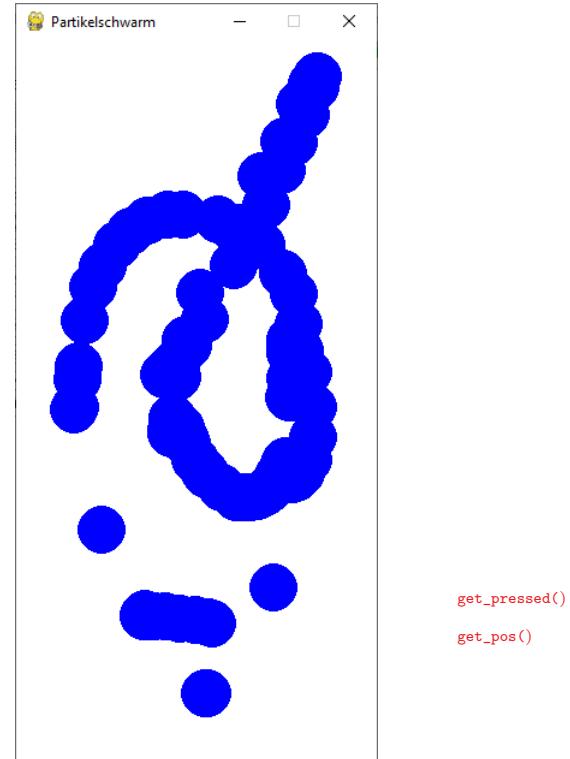


Abb. 2.5: Sicher keine Partikelfontaine

Quelltext 2.4: Partikelfontaine, Version 1.0

```

1 import os
2 from random import randint
3
4 import pygame
5
6
7 class Circle:                      # Nicht wirklich nötig, aber hilfreich
8     def __init__(self, pos) -> None:
9         self.posx = pos[0]
10        self.posy = pos[1]
11        self.radius = 20
12        self.color = "blue"
13
14    def draw(self):
15        screen = pygame.display.get_surface()
16        pygame.draw.circle(screen, self.color, (self.posx, self.posy), self.radius)
17
18
19 def main():
20     size = (300, 600)                  # Bildschirmgröße
21     os.environ['SDL_VIDEO_WINDOW_POS'] = "10,50"
22     pygame.init()
23     pygame.display.set_caption('Partikelschwarm')
24     screen = pygame.display.set_mode(size) #
25     clock = pygame.time.Clock()
26     circles = []                      # Kontainer für die Kreise
27
28     running = True
29     while running:
30         for event in pygame.event.get():
31             if event.type == pygame.QUIT:
32                 running = False
33
34             if pygame.mouse.get_pressed()[0]: # Linke Maustaste?
35                 circles.append(Circle(pygame.mouse.get_pos()))
36
37             screen.fill("white")
38             for p in circles:
39                 p.draw()
40
41             pygame.display.flip()
42             clock.tick(60)
43
44     pygame.quit()
45
46
47 if __name__ == '__main__':
48     main()

```

Im nächsten Schritt wollen wir aus den klobigen Kreisen bunte Partikel machen. Auch sollen diese nicht genau auf der Mausposition landen, sondern darum verstreut. Dazu müssen nur minimal Änderungen in der Klasse `Circle` vorgenommen werden. Die beiden Positionsangaben werden nun durch eine Zufallszahl zwischen -2 und $+2$ ergänzt. Auch wird der Radius auf 2 px reduziert. Die Farbe wird ebenfalls durch zufällige Werte gestreut. Hier habe ich einige Kombinationen ausprobiert und mit gefällt

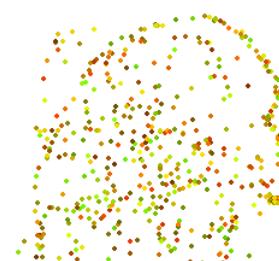


Abb. 2.6: Version 2

diese Farbvariation ganz gut. Spielen Sie ruhig selber mal mit den Farbkanälen und den Zufallswerten rum. Das Ergebnis in Abbildung 2.6 auf der vorherigen Seite sieht schon besser aus.

Quelltext 2.5: Partikelfontaine, Version 2.0

```

7  class Circle:
8      def __init__(self, pos) -> None:
9          self.posx = pos[0] + randint(-2, 2)
10         self.posy = pos[1] + randint(-2, 2)
11         self.radius = 2
12         self.color = [randint(100, 255), randint(50, 255), 0]

```

Nun wollen wir ein wenig Dynamik ins Spiel bringen. Die Partikel sollen zuerst nach oben steigen und dann nach unten fallen. Dazu habe ich in `Circle` die vertikale Geschwindigkeit `speedy` hinzugefügt und mit einem zufälligen Startwert versehen (Zeile 15). Die Division durch 10.1 sorgt dafür, dass keine glatten Werte entstehen. Spielen Sie auch hier mal mit den Werten rum, um die Effekte zu sehen.

Auch muss die Klasse um die Methode `update()` erweitert werden. In dieser Methode wird die neue vertikale Position `posy` anhand der vertikalen Geschwindigkeit `speedy` berechnet und die Geschwindigkeit wiederum bzgl. der Schwerkraftn `gravity` verändert. Damit alle Partikel immer der gleichen Schwerkraft unterliegen, habe ich `gravity` als statisches Attribut definiert (Zeile 8).

Schwerkraft

Quelltext 2.6: Partikelfontaine, Version 3.0, Klasse `Circle`

```

7  class Circle:
8      gravity = 0.3                                # Schwerkraft als statisches Element
9
10     def __init__(self, pos) -> None:
11         self.posx = pos[0] + randint(-2, 2)
12         self.posy = pos[1] + randint(-2, 2)
13         self.radius = 2
14         self.color = [randint(100, 255), randint(50, 255), 0]
15         self.speedy = randint(-100, 0) / 10.01      #
16
17     def update(self):
18         self.speedy += Circle.gravity
19         self.posy += self.speedy

```

Verbleibt noch der Aufruf von `update()` in der Hauptprogrammschleife.

Quelltext 2.7: Partikelfontaine, Version 3.0, Aufruf von `update()`

```

44     for p in circles:
45         p.update()
46
47     screen.fill("white")
48     for p in circles:
49         p.draw()

```

So richtig spritzig ist die Fontaine aber immer noch nicht. Streuen wir also die Partikel auch horizontal. Im Konstruktor wird dazu das Attribut `speedx` hinzugefügt. Die obere

und untere Grenze des Zufallszahlengenerators bestimmen die Breite der Partikelfontaine. Probieren Sie hier Werte aus, die ihrer Ästhetik entsprechen. In `update()` muss dann die neue horizontale Position `posx` berechnet werden.

Die horizontale Geschwindigkeit muss nicht angepasst werden, da `gravity` nur nach unten wirken soll.

Quelltext 2.8: Partikelfontaine, Version 4.0, `Circle.update()`

```

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

```

Die Liste `circles` enthält nach einiger Zeit viele Partikel, die überhaupt nicht mehr angezeigt werden. Wir wollen diese löschen. Dazu soll in der Klasse `Circle` festgestellt werden, ob man gelöscht werden kann. Im ersten Schritt fügen wir der Klasse das Löschflag `todelete` hinzu (siehe Zeile 17), welches auf `False` initialisiert wird; ein neuer Partikel soll natürlich nicht sofort gelöscht werden.

In `update()` wird dann nach der Berechnung der neuen Position überprüft, ob der Partikel zu löschen ist. Unser Kriterium soll sein, dass der Partikel den Bildschirm verlassen hat.

In Zeile 23 wird überprüft, ob der rechte Rand des Partikels (Mittelpunkt plus Radius), links außerhalb des Bildschirm liegt. Falls ja, muss das Löschflag auf `True` gesetzt werden. Analog werden in Zeile 25 und Zeile 27 der rechte und der untere Rand des Bildschirm überprüft.

`get_window_size()`

Dabei wird mit Hilfe der Methode `pygame.display.get_window_size()` jeweils die Breite bzw. Höhe des Bildschirms ermittelt. Diese Methode liefert mir an jedem Punkt meines Spiels die Bildschirmgröße als 2-Tupel wieder. Der nullte Wert ist dabei die Breite und der erste die Höhe. Ein Test, ob der Partikel nach oben verschwunden ist, wird nicht benötigt, da er ja irgendwann wieder runterfällt und damit wieder sichtbar wird.



Abb. 2.7: Partikelfontaine,
Version 5: fast fertig

Quelltext 2.9: Partikelfontaine, Version 5.0, Klasse Circle

```

10  def __init__(self, pos) -> None:
11      self.posx = pos[0] + randint(-2, 2)
12      self.posy = pos[1] + randint(-2, 2)
13      self.radius = 2
14      self.color = [randint(100, 255), randint(50, 255), 0]
15      self.speedx = randint(-10, 10) / 10.01
16      self.speedy = randint(-100, 0) / 10.01
17      self.todelete = False           # Löschflag
18
19  def update(self):
20      self.speedy += Circle.gravity
21      self.posx += self.speedx
22      self.posy += self.speedy
23      if self.posx - self.radius < 0:    # links raus
24          self.todelete = True
25      elif self.posx + self.radius > pygame.display.get_window_size()[0]:  # rechts raus
26          self.todelete = True
27      elif self.posy - self.radius > pygame.display.get_window_size()[1]:  # unten raus
28          self.todelete = True

```

Im Hauptprogramm muss ich nun einen passende Löschlogik implementieren. Vorab soll meine Fontaine aber noch mehr *Wumms* bekommen: In Zeile 51 wird nicht ein Partikel erzeugt, sondern immer gleich 5.

In Zeile 54 wird eine leere Liste erstellt, die die zu löschen Partikel enthalten wird. Innerhalb der Update-Schleife wird nun zusätzlich überprüft, ob der Partikel zu löschen ist (Zeile 57). Wenn ja, wird dieser Partikel in die Liste `todelete` aufgenommen. Nach Beendigung der Update-Schleife werden die zu löschen Partikel ab Zeile 59 aus der Liste `circles` entfernt.

In Abbildung 2.7 auf der vorherigen Seite können Sie eine Fontaine sehen. So richtig cool sieht das aber erst aus, wenn Sie die Maus dabei bewegen.

Quelltext 2.10: Partikelfontaine, Version 5.0, Hauptprogrammschleife

```

45  while running:
46      for event in pygame.event.get():
47          if event.type == pygame.QUIT:
48              running = False
49
50      if pygame.mouse.get_pressed()[0]:
51          for i in range(5):           # 5 Partikel gleichzeitig
52              circles.append(Circle(pygame.mouse.get_pos()))
53
54      todelete = []                  # Zwischenspeicher
55      for p in circles:
56          p.update()
57          if p.todelete:            # Zu löschen?
58              todelete.append(p)
59      for p in todelete:           # Löschen
60          circles.remove(p)
61
62      screen.fill("white")
63      for p in circles:
64          p.draw()
65
66      pygame.display.flip()
67      clock.tick(60)

```

Warum rufe ich nicht den `remove()` schon innerhalb der Update-Schleife auf? Deshalb: *Verlängern oder verkürzen Sie nie eine Liste, die Sie gerade durchwandern.* Es können höchst seltsame Effekte entstehen. Schätzen Sie mal die Anzahl der Schleifendurchgänge des folgendes Programmes.

```
1 klein = [1, 2, 3]
2 for a in klein:
3     klein.append(a*10)
4     print(klein)
```

Kleine Änderungen der Parameter können schon interessante visuelle Effekte haben. Leider können die hier nicht so gut durch Abbildungen gezeigt werden, daher: Selbst programmieren und ausprobieren.

Quelltext 2.11: Partikelfontaine, Version 6.0

```
1 import os
2 from random import randint
3
4 import pygame
5
6
7 class Circle:
8     gravity = 0.3
9     radius_inc = -0.1                                # Radiusinkrement
10
11     def __init__(self, pos) -> None:
12         self.posx = pos[0] + randint(-4, 4)           # Veränderte Streuung
13         self.posy = pos[1] + randint(-4, 4)
14         self.radius = 8
15         self.color = [randint(100, 255), randint(50, 255), 0]
16         self.speedx = randint(-15, 15) / 10.01          # Fontaine breiter
17         self.speedy = randint(-100, 0) / 10.01
18         self.todelete = False
19
20     def update(self):
21         self.speedy -= Circle.gravity
22         self.posx += self.speedx
23         self.posy += self.speedy
24         self.radius += Circle.radius_inc
25         if self.posx - self.radius < 0:
26             self.todelete = True
27         elif self.posx + self.radius > pygame.display.get_window_size()[0]:
28             self.todelete = True
29         elif self.posy - self.radius > pygame.display.get_window_size()[1]:
30             self.todelete = True
31         elif self.radius <= 0.0:                         # Können raus
32             self.todelete = True
33
34     def draw(self):
35         screen = pygame.display.get_surface()
36         pygame.draw.circle(screen, self.color, (self.posx, self.posy), self.radius)
37
38
39 def main():
40     size = (300, 600)
41     os.environ['SDL_VIDEO_WINDOW_POS'] = "10, 50"
42     pygame.init()
43     pygame.display.set_caption('Partikelschwarm')
44     screen = pygame.display.set_mode(size)
45     clock = pygame.time.Clock()
46     circles = []
```

```
47
48     running = True
49     while running:
50         for event in pygame.event.get():
51             if event.type == pygame.QUIT:
52                 running = False
53
54             if pygame.mouse.get_pressed()[0]:
55                 for i in range(5):
56                     circles.append(Circle(pygame.mouse.get_pos()))
57
58             todelete = []
59             for p in circles:
60                 p.update()
61                 if p.todelete:
62                     todelete.append(p)
63             for p in todelete:
64                 circles.remove(p)
65
66             screen.fill("white")
67             for p in circles:
68                 p.draw()
69
70             pygame.display.flip()
71             clock.tick(60)
72
73     pygame.quit()
74
75
76 if __name__ == '__main__':
77     main()
```

Was war neu?

Mit Hilfe von Grafikprimitiven können eigene Zeichnungen erstellt und verwendet werden. Sie stehen meist in einer gefüllten und ungefüllten Variante zur Verfügung. Farben können selbst definiert werden oder aus einer Liste von vordefinierten Farben ausgewählt werden.

Es wurden folgende Pygame-Elemente eingeführt:

- Vordefinierte Farbnamen:
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.display.get_window_size()`:
https://pyga.me/docs/ref/display.html#pygame.display.get_window_size
- `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

2.3 Bitmaps laden und ausgeben

Quelltext 2.12: Bitmaps laden und ausgeben, Version 1.0

```

1 import os
2
3 import pygame
4
5
6 class Settings:
7     WINDOW_WIDTH = 600
8     WINDOW_HEIGHT = 400
9     FPS = 60
10
11
12 def main():
13     os.environ['SDL_VIDEO_WINDOW_POS'] = "10, 50"
14     pygame.init()
15
16     screen = pygame.display.set_mode((Settings.WINDOW_WIDTH, Settings.WINDOW_HEIGHT))
17     pygame.display.set_caption("Bitmaps laden und ausgeben")
18     clock = pygame.time.Clock()
19
20     defender_image = pygame.image.load("images/defender01.png")      # Bitmap laden
21     enemy_image = pygame.image.load("images/alienbig0101.png")
22
23     running = True
24     while running:
25         for event in pygame.event.get():
26             if event.type == pygame.QUIT:
27                 running = False
28
29         screen.fill("white")
30         screen.blit(enemy_image, (10, 10))                                # Bitmap ausgeben
31         screen.blit(defender_image, (10, 80))
32         pygame.display.flip()
33         clock.tick(Settings.FPS)
34
35     pygame.quit()
36
37
38 if __name__ == '__main__':
39     main()

```

In Quelltext 2.12 werden zwei Bitmaps – hier zwei png-Dateien – geladen und auf den Bildschirm ausgegeben.

Das Laden erfolgt über die Funktion `pygame.image.load()`. In Zeile 20f. werden die Bitmaps – auch **Sprites** genannt – geladen und in ein Surface-Objekt umgewandelt. Die beiden Bitmaps werden dann, ohne sie weiter zu verarbeiten, mit Hilfe von `pygame.Surface.blit()` auf das `screen`-Surface gedruckt (Zeile 30). Der erste Parameter von `blit()` ist das Surface-Objekt, welches gedruckt werden soll, und danach erfolgt die Angabe der Position. Dabei wird zuerst die horizontale (waagerechte) und dann die vertikale (senkrechte) Koordinate angegeben. Der 0-Punkt ist dabei anders als in der Schulmathematik nicht links unten, sondern links oben. Das Ergebnis können Sie in Abbildung 2.8 auf der nächsten Seite *bewundern*.

`load()`

`blit()`

Wir wollen nun die Bitmaps ein wenig unseren Bedürfnissen anpassen. Zunächst emp-

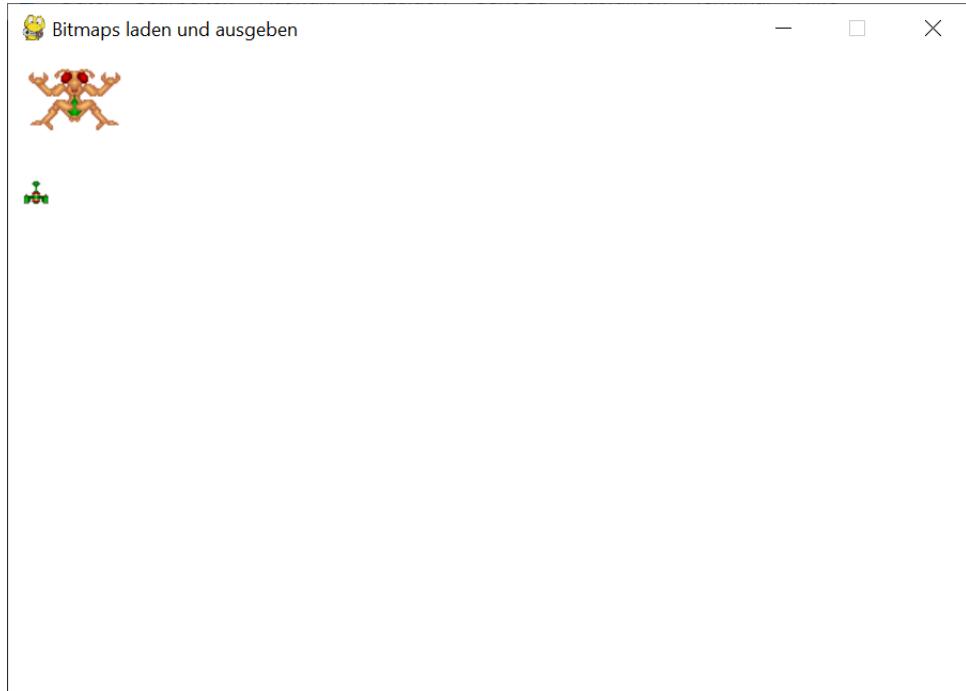


Abbildung 2.8: Bitmaps laden und ausgeben, Version 1.0

fiehlt das Handbuch, dass das Bitmap nach dem Laden in ein für Pygame leichter zu verarbeitendes Format konvertiert wird. Darüber hinaus möchte ich die Größenverhältnisse der beiden Bitmaps angleichen, da mir der Enemy im Verhältnis zum Defender zu groß ist.

Quelltext 2.13: Bitmaps laden und ausgeben, Version 1.1

```

20  defender_image = pygame.image.load("images/defender01.png").convert() # konvertieren
21  defender_image = pygame.transform.scale(defender_image, (30, 30)) # skalieren
22
23  enemy_image = pygame.image.load("images/alienbig0101.png").convert()
24  enemy_image = pygame.transform.scale(enemy_image, (50, 45))

```

Die Funktion `pygame.Surface.load()` lieferte mir ein Surface-Objekt zurück. Die Klasse Surface hat nun eine Methode, die mir die gewünschte Konvertierung vornimmt: `pygame.Surface.convert()`. Beispielhaft sei hier auf Zeile 20 verwiesen.

Das Verändern der Größe erfolgt durch `pygame.transform.scale()`. In Zeile 21 wird das Image auf die angegeben (*width, height*) in der Maßeinheit Pixel skaliert. Das Ergebnis an Abbildung 2.9 entspricht nicht ganz meinen Erwartungen.

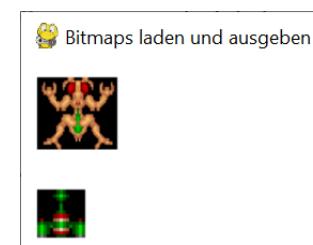


Abb. 2.9: Größen OK

Die Größenverhältnisse gefallen mir zwar jetzt, aber warum erscheint plötzlich ein schwarzer Hintergrund? Die Ursache dafür ist, dass durch die Konvertierung mit `convert()` die Information für die Transparenz verloren gegangen ist. Die Transparenz steuert die *Durchsichtigkeit* eines Pixels. Erreicht wird dies dadurch, dass zusätzlich zu jedem Pixel nicht nur die drei RGB-Werte, sondern auch eine Durchsichtigkeit abgespeichert wird. Diese zusätzliche Information nennt man den *Alpha-Kanal*.

Alpha-Kanal

Ich habe nun zwei Möglichkeiten, diese Transparenz wieder verfügbar zu machen:

- `pygame.Surface.convert_alpha()`: Ganz einfach formuliert bleibt bei der Konvertierung der Alpha-Kanal erhalten. Wenn möglich, sollte das das Mittel Ihrer Wahl sein.
- `pygame.Surface.set_colorkey()`: Als Übergabeparameter übergeben Sie die Farbe, die von Pygame beim Drucken auf das Ziel-Surface übersprungen werden soll. Dabei können zwei Nachteile entstehen. Zum einen können Transparenzen, die zwischen sichtbar und unsichtbar liegen, nicht abgebildet werden. Es wäre also nicht möglich, einen Pixel *halbdurchscheinen* zu lassen. Zum anderen werden Teile der Figur, die die gleiche Farbe wie der Hintergrund haben, ebenfalls transparent erscheinen. Würde unser Alien in der Mitte ein schwarzes Auge haben, würde es verschwinden und der Alien hätte ein Loch in der Mitte.

convert_alpha()

set_colorkey()

Quelltext 2.14: Bitmaps laden und ausgeben, Version 1.2

```
20  defender_image = pygame.image.load("images/defender01.png").convert_alpha()  #
21  defender_image = pygame.transform.scale(defender_image, (30, 30))
22
23  enemy_image = pygame.image.load("images/alienbig0101.png").convert()
24  enemy_image.set_colorkey("black")  # Transparenzfarbe setzen
25  enemy_image = pygame.transform.scale(enemy_image, (50, 45))
```

In Quelltext 2.14 habe ich beide Varianten mal ausprobiert und in Abbildung 2.10 können Sie das Ergebnis sehen. Nun sind beide Bitmaps ohne schwarze Hintergrund sichtbar, der weiße Hintergrund scheint wieder durch.

Was mir nun immer noch gefällt ist die Position und die Anzahl der Angreifer. Ich möchte den Verteidiger mittig unten platzieren und die Angreifer am oberen Bildschirmrand und zwar so, dass sie horizontal *äquidistant* sind. Dabei gibt es zwei Möglichkeiten: Ich gebe einen Mindestabstand an und die Anzahl wird ausgerechnet, oder ich gebe die maximale Anzahl an und der Abstand wird ausgerechnet. Welchen Weg ich wähle, hängt von meiner Spiellogik ab; meist ist die Anzahl vorgegeben.

Bitmaps laden und ausgeben

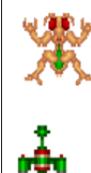


Abb. 2.10: Transparenz OK

äquidistant

Quelltext 2.15: Bitmap: Positionen, Version 1.4

```
13  def main():
14      os.environ['SDL_VIDEO_WINDOW_POS'] = "10, 50"
15      pygame.init()
```

```

16
17     screen = pygame.display.set_mode((Settings.WINDOW_WIDTH, Settings.WINDOW_HEIGHT))
18     pygame.display.set_caption("Bitmaps laden und ausgeben")
19     clock = pygame.time.Clock()
20
21     defender_image = pygame.image.load("images/defender01.png").convert_alpha()
22     defender_image = pygame.transform.scale(defender_image, (30, 30))
23     defender_pos_left = (Settings.WINDOW_WIDTH - 30) // 2           # linke Koordinate
24     defender_pos_top = Settings.WINDOW_HEIGHT - 30 - 5             # obere Koordinate
25     defender_pos = (defender_pos_left, defender_pos_top)           # Mache ein 2-Tupel
26
27     alien_image = pygame.image.load("images/alienbig0101.png").convert_alpha()
28     alien_image = pygame.transform.scale(alien_image, (50, 45))
29     space_for_aliens = Settings.ALIENS_NOF * 50                      # Verbrauchter Platz
30     space_available = Settings.WINDOW_WIDTH - space_for_aliens      # Verfügbarer Platz
31     space_nof = Settings.ALIENS_NOF + 1                         # Anzahl Freiräume
32     space_between_aliens = space_available // space_nof            # Platz Freiräume
33
34     running = True
35     while running:
36         for event in pygame.event.get():
37             if event.type == pygame.QUIT:
38                 running = False
39
40         screen.fill("white")                                         # Abstand von oben
41         alien_top = 10                                              # Berechnung/Ausgabe
42         for i in range(Settings.ALIENS_NOF):
43             alien_left = space_between_aliens + i * (space_between_aliens + 50)
44             alien_pos = (alien_left, alien_top)
45             screen.blit(alien_image, alien_pos)
46             screen.blit(defender_image, defender_pos)                 # Benutze Position
47             pygame.display.flip()
48             clock.tick(Settings.FPS)
49
50     pygame.quit()

```

In Quelltext 2.15 auf der vorherigen Seite sind die obigen Anforderungen umgesetzt worden. Schauen wir uns die einzelnen Aspekte genauer an.

Der Verteidiger sollte unten mittig positioniert werden. Wir erinnern uns, dass der Funktion `blit()` auch die Koordinaten der linken oberen Ecke mitgegeben werden.

Diese Angabe muss erst berechnet werden. Der Übersichtlichkeit wegen – in einem normalen Quelltext würde ich die Berechnung nicht so kleinteilig programmieren – berechne ich hier die Koordinaten einzeln.

Die obere Kante ist dabei recht einfach zu ermitteln. Würden wir `defender_top` auf die gesamte Höhe des Bildschirms `Settings.window_height` setzen, würden wir den Verteidiger nicht sehen, da er komplett unten aus dem Bildschirm rausragen würden. Um wie viele Pixel müssen wir also die obere Kante anheben? Genau um die Höhe des Raumschiffs, 30 *px*:

```
24     defender_pos_top = Settings.window_height - 30
```

Mir gefällt aber nicht, dass der Verteidiger dabei so an den Rand angeklebt aussieht. Ich spendiere ihm noch weitere 5 *px*, damit er mehr danach aussieht, als schwebt er im Raum:

```
24     defender_pos_top = Settings.window_height - 30 - 5
```

In Zeile 23 wird der Abstand des linken Rands des Bitmaps vom Spielfeldrand berechnet. Mit

```
23     defender_pos_left = Settings.window_width // 2
```

würden wir die horizontale Mitte des Bildschirmes ausrechnen. Diesen Wert können wir aber nicht einsetzen, da dann der linke Rand des Verteidigers in der horizontalen Mitte stehen würde – also zu weit rechts (siehe Abbildung 2.11).

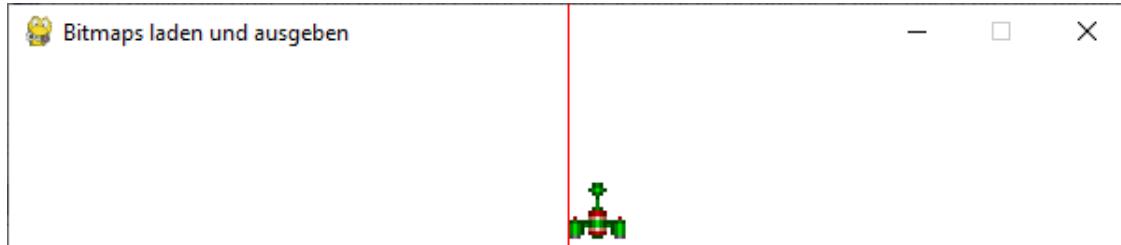


Abbildung 2.11: Bitmaps positionieren (Verteidiger)

Die Anzahl der Pixel, die wir zu weit nach rechts gerutscht sind, können wir aber genau bestimmen und dann abziehen: Es ist genau die Hälfte der Breite des Verteidigers (hier 30 px):

```
23     defender_pos_left = Settings.window_width // 2 - 30 // 2
```

Mit Hilfe von ein wenig Bruchrechnen lässt sich der Ausdruck vereinfachen:

```
23     defender_pos_left = (Settings.window_width - 30) // 2
```

Jetzt kommen die Angreifer. Im ersten Ansatz wollen wir diese hintereinander ohne Überschneidungen oben ausgeben. Die obere Kante `alien_top` können wir konstant mit einem angenehmen Abstand von 10 px vom oberen Rand setzen:

```
42     alien_top = 10
```

Die linke Position `alien_left` muss für jedes Alien einzeln bestimmt werden. Da diese erstmal direkt nebeneinander liegen, ist ein linker Rand genau die Breite eines Aliens vom nächsten linken Rand entfernt. Wenn ich also beim 0ten Alien bin, liegt die horizontale Koordinate direkt am linken Bildschirmrand. Beim 1ten Alien genau $1 \times 50 \text{ px}$, beim 2ten genau $2 \times 50 \text{ px}$ usw., da die Breite des Aliens 50 px beträgt. In eine for-Schleife gegossen, sieht das so aus:

```
43     for i in range(Settings.aliens_nof):
44         alien_left = i * 50
45         alien_pos = (alien_left, alien_top)
46         screen.blit(alien_image, alien_pos)
```

Der ganze Platz hinter dem letzten Alien kann jetzt aber vor, zwischen und nach den Aliens verteilt werden und zwar so, dass zwischen den Aliens, dem linken Alien und dem

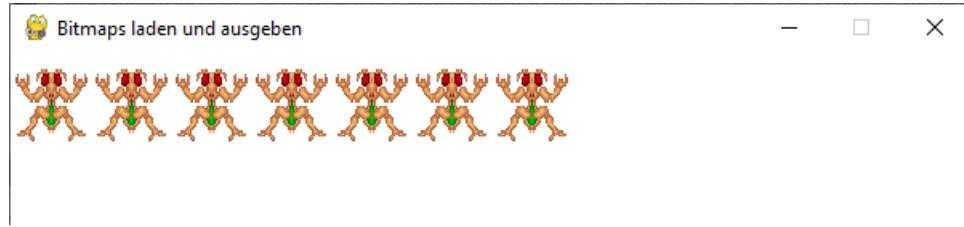


Abbildung 2.12: Bitmaps positionieren (Angreifer, Version 1)

linken Bildschirmrand und dem rechten Alien und dem rechten Bildschirmrand immer gleich viel Abstand liegt. Wie viele Zwischenräume sind es denn? Nun einmal die beiden ganz rechts und ganz links, also 2:

31 `space_nof = 2`

Dann die Anzahl der Zwischenräume zwischen den Aliens. Dies ist immer 1 weniger als die der Aliens (zählen Sie nach!):

31 `space_nof = Settings.aliens_nof - 1 + 2`

also:

31 `space_nof = Settings.aliens_nof + 1`

Nun muss der verfügbare Platz `space_available` hinter den Aliens noch ausgerechnet werden. Ich erreiche dies, indem ich den Platz, den die Aliens verbrauchen, `space_for_aliens` ausrechne

29 `space_for_aliens = Settings.aliens_nof * 50`

und diesen von der Bildschirmbreite abziehe.

30 `space_available = Settings.window_width - space_for_aliens`

Ich habe also den verfügbaren Platz in `space_available` und die Anzahl der Räume, die gefüllt werden müssen in `space_nof`. Wenn ich jetzt die Breite der Räume `space_between_aliens` ermitteln will, muss ich diese beiden Werte dividieren:

32 `space_between_aliens = space_available // space_nof`

Jetzt müssen wir nur noch die Berechnung von `alien_left` anpassen. Erstmal verschieben wir den Start um einen solchen Freiraum (siehe Abbildung 2.13 auf der nächsten Seite):

43 `for i in range(Settings.aliens_nof):`
44 `alien_left = space_between_aliens + i * 50`
45 `alien_pos = (alien_left, alien_top)`
46 `screen.blit(alien_image, alien_pos)`

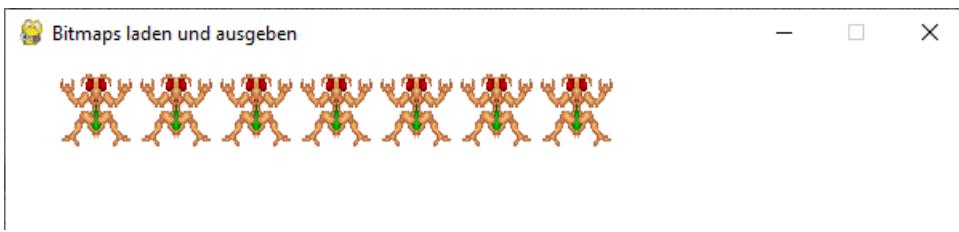


Abbildung 2.13: Bitmaps positionieren (Angreifer, Version 2)

Nun muss der Abstand von einem linken Rand zum anderen, der bisher nur aus der Breite des Aliens bestand, um den Abstand `space_between.aliens` erweitert werden:

```
43 for i in range(Settings.aliens_nof):
44     alien_left = space_between.aliens + i * (space_between.aliens + 50)
45     alien_pos = (alien_left, alien_top)
46     screen.blit(alien_image, alien_pos)
```

Und schon passt alles (siehe Abbildung 2.14).

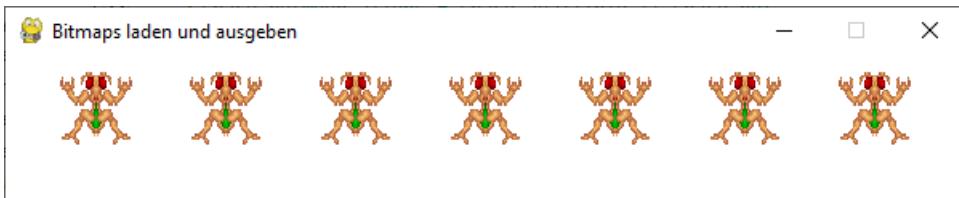


Abbildung 2.14: Bitmaps positionieren (Angreifer, Version 3)

Was war neu?

Die Positionsangaben werden bei der Ausgabe auf dem Bildschirm benötigt. Wir werden später sehen, dass wir die Positionsangaben auch noch für andere Fragestellungen brauchen, wie beispielsweise die [Kollisionserkennung](#). Die Positionsangabe bezieht sich immer auf die linke, obere Ecke des Bitmaps.

Das Koordinatensystem hat seinen 0-Punkt linksoben und nicht linksunten.

Wir müssen häufig elementare Geometrieberechnungen durchführen und am besten macht man diese Schritt für Schritt. Für solche Geometrieberechnungen werden folgende Informationen gebraucht: die Position des Bitmap, seine Breite und Höhe. Breite und Höhe haben wir hier noch als Konstanten verarbeitet, dass ist nicht zukunftsweisend.

Es wurden folgende Pygame-Elemente eingeführt:

- `pygame.image` :
<https://pyga.me/docs/ref/image.html>

- `pygame.image.load()` :
<https://pyga.me/docs/ref/image.html#pygame.image.load>
- `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.transform.scale()` :
<https://pyga.me/docs/ref/transform.html#pygame.transform.scale>

2.4 Bitmaps bewegen

2.4.1 Grundlagen

In der Zusammenfassung des vorherigen Kapitels haben wir für die Darstellung von Bitmaps notiert, dass wir die linke, obere Ecke als Positionsangabe und die Höhe und Breite beispielsweise für Abstandsberechnungen brauchen. Diese Angaben lassen sich gut einem Rechteck kodieren. Pygame stellt dazu die Klassen `pygame.Rect` (nur ganze Zahlen) und `pygame.Rect.FRect` (Fließkommazahlen) zur Verfügung. In Abbildung 2.15 finden Sie die meiner Ansicht nach wichtigsten Attribute der Klasse.

`Rect`
`FRect`

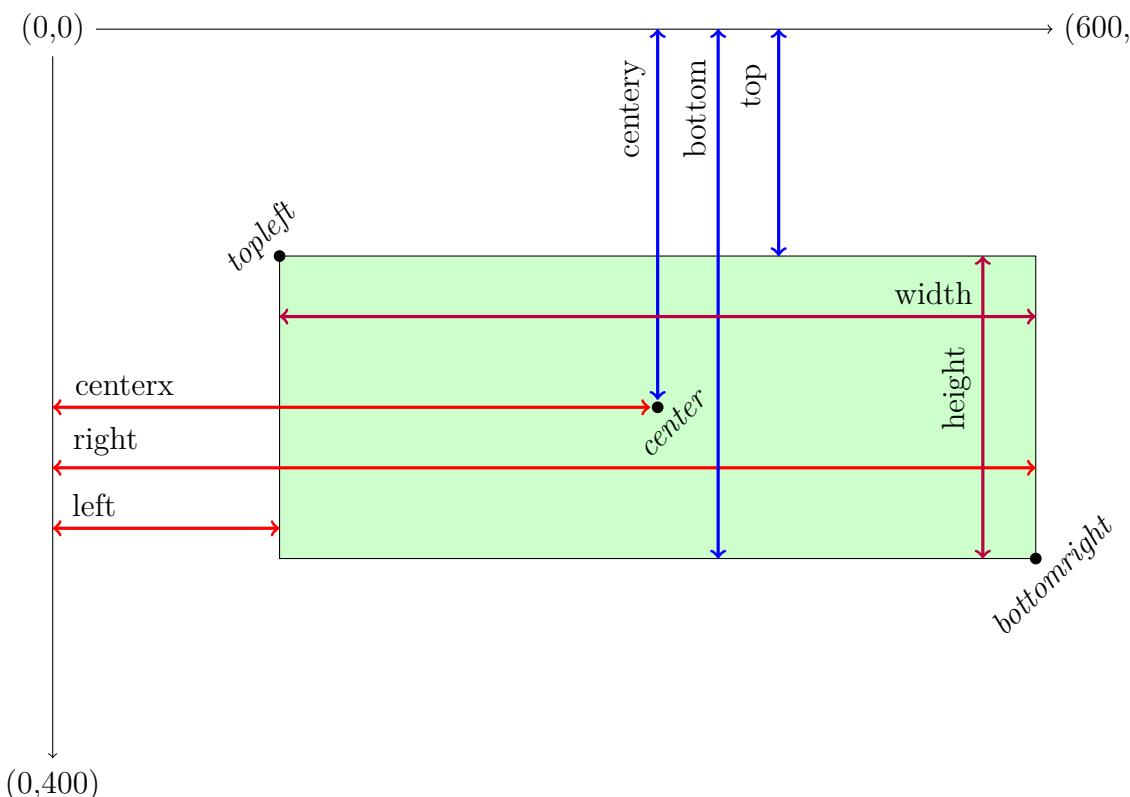


Abbildung 2.15: Elemente eines `Rect`-Objekts

In der Abbildung werden Strecken in normaler Schrift und Punkte in *kursiver Schrift* angegeben. Die Strecken sind eindimensional und die Punkte zweidimensional (x, y). Die Koordinate x ist dabei der horizontale und y der vertikale Abstand zum 0-Punkt des Koordinatensystems. Die Bedeutung der einzelnen Angaben sollte selbsterklärend sein. Der schöne Vorteil ist, dass die Angaben sich gegenseitig berechnen. Setze ich beispielsweise `topleft = (10,10)` und `width, height = 30, 40`, so werden alle anderen Angaben für mich ermittelt. Ich muss also nicht mehr den rechten Rand mit `left + width` ausrechnen; ich kann vielmehr sofort `right` verwenden. Auch oft nützlich ist die

Berechnung des Mittelpunktes `center` oder die entsprechenden Längen `centerx` und `centery`. Ändere ich nun das Zentrum durch `center = (100, 10)`, so verschieben sich alle anderen Angaben ebenfalls und müssen nicht von mir neu bestimmt werden – sehr praktisch.

Schauen wir uns dazu eine reduzierte Version des letzten Quelltextes an. In Quelltext 2.16 wird die `Rect`-Klasse schon verwendet. So werden beispielsweise in Zeile 7 die Fenstermaße in einem `Rect`-Objekt verwaltet. In den Zeilen Zeile 15, Zeile 22 und Zeile 23 können dadurch die Bildschirminformationen bequem und ohne eigene Berechnungen ausgelesen werden.

Quelltext 2.16: Bitmaps bewegen, Version 1.0

```

1  import os
2
3  import pygame
4
5
6  class Settings:
7      WINDOW = pygame.rect.Rect((0, 0), (600, 100))           # Rect-Objekt
8      FPS = 60
9
10
11 def main():
12     os.environ['SDL_VIDEO_WINDOW_POS'] = "10, 50"
13     pygame.init()
14
15     screen = pygame.display.set_mode(Settings.WINDOW.size)  # Zugriff auf ein Rect-Attribut
16     pygame.display.set_caption("Bewegung")
17     clock = pygame.time.Clock()
18
19     defender_image = pygame.image.load("images/defender01.png").convert_alpha()
20     defender_image = pygame.transform.scale(defender_image, (30, 30))
21     defender_rect = defender_image.get_rect()                 # Rect-Objekt
22     defender_rect.centerx = Settings.WINDOW.centerx         # Nicht nur left
23     defender_rect.bottom = Settings.WINDOW.height - 5       # Nicht nur top
24
25     running = True
26     while running:
27         # Events
28         for event in pygame.event.get():
29             if event.type == pygame.QUIT:
30                 running = False
31
32         # Update
33
34         # Draw
35         screen.fill("white")
36         screen.blit(defender_image, defender_rect)           # blit kann auch rect
37         pygame.display.flip()
38         clock.tick(Settings.FPS)
39
40     pygame.quit()
41
42
43 if __name__ == '__main__':
44     main()

```

`get_rect()` Für Surface-Objekte können wir sehr bequem mit `pygame.Surface.get_rect()` das `Rect`-Objekt erstellen lassen (Zeile 21). Die Positionierung kann nun leichter über die

Attribute erfolgen. Das Zentrum muss beispielsweise nicht mehr in die Berechnung einfließen, ich kann vielmehr das horizontale Zentrum direkt als halbe Fensterbreite festlegen (Zeile 22). Auch muss die vertikale Koordinate nicht mehr vom oberen Rand aus betrachtet werden, sondern ich kann viel intuitiver den Abstand des unteren Randes vom Bildschirmrand angeben (Zeile 23). Und als Sahnehäubchen kann das `Rect`-Objekt auch noch als Parameter der `blit()`-Funktion übergeben werden (Zeile 36).

blit()



Abbildung 2.16: Bitmaps bewegen, Version 1.0

Das Ergebnis ist unspektakulär (siehe Abbildung 2.16) und hat noch nichts mit Bewegung zu tun.

Bewegung wird in Spielen durch veränderte Positionen animiert. Soll das Raumschiff sich nach rechts bewegen, muss sich daher die horizontale Koordinate des Schiffs erhöhen. Welche horizontale Koordinate Sie dazu verwenden – `left`, `right` oder `centerx` –, können Sie von ihrer Spiellogik abhängig machen. In unserem Beispiel ist das egal; ich nehme daher `left`.

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

Allein diese kleine Ergänzung lässt unser Raumschiff nun nach rechts wandern. Die `+1` kodiert dabei zwei Informationen:

- **Richtung:** Hier ist das Vorzeichen `+`. Dadurch erhöht sich die Angabe `left` bei jedem Schleifendurchlauf; der linke Rand der Grafik wandert damit nach rechts. Wollte man nach links wandern, müsste das Vorzeichen ein `-` sein. Die horizontale Koordinate wird dadurch immer kleiner und nähert sich damit der 0. Völlig analog würde das Vorzeichen die Richtung in der Vertikalen steuern. Ein `+` würde die Grafik nach unten und ein `-` nach oben bewegen. Probieren Sie es aus!
- **Geschwindigkeit:** Die `1` legt fest, um welche Größenordnung sich `left` verändert. Je größer der Wert ist, desto größer sind die Sprünge zwischen den Frames; die Bewegung erscheint schneller.

Richtung

Geschwindigkeit

Quelltext 2.17: Bitmaps bewegen, Version 1.2

```

24  defender_speed = 2
25  defender_direction_h = +1
26
27  running = True
28  while running:
29      # Events
30
31  # Geschwindigkeit
32  # Richtung
33
34  # Update
35
36  # Blit
37
38  defender_rect.left = defender_rect.left + 1
39
40  # Update
41
42  # Blit
43
44  # Update
45
46  # Blit
47
48  # Update
49
50  # Blit
51
52  # Update
53
54  # Blit
55
56  # Update
57
58  # Blit
59
60  # Update
61
62  # Blit
63
64  # Update
65
66  # Blit
67
68  # Update
69
70  # Blit
71
72  # Update
73
74  # Blit
75
76  # Update
77
78  # Blit
79
80  # Update
81
82  # Blit
83
84  # Update
85
86  # Blit
87
88  # Update
89
90  # Blit
91
92  # Update
93
94  # Blit
95
96  # Update
97
98  # Blit
99
100 # Update
101
102 # Blit
103
104 # Update
105
106 # Blit
107
108 # Update
109
110 # Blit
111
112 # Update
113
114 # Blit
115
116 # Update
117
118 # Blit
119
120 # Update
121
122 # Blit
123
124 # Update
125
126 # Blit
127
128 # Update
129
130 # Blit
131
132 # Update
133
134 # Blit
135
136 # Update
137
138 # Blit
139
140 # Update
141
142 # Blit
143
144 # Update
145
146 # Blit
147
148 # Update
149
150 # Blit
151
152 # Update
153
154 # Blit
155
156 # Update
157
158 # Blit
159
160 # Update
161
162 # Blit
163
164 # Update
165
166 # Blit
167
168 # Update
169
170 # Blit
171
172 # Update
173
174 # Blit
175
176 # Update
177
178 # Blit
179
180 # Update
181
182 # Blit
183
184 # Update
185
186 # Blit
187
188 # Update
189
190 # Blit
191
192 # Update
193
194 # Blit
195
196 # Update
197
198 # Blit
199
200 # Update
201
202 # Blit
203
204 # Update
205
206 # Blit
207
208 # Update
209
210 # Blit
211
212 # Update
213
214 # Blit
215
216 # Update
217
218 # Blit
219
220 # Update
221
222 # Blit
223
224 # Update
225
226 # Blit
227
228 # Update
229
230 # Blit
231
232 # Update
233
234 # Blit
235
236 # Update
237
238 # Blit
239
240 # Update
241
242 # Blit
243
244 # Update
245
246 # Blit
247
248 # Update
249
250 # Blit
251
252 # Update
253
254 # Blit
255
256 # Update
257
258 # Blit
259
260 # Update
261
262 # Blit
263
264 # Update
265
266 # Blit
267
268 # Update
269
270 # Blit
271
272 # Update
273
274 # Blit
275
276 # Update
277
278 # Blit
279
280 # Update
281
282 # Blit
283
284 # Update
285
286 # Blit
287
288 # Update
289
290 # Blit
291
292 # Update
293
294 # Blit
295
296 # Update
297
298 # Blit
299
300 # Update
301
302 # Blit
303
304 # Update
305
306 # Blit
307
308 # Update
309
310 # Blit
311
312 # Update
313
314 # Blit
315
316 # Update
317
318 # Blit
319
320 # Update
321
322 # Blit
323
324 # Update
325
326 # Blit
327
328 # Update
329
330 # Blit
331
332 # Update
333
334 # Blit
335
336 # Update
337
338 # Blit
339
340 # Update
341
342 # Blit
343
344 # Update
345
346 # Blit
347
348 # Update
349
350 # Blit
351
352 # Update
353
354 # Blit
355
356 # Update
357
358 # Blit
359
360 # Update
361
362 # Blit
363
364 # Update
365
366 # Blit
367
368 # Update
369
370 # Blit
371
372 # Update
373
374 # Blit
375
376 # Update
377
378 # Blit
379
380 # Update
381
382 # Blit
383
384 # Update
385
386 # Blit
387
388 # Update
389
390 # Blit
391
392 # Update
393
394 # Blit
395
396 # Update
397
398 # Blit
399
400 # Update
401
402 # Blit
403
404 # Update
405
406 # Blit
407
408 # Update
409
410 # Blit
411
412 # Update
413
414 # Blit
415
416 # Update
417
418 # Blit
419
420 # Update
421
422 # Blit
423
424 # Update
425
426 # Blit
427
428 # Update
429
430 # Blit
431
432 # Update
433
434 # Blit
435
436 # Update
437
438 # Blit
439
440 # Update
441
442 # Blit
443
444 # Update
445
446 # Blit
447
448 # Update
449
450 # Blit
451
452 # Update
453
454 # Blit
455
456 # Update
457
458 # Blit
459
460 # Update
461
462 # Blit
463
464 # Update
465
466 # Blit
467
468 # Update
469
470 # Blit
471
472 # Update
473
474 # Blit
475
476 # Update
477
478 # Blit
479
480 # Update
481
482 # Blit
483
484 # Update
485
486 # Blit
487
488 # Update
489
490 # Blit
491
492 # Update
493
494 # Blit
495
496 # Update
497
498 # Blit
499
500 # Update
501
502 # Blit
503
504 # Update
505
506 # Blit
507
508 # Update
509
510 # Blit
511
512 # Update
513
514 # Blit
515
516 # Update
517
518 # Blit
519
520 # Update
521
522 # Blit
523
524 # Update
525
526 # Blit
527
528 # Update
529
530 # Blit
531
532 # Update
533
534 # Blit
535
536 # Update
537
538 # Blit
539
540 # Update
541
542 # Blit
543
544 # Update
545
546 # Blit
547
548 # Update
549
550 # Blit
551
552 # Update
553
554 # Blit
555
556 # Update
557
558 # Blit
559
560 # Update
561
562 # Blit
563
564 # Update
565
566 # Blit
567
568 # Update
569
570 # Blit
571
572 # Update
573
574 # Blit
575
576 # Update
577
578 # Blit
579
580 # Update
581
582 # Blit
583
584 # Update
585
586 # Blit
587
588 # Update
589
590 # Blit
591
592 # Update
593
594 # Blit
595
596 # Update
597
598 # Blit
599
600 # Update
601
602 # Blit
603
604 # Update
605
606 # Blit
607
608 # Update
609
610 # Blit
611
612 # Update
613
614 # Blit
615
616 # Update
617
618 # Blit
619
620 # Update
621
622 # Blit
623
624 # Update
625
626 # Blit
627
628 # Update
629
630 # Blit
631
632 # Update
633
634 # Blit
635
636 # Update
637
638 # Blit
639
640 # Update
641
642 # Blit
643
644 # Update
645
646 # Blit
647
648 # Update
649
650 # Blit
651
652 # Update
653
654 # Blit
655
656 # Update
657
658 # Blit
659
660 # Update
661
662 # Blit
663
664 # Update
665
666 # Blit
667
668 # Update
669
670 # Blit
671
672 # Update
673
674 # Blit
675
676 # Update
677
678 # Blit
679
680 # Update
681
682 # Blit
683
684 # Update
685
686 # Blit
687
688 # Update
689
690 # Blit
691
692 # Update
693
694 # Blit
695
696 # Update
697
698 # Blit
699
700 # Update
701
702 # Blit
703
704 # Update
705
706 # Blit
707
708 # Update
709
710 # Blit
711
712 # Update
713
714 # Blit
715
716 # Update
717
718 # Blit
719
720 # Update
721
722 # Blit
723
724 # Update
725
726 # Blit
727
728 # Update
729
730 # Blit
731
732 # Update
733
734 # Blit
735
736 # Update
737
738 # Blit
739
740 # Update
741
742 # Blit
743
744 # Update
745
746 # Blit
747
748 # Update
749
750 # Blit
751
752 # Update
753
754 # Blit
755
756 # Update
757
758 # Blit
759
760 # Update
761
762 # Blit
763
764 # Update
765
766 # Blit
767
768 # Update
769
770 # Blit
771
772 # Update
773
774 # Blit
775
776 # Update
777
778 # Blit
779
780 # Update
781
782 # Blit
783
784 # Update
785
786 # Blit
787
788 # Update
789
790 # Blit
791
792 # Update
793
794 # Blit
795
796 # Update
797
798 # Blit
799
800 # Update
801
802 # Blit
803
804 # Update
805
806 # Blit
807
808 # Update
809
810 # Blit
811
812 # Update
813
814 # Blit
815
816 # Update
817
818 # Blit
819
820 # Update
821
822 # Blit
823
824 # Update
825
826 # Blit
827
828 # Update
829
830 # Blit
831
832 # Update
833
834 # Blit
835
836 # Update
837
838 # Blit
839
840 # Update
841
842 # Blit
843
844 # Update
845
846 # Blit
847
848 # Update
849
850 # Blit
851
852 # Update
853
854 # Blit
855
856 # Update
857
858 # Blit
859
860 # Update
861
862 # Blit
863
864 # Update
865
866 # Blit
867
868 # Update
869
870 # Blit
871
872 # Update
873
874 # Blit
875
876 # Update
877
878 # Blit
879
880 # Update
881
882 # Blit
883
884 # Update
885
886 # Blit
887
888 # Update
889
890 # Blit
891
892 # Update
893
894 # Blit
895
896 # Update
897
898 # Blit
899
900 # Update
901
902 # Blit
903
904 # Update
905
906 # Blit
907
908 # Update
909
910 # Blit
911
912 # Update
913
914 # Blit
915
916 # Update
917
918 # Blit
919
920 # Update
921
922 # Blit
923
924 # Update
925
926 # Blit
927
928 # Update
929
930 # Blit
931
932 # Update
933
934 # Blit
935
936 # Update
937
938 # Blit
939
940 # Update
941
942 # Blit
943
944 # Update
945
946 # Blit
947
948 # Update
949
950 # Blit
951
952 # Update
953
954 # Blit
955
956 # Update
957
958 # Blit
959
960 # Update
961
962 # Blit
963
964 # Update
965
966 # Blit
967
968 # Update
969
970 # Blit
971
972 # Update
973
974 # Blit
975
976 # Update
977
978 # Blit
979
980 # Update
981
982 # Blit
983
984 # Update
985
986 # Blit
987
988 # Update
989
990 # Blit
991
992 # Update
993
994 # Blit
995
996 # Update
997
998 # Blit
999
999 # Update
1000
1000 # Blit
1001
1002 # Update
1003
1004 # Blit
1005
1006 # Update
1007
1008 # Blit
1009
10010 # Update
10011
10012 # Blit
10013
10014 # Update
10015
10016 # Blit
10017
10018 # Update
10019
10020 # Blit
10021
10022 # Update
10023
10024 # Blit
10025
10026 # Update
10027
10028 # Blit
10029
10030 # Update
10031
10032 # Blit
10033
10034 # Update
10035
10036 # Blit
10037
10038 # Update
10039
10040 # Blit
10041
10042 # Update
10043
10044 # Blit
10045
10046 # Update
10047
10048 # Blit
10049
10050 # Update
10051
10052 # Blit
10053
10054 # Update
10055
10056 # Blit
10057
10058 # Update
10059
10060 # Blit
10061
10062 # Update
10063
10064 # Blit
10065
10066 # Update
10067
10068 # Blit
10069
10070 # Update
10071
10072 # Blit
10073
10074 # Update
10075
10076 # Blit
10077
10078 # Update
10079
10080 # Blit
10081
10082 # Update
10083
10084 # Blit
10085
10086 # Update
10087
10088 # Blit
10089
10090 # Update
10091
10092 # Blit
10093
10094 # Update
10095
10096 # Blit
10097
10098 # Update
10099
100100 # Blit
100101
100102 # Update
100103
100104 # Blit
100105
100106 # Update
100107
100108 # Blit
100109
100110 # Update
100111
100112 # Blit
100113
100114 # Update
100115
100116 # Blit
100117
100118 # Update
100119
100120 # Blit
100121
100122 # Update
100123
100124 # Blit
100125
100126 # Update
100127
100128 # Blit
100129
100130 # Update
100131
100132 # Blit
100133
100134 # Update
100135
100136 # Blit
100137
100138 # Update
100139
100140 # Blit
100141
100142 # Update
100143
100144 # Blit
100145
100146 # Update
100147
100148 # Blit
100149
100150 # Update
100151
100152 # Blit
100153
100154 # Update
100155
100156 # Blit
100157
100158 # Update
100159
100160 # Blit
100161
100162 # Update
100163
100164 # Blit
100165
100166 # Update
100167
100168 # Blit
100169
100170 # Update
100171
100172 # Blit
100173
100174 # Update
100175
100176 # Blit
100177
100178 # Update
100179
100180 # Blit
100181
100182 # Update
100183
100184 # Blit
100185
100186 # Update
100187
100188 # Blit
100189
100190 # Update
100191
100192 # Blit
100193
100194 # Update
100195
100196 # Blit
100197
100198 # Update
100199
100200 # Blit
100201
100202 # Update
100203
100204 # Blit
100205
100206 # Update
100207
100208 # Blit
100209
100210 # Update
100211
100212 # Blit
100213
100214 # Update
100215
100216 # Blit
100217
100218 # Update
100219
100220 # Blit
100221
100222 # Update
100223
100224 # Blit
100225
100226 # Update
100227
100228 # Blit
100229
100230 # Update
100231
100232 # Blit
100233
100234 # Update
100235
100236 # Blit
100237
100238 # Update
100239
100240 # Blit
100241
100242 # Update
100243
100244 # Blit
100245
100246 # Update
100247
100248 # Blit
100249
100250 # Update
100251
100252 # Blit
100253
100254 # Update
100255
100256 # Blit
100257
100258 # Update
100259
100260 # Blit
100261
100262 # Update
100263
100264 # Blit
100265
100266 # Update
100267
100268 # Blit
100269
100270 # Update
100271
100272 # Blit
100273
100274 # Update
100275
100276 # Blit
100277
100278 # Update
100279
100280 # Blit
100281
100282 # Update
100283
100284 # Blit
100285
100286 # Update
100287
100288 # Blit
100289
100290 # Update
100291
100292 # Blit
100293
100294 # Update
100295
100296 # Blit
100297
100298 # Update
100299
100300 # Blit
100301
100302 # Update
100303
100304 # Blit
100305
100306 # Update
100307
100308 # Blit
100309
100310 # Update
100311
100312 # Blit
100313
100314 # Update
100315
100316 # Blit
100317
100318 # Update
100319
100320 # Blit
100321
100322 # Update
100323
100324 # Blit
100325
100326 # Update
100327
100328 # Blit
100329
100330 # Update
100331
100332 # Blit
100333
100334 # Update
100335
100336 # Blit
100337
100338 # Update
100339
100340 # Blit
100341
100342 # Update
100343
100344 # Blit
100345
100346 # Update
100347
100348 # Blit
100349
100350 # Update
100351
100352 # Blit
100353
100354 # Update
100355
100356 # Blit
100357
100358 # Update
100359
100360 # Blit
100361
100362 # Update
100363
100364 # Blit
100365
100366 # Update
100367
100368 # Blit
100369
100370 # Update
100371
100372 # Blit
100373
100374 # Update
100375
100376 # Blit
100377
100378 # Update
100379
100380 # Blit
100381
100382 # Update
100383
100384 # Blit
100385
100386 # Update
100387
100388 # Blit
100389
100390 # Update
100391
100392 # Blit
100393
100394 # Update
100395
100396 # Blit
100397
100398 # Update
100399
100400 # Blit
100401
100402 # Update
100403
100404 # Blit
100405
100406 # Update
100407
100408 # Blit
100409
100410 # Update
100411
100412 # Blit
100413
100414 # Update
100415
100416 # Blit
100417
100418 # Update
100419
100420 # Blit
100421
100422 # Update
100423
100424 # Blit
100425
100426 # Update
100427
100428 # Blit
100429
100430 # Update
100431
100432 # Blit
100433
100434 # Update
100435
100436 # Blit
100437
100438 # Update
100439
100440 # Blit
100441
100442 # Update
100443
100444 # Blit
100445
100446 # Update
100447
100448 # Blit
100449
100450 # Update
100451
100452 # Blit
100453
100454 # Update
100455
100456 # Blit
100457
100458 # Update
100459
100460 # Blit
100461
100462 # Update
100463
100464 # Blit
100465
100466 # Update
100467
100468 # Blit
100469
100470 # Update
100471
100472 # Blit
100473
100474 # Update
100475
100476 # Blit
100477
100478 # Update
100479
100480 # Blit
100481
100482 # Update
100483
100484 # Blit
100485
100486 # Update
100487
100488 # Blit
100489
100490 # Update
100491
100492 # Blit
100493
100494 # Update
100495
100496 # Blit
100497
100498 # Update
100499
100500 # Blit
100501
100502 # Update
100503
100504 # Blit
100505
100506 # Update
100507
100508 # Blit
100509
100510 # Update
100511
100512 # Blit
100513
100514 # Update
100515
100516 # Blit
100517
100518 # Update
100519
100520 # Blit
100521
100522 # Update
100523
100524 # Blit
100525
100526 # Update
100527
100528 # Blit
100529
100530 # Update
100531
100532 # Blit
100533
100534 # Update
100535
100536 # Blit
100537
100538 # Update
100539
100540 # Blit
100541
100542 # Update
100543
100544 # Blit
100545
100546 # Update
100547
100548 # Blit
100549
100550 # Update
100551
100552 # Blit
100553
100554 # Update
100555
100556 # Blit
100557
100558 # Update
100559
100560 # Blit
100561
100562 # Update
100563
100564 # Blit
100565
100566 # Update
100567
100568 # Blit
100569
100570 # Update
100571
100572 # Blit
100573
100574 # Update
100575
100576 # Blit
100577
100578 # Update
100579
100580 # Blit
100581
100582 # Update
100583
100584 # Blit
100585
100586 # Update
100587
100588 # Blit
100589
100590 # Update
100591
100592 # Blit
100593
100594 # Update
100595
100596 # Blit
100597
100598 # Update
100599
100600 # Blit
100601
100602 # Update
100603
100604 # Blit
100605
100606 # Update
100607
100608 # Blit
100609
100610
```

```

30     for event in pygame.event.get():
31         if event.type == pygame.QUIT:
32             running = False
33
34         # Update
35         defender_rect.left += defender_direction_h * defender_speed # Flexible
36
37         # Draw
38         screen.fill("white")
39         screen.blit(defender_image, defender_rect)
40         pygame.display.flip()

```

Diese beiden Informationen werden nun in Quelltext 2.17 auf der vorherigen Seite dazu genutzt, die Bewegung erheblich flexibler zu gestalten. In Zeile 24 die Geschwindigkeit nun durch die Variable `defender_speed` repräsentiert. So könnten wir im Laufe des Spiels die Geschwindigkeit dynamisch gestalten, z.B. bei einer Beschleunigung durch Raketentreibstoffausstoß.

Die Richtung wird in Zeile 25 ebenfalls in einer Variablen abgelegt: `defender_direction`. Derzeit ist sie positiv, aber wir werden schon bald sehen, dass wir diese auch für Richtungswechsel nutzen können.

Beide Informationen können nun in Zeile 35 zur Berechnung der neuen horizontalen Position genutzt werden.

Wenn Sie das Programm laufen lassen, verabschiedet sich der Verteidiger nach einiger Zeit und verschwindet hinter dem rechten Bildschirmrand und wird nicht mehr gesehen. Nutzen wir nun unser Rechteck zu einer ersten einfachen Kollisionsprüfung. Ich möchte, dass das Raumschiff von den Rändern *abprallt* und die Richtung wechselt.

Quelltext 2.18: Bitmaps bewegen, Version 1.3

```

35     defender_rect.left += defender_direction_h * defender_speed
36     if defender_rect.right >= Settings.WINDOW.right:      # Rechter Rand erreicht
37         defender_direction_h *= -1                         # Richtungswechsel
38     elif defender_rect.left <= Settings.WINDOW.left:    # Linker Rand erreicht
39         defender_direction_h *= -1

```

Ich hoffe, dass Sie die Idee hinter dem Code erkennen. Nach Berechnung der neuen horizontalen Position, wird in Zeile 36 überprüft, ob der neue rechte Rand des Bitmaps die rechte Bildschirmseite erreicht oder überschreitet. Wenn ja, dann wird einfach das Vorzeichen der Richtungsvariable vertauscht! Analog klappt das beim Erreichen des linken Bildschirmrandes.

Probieren Sie doch mal aus, das Ganze mit einer vertikalen Bewegung zu kombinieren.

Ein Problem habe ich noch: In Zeile 35 wird die neue Position dem `Rect`-Objekt zugewiesen, obwohl sie vielleicht schon über den Rand ragt. Bei einer Geschwindigkeit von 1 oder 2 mag das nicht so ins Auge fallen, aber wenn wir die Geschwindigkeit auf die Raumschiffbreite einstellen, wird das Problem offensichtlich (setzen Sie kurzfristig mal `Settings.FPS = 5`, damit man was sieht). Das Raumschiff verlässt zur Hälfte den Bildschirm.

Richtungswechsel

Wir sollten somit die neue Position überprüfen und erst dann diese dem `Rect`-Objekt `defender_rect` zuweisen. Führen wir in diesem Zusammenhang eine recht nützliche Methode der `Rect`-Klasse ein: `pygame.rect.Rect.move()`.

`move()`

Quelltext 2.19: Bitmaps bewegen, Version 1.4

```

35     newpos = defender_rect.move(defender_direction_h * defender_speed, 0)  #
36     # Testposition
37     if newpos.right >= Settings.WINDOW.right:
38         defender_direction_h *= -1
39     elif newpos.left <= Settings.WINDOW.left:
40         defender_direction_h *= -1
41     else:
42         defender_rect = newpos  # Übernehme neue Position

```

Die neue Funktion taucht in Zeile 35 zum ersten Mal auf. Sie hat zwei Parameter. Mit dem ersten wird die Verschiebung der horizontalen Koordinate angegeben und mit der zweiten die vertikale Verschiebung. Da wir keine Höhenposition ändern wollen, ist dieser Parameter in unserem Beispiel konstant 0. Als Rückgabe liefert die Funktion ein neues `Rect`-Objekt mit den neuen Positionsangaben. Dieses speichern wir in `newpos` zwischen.

Die nachfolgenden Kollisionsprüfungen werden dann mit dem `newpos`-Rechteck durchgeführt. Bei einer Kollision werden wie eben die Richtungswerte verändert. Falls keine Kollision mit dem Rand vorliegt, wird `newpos` zu unserem neuen Rechteck für den Verteidiger (Zeile 41).

Wenn Sie jetzt das Programm ausführen, wird die Position bei einer Kollision eben nicht verändert, sondern erst im nächsten Frame.



Abbildung 2.17: Der Verteidiger bewegt sich und prallt ab

2.4.2 Geschwindigkeiten normalisieren (*deltatime*)

Die Bewegung ist derzeit nicht nur von `defender_speed` abhängig, sondern auch von der Framerate `Settings.FPS`. Um diese Abhängigkeit zu verdeutlichen, habe ich den vorherigen Quelltext für ein kleines Experiment umgebaut (siehe Quelltext 2.20 auf der nächsten Seite).

In Zeile 9 sehen Sie die unterschiedlichen Frameraten, mit denen das Experiment durchgeführt wurde. In der Zeile davor werden die Fenstermaße so eingestellt, dass das Fenster

hoch und schmal ist, und in der Zeile darunter wird die absolute Anzahl der Millisekunden angegeben, die das Raumschiff nach oben steigt.

Zeile 29 merkt sich die Zeit, wann das Aufsteigen des Raumschiffs begonnen hat. Dazu liefert mir die Funktion `pygame.time.get_ticks()` die Anzahl der Millisekunden seit dem Aufruf von `pygame.init()`; also beispielsweise 5 ms.

Innerhalb der Hauptprogrammschleife wandert das Raumschiff nun pro Frame eine gewisse Anzahl von Pixel nach oben. Dabei wird die neue Position dadurch ermittelt, dass auf die `top`-Koordinate der alten Position das Produkt aus Richtung und Geschwindigkeit addiert wird (Zeile 40) – also nix Neues an dieser Stelle.

Nach einer festen Zeitspanne (`Settings.LIMIT`, hier 500 ms) wird die Richtung in `defender_direction` auf 0 gesetzt, so dass die Bewegung stoppt. Dazu wird in Zeile 32 abgefragt, ob die aktuelle Anzahl von Millisekunden seit dem Start des Programmes größer als `start_time` plus `Settings.LIMIT` ist. Oder in Zahlen: Beim ersten Schleifendurchlauf (Frame 1) stünde dort beispielsweise die Abfrage *Sind 17 ms größer als 5 ms + 500 ms*. Die Antwort ist *Nein*, so dass das Raumschiff sich nach oben bewegt. Bei Frame 61 stünde dort die Abfrage *Sind 508 ms größer als 5 ms + 500 ms*. Nun ist die Antwort *Ja* und die Richtungsvariable wird deshalb auf 0 gesetzt, die Bewegung stoppt.

Quelltext 2.20: Nicht normalisierte Bewegung

```

7  class Settings:
8      WINDOW = pygame.Rect((0, 0), (120, 650))
9      FPS = 600 # 10 30 60 120 240 300 600
10     LIMIT = 60
11
12
13 def main():
14     os.environ['SDL_VIDEO_WINDOW_POS'] = "10,50"
15     pygame.init()
16
17     screen = pygame.display.set_mode(Settings.WINDOW.size)
18     pygame.display.set_caption("Bewegung")
19     clock = pygame.time.Clock()
20
21     defender_image = pygame.image.load("images/defender01.png").convert_alpha()
22     defender_image = pygame.transform.scale(defender_image, (30, 30))
23     defender_rect = defender_image.get_rect()
24     defender_rect.centerx = Settings.WINDOW.centerx
25     defender_rect.bottom = Settings.WINDOW.bottom - 5
26     defender_speed = 2
27     defender_direction_v = -1
28
29     start_time = pygame.time.get_ticks() # Startzeit der Bewegung
30     running = True
31     while running:
32         if pygame.time.get_ticks() > start_time + Settings.LIMIT: # Fertig?
33             defender_speed = 0
34         # Events
35         for event in pygame.event.get():
36             if event.type == pygame.QUIT:
37                 running = False
38
39         # Update
40         defender_rect.top += defender_direction_v * defender_speed # Neue Höhe

```

```

41     if defender_rect.bottom >= Settings.WINDOW.bottom:
42         defender_direction_v *= -1
43     elif defender_rect.top <= 0:
44         defender_direction_v *= -1
45
46     # Draw
47     screen.fill("white")
48     screen.blit(defender_image, defender_rect)
49     pygame.display.flip()
50     clock.tick(Settings.FPS)
51     print(f"top={defender_rect.top}")
52
53     pygame.quit()

```

In Abbildung 2.18 auf der nächsten Seite können Sie Bildschirmfotos der Strecken sehen, die das Raumschiff nach einer halben Sekunde zurückgelegt hat. In allen Experimenten blieb die Geschwindigkeit `defender_speed` immer gleich – nämlich 2. Nur die Framerate hat sich erhöht.

Wie kommen diese unterschiedlichen Höhen zustande? Soll doch eigentlich nur `defender_speed` die Geschwindigkeit definieren. In Tabelle 2.1 wird der Zusammenhang hoffentlich deutlich. In der ersten Spalte ist die Geschwindigkeit eines Objektes zu sehen; in unserem Beispiel ist dies die Variable `defender_speed`. Dieser Wert gibt an, um wie viele Pixel pro Frame das Objekt bewegt wird; dieser Wert wird nicht verändert. Die zweite Spalte gibt die Framerate an, also die Anzahl der Frames pro Sekunde. In unserem Beispiel ist dieser Wert in `Settings.FPS` definiert. Die Dauer der Bewegung ist in der dritten Spalte zu sehen. Wir haben eine Dauer von 500 ms also 0.5 s. In unserem Beispiels liegt der Wert in `Settings.LIMIT` und ist ebenfalls für alle Experimente gleich.

In der letzten Spalte steht die errechnete Wegstrecke in Pixel, die das bewegliche Objekt dann zurückgelegt hat. Nun ist Zusammenhang klar: Da wir die Hauptprogrammschleife wegen der unterschiedlichen Framerate unterschiedlich oft wiederholen, wird bei konstanter Zeit eine unterschiedliche Strecke zurückgelegt.

Tabelle 2.1: Strecke bei nicht normalisierter Geschwindigkeit

$$\text{Geschw. } \left(\frac{px}{f} \right) * \text{fps } \left(\frac{f}{s} \right) * \text{Zeit } (s) = \text{Strecke } (px)$$

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

Was wir also brauchen, ist eine Mechanismus, der die Framerate wieder rausrechnet. Dieser Faktor müsste so gebaut sein, dass er mit der Framerate multipliziert immer eine 1 als Ergebnis auswirkt. Damit würde die Framerate im Gesamtprodukt wie eine Multiplikation mit 1 wirken, also keinen Einfluss mehr haben. Der naheliegende Weg wäre das Inverse der Framerate zu nehmen, also $\frac{1}{fps}$. Dieser Korrekturwert wird *deltatime* (*dt*)

deltatime

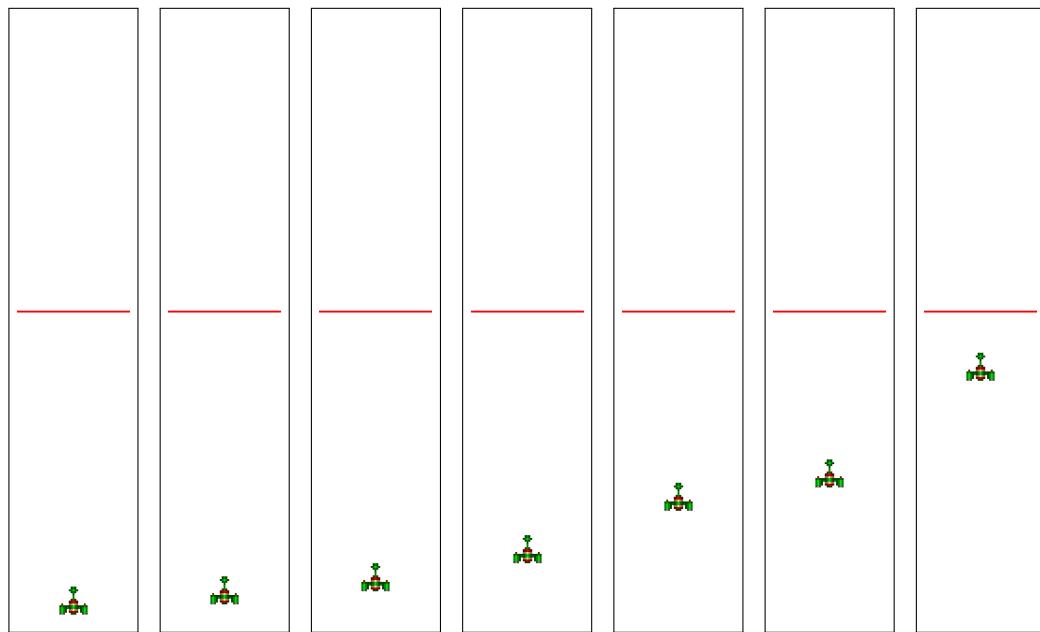


Abbildung 2.18: Nicht normalisierte Bewegung bei gleicher Geschwindigkeit aber unterschiedlichen Frameraten (von links nach rechts: 10, 30, 60, 120, 240, 300, 600)

genannt. Die Berechnung würde dann beispielsweise wie in Tabelle 2.2 aussehen. Die zweite und die dritte Spalte heben sich auf, so dass die Strecke immer – unabhängig von der gewählten Framerate – gleich bleibt.

Tabelle 2.2: Strecke bei normalisierter Geschwindigkeit

$$\text{Geschw. } \left(\frac{px}{s}\right) * \text{fps } \left(\frac{f}{s}\right) * \text{dt } \left(\frac{s}{f}\right) * \text{Zeit } (s) = \text{Strecke } (px)$$

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 diesem Zusammenhang fällt auf, dass die Strecke erschreckend kurz ist: Nur 1 px pro Sekunde. Bitte beachten Sie, dass sich auch die Maßeinheit der ersten Spalte geändert hat. Die Geschwindigkeit gibt nun nicht mehr die Anzahl der Pixel pro Frame, sondern pro Sekunde an! Setzen wir daher eine andere Geschwindigkeit fest; ich habe mich passend zu unserer Fenstergröße mal für 600 px/s entschieden. Nach einer Sekunde kommt unser Raumschiff also oben an.

In Tabelle 2.3 auf der nächsten Seite habe ich mal berechnet, welche Endposition (.top) wir nach einer halben Sekunde erwarten können. In der linken Hälfte wird die Weg-

strecke ausgerechnet. Überraschung: Sie beträgt immer 300 *px*. Von der Fensterhöhe (`WINDOW.height`) müssen wir diese Wegstrecke abziehen. Ebenso die Höhe unseres Raumschiffs (30 *px*) und den kleinen Offset von 5 *px*, da wir unser Raumschiff nicht ganz unten Rand starten lassen wollten. Wir erwarten also, dass unser Raumschiff nach einer halben Sekunde die in Tabelle 2.3 berechnete Endposition einnimmt.

Tabelle 2.3: Pixelkoordinaten bei normalisierter Geschwindigkeit

Geschw. * fps * dt * Zeit = Strecke → `WINDOW.height` - Höhe - Offset = `.top`

$600 * 10 * \frac{1}{10} * 0.5 = 300 \rightarrow 650-300$	-	30	-	5 = 315
$600 * 30 * \frac{1}{30} * 0.5 = 300 \rightarrow 650-300$	-	30	-	5 = 315
$600 * 60 * \frac{1}{60} * 0.5 = 300 \rightarrow 650-300$	-	30	-	5 = 315
$600 * 120 * \frac{1}{120} * 0.5 = 300 \rightarrow 650-300$	-	30	-	5 = 315
$600 * 240 * \frac{1}{240} * 0.5 = 300 \rightarrow 650-300$	-	30	-	5 = 315
$600 * 300 * \frac{1}{300} * 0.5 = 300 \rightarrow 650-300$	-	30	-	5 = 315

Tabelle 2.3 sieht zwar kompliziert aus, die Umsetzung ist aber überraschend einfach. In Zeile 11 wird der Korrekturwert – wie oben besprochen – als Inverses von der Frame-rate definiert. Die Geschwindigkeit wird von 2 auf 600 in Zeile 27 angepasst und in Zeile 41 wird der Korrekturwert `DELTATIME` in Berechnung als Faktor eingebaut. Das war's; in Abbildung 2.22 auf Seite 43 können wir das *perfekte* Ergebnis auf einem meiner langsameren Rechner bewundern.

Quelltext 2.21: Normalisierte Bewegung mit 1/fps

```

7  class Settings:
8      WINDOW = pygame.rect.Rect((0, 0), (120, 650))
9      FPS = 300
10     LIMIT = 500
11     DELTATIME = 1.0/FPS
12
13
14 def main():
15     os.environ['SDL_VIDEO_WINDOW_POS'] = "10,150"
16     pygame.init()
17
18     screen = pygame.display.set_mode(Settings.WINDOW.size)
19     pygame.display.set_caption("Bewegung")
20     clock = pygame.time.Clock()
21
22     defender_image = pygame.image.load("images/defender01.png").convert_alpha()
23     defender_image = pygame.transform.scale(defender_image, (30, 30))
24     defender_rect = defender_image.get_rect()
25     defender_rect.centerx = Settings.WINDOW.centerx
26     defender_rect.bottom = Settings.WINDOW.bottom - 5
27     defender_speed = 600
28     defender_direction_v = -1
29
30     start_time = pygame.time.get_ticks()
31     running = True
32     while running:
33         if pygame.time.get_ticks() > start_time + Settings.LIMIT:
34             defender_speed = 0

```

```

35      # Events
36      for event in pygame.event.get():
37          if event.type == pygame.QUIT:
38              running = False
39
40      # Update
41      defender_rect.top += defender_direction_v * defender_speed * Settings.DELTATIME # 
42      if defender_rect.bottom >= Settings.WINDOW.bottom:
43          defender_direction_v *= -1
44      elif defender_rect.top <= 0:
45          defender_direction_v *= -1
46
47      # Draw
48      screen.fill("white")
49      pygame.draw.line(screen, "red", (0, 315), (Settings.WINDOW.width, 315), 2)
50      screen.blit(defender_image, defender_rect)
51      pygame.display.flip()
52      clock.tick(Settings.FPS)
53      print(f"top={defender_rect.top}")
54
55      pygame.quit()

```

Zwei Probleme verursachen diese Fehler:

- Rundungsfehler: Eigentlich sollte die Multiplikation der Framerate mit der Deltatime immer 1.0 ergeben. Das passiert leider aber nicht. Bei der Berechnung der Deltatime wird wegen des Formats einer **Fließkommazahl** ein Wert nahe des tatsächlichen Wertes abgespeichert; also beispielsweise bei $\frac{1.0}{30.0}$ nicht $0.0\bar{3}$, sondern 0.0333333333333330. Dieser Rundungsfehler addiert sich im Laufe der Zeit zu wahrnehmbaren Größen auf.
- Falsches Verständnis von *fps*: Die Framerate definiert nicht, dass *immer* die Hauptprogrammschleife beispielsweise 60 mal in der Sekunde durchlaufen wird, sondern dass sie *maximal* 60 mal in der Sekunde durchlaufen wird. Nimmt die Spielogik oder das Zeichnen der Oberfläche mehr Zeit in Anspruch als $1/60\text{ s}$, so wird mindestens ein Frame übersprungen. Dies tritt auch auf, wenn der Rechner durch andere Operationen (z.B. einer Cloud-Sync) Performance verliert.

Rundungsfehler

tick()

Das erste Problem können wir nicht ohne erheblichen Performanceverlust lösen und wird daher nicht weiter betrachtet. Das zweite Problem schon. Wir brauchen anstelle der festen Deltatime einen Wert, der sich aus der tatsächlichen Dauer eines Frames ergibt. Die Methode `pygame.clock.tick()` in Zeile 52 liefert mir nämlich eine gute Schätzung über die Laufzeit des Frames. Dieses Feature ist zum Glück schon eingebaut und kann daher einfach so verwendet werden (siehe Quelltext 2.22 auf der nächsten Seite). Das Ergebnis in Abbildung 2.23 auf Seite 43 ist zwar besser aber trotzdem noch nicht befriedigend :-(. In Abbildung 2.19 auf der nächsten Seite können Sie sehen, dass die rote Linie mehr oder weniger um die grüne herumtanzt und keine eindeutige Tendenz sichtbar ist.

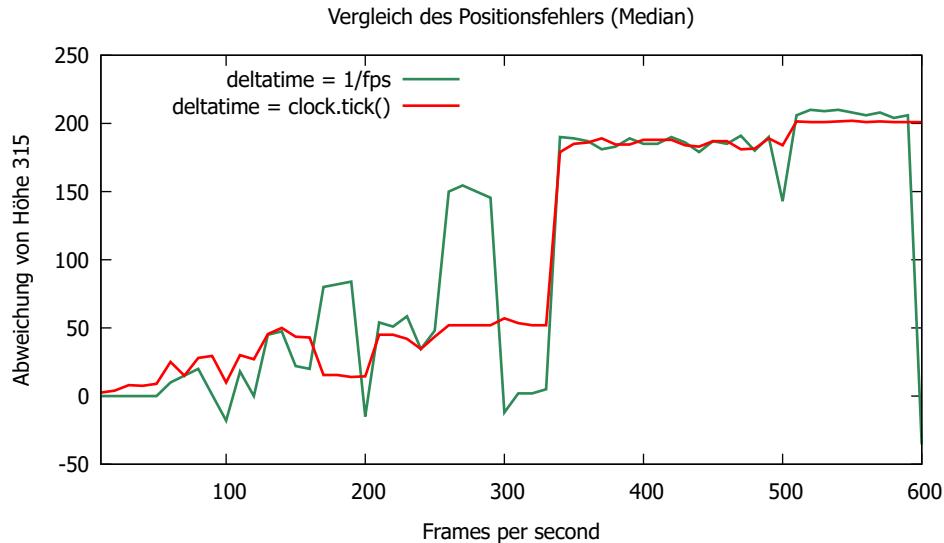


Abbildung 2.19: Vergleich des Positionsfehlers von $1/fps$ und `pygame.clock.tick()`

Quelltext 2.22: Normalisierte Bewegung mit `pygame.clock.tick()`

```

47 # Draw
48 screen.fill("white")
49 pygame.draw.line(screen, "red", (0, 315), (Settings.WINDOW.width, 315), 2)
50 screen.blit(defender_image, defender_rect)
51 pygame.display.flip()
52 Settings.DELTATIME = clock.tick(Settings.FPS) / 1000.0      # Korrekturwert ermitteln

```

Ursache ist ein Problem, welches wir hätten sofort lösen müssen. Bei der Zuweisung in Zeile 41 in Quelltext 2.21 auf Seite 37 steht auf der rechten Seite eine Fließkommazahl und auf der linken ein **Ganzzahl**. Dies führt dazu, dass die Nachkommastellen bei jedem Schleifendurchlauf abgeschnitten werden. Würde beispielsweise in jedem Frame das Raumschiff sich um 5.8 px bewegen müssen, entstünden diese Werte:

Tabelle 2.4: Fehlerfortpflanzung

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

Seit kurzem gibt es in Pygame eine Variante von `Rect`, nämlich `FRect`. Dort werden alle Werte als `float` abgespeichert, so dass Nachkommastellen nicht mehr abgeschnitten werden.¹

`FRect`

¹ Alternativ müssten wir die Positionswerte unabhängig vom `Rect`-Objekt in einer zusätzlichen `float`-Variablen abspeichern, damit die Nachkommastellen erhalten bleiben, z.B. in ein `pygame.math.Vector2`-Objekt.

Quelltext 2.23: Normalisierte Bewegung mit Positionsangaben in float

```

14 def main():
15     os.environ['SDL_VIDEO_WINDOW_POS'] = "10, 50"
16     pygame.init()
17
18     screen = pygame.display.set_mode(Settings.WINDOW.size)
19     pygame.display.set_caption("Bewegung")
20     clock = pygame.time.Clock()
21
22     defender_image = pygame.image.load("images/defender01.png").convert_alpha()
23     defender_image = pygame.transform.scale(defender_image, (30, 30))
24     defender_rect = pygame.Rect(defender_image.get_rect())      # float
25     defender_rect.centerx = Settings.WINDOW.centerx
26     defender_rect.bottom = Settings.WINDOW.bottom - 5
27     defender_speed = 600
28     defender_direction_v = -1
29
30     start_time = pygame.time.get_ticks()
31     running = True
32     while running:
33         if pygame.time.get_ticks() > start_time + Settings.LIMIT:
34             defender_speed = 0
35
36         # Events
37         for event in pygame.event.get():
38             if event.type == pygame.QUIT:
39                 running = False
40
41         # Update
42         defender_rect.top += defender_direction_v * defender_speed * Settings.DELTATIME
43         if defender_rect.bottom >= Settings.WINDOW.bottom:
44             defender_direction_v *= -1
45         elif defender_rect.top <= 0:
46             defender_direction_v *= -1
47
48         # Draw
49         screen.fill("white")
50         pygame.draw.line(screen, "red", (0, 315), (Settings.WINDOW.width, 315), 2)
51         screen.blit(defender_image, defender_rect)
52         pygame.display.flip()
53         Settings.DELTATIME = clock.tick(Settings.FPS) / 1000.0
54         print(f"top={defender_rect.top}")
55
56     pygame.quit()

```

In Abbildung 2.24 auf Seite 44 können wir sehen, dass das Ergebnis schon deutlich besser geworden ist. Auch hat sich die Abweichung von optimalen Wert 315 dramatisch verbessert. In Abbildung 2.20 auf der nächsten Seite wird der Unterschied sichtbar gemacht.

Es gibt aber noch eine weitere Fehlerquelle: `pygame.clock.tick()` liefert mir nicht genug Nachkommastellen. Bei langen Laufzeiten multiplizieren sich diese fehlenden Angaben nach vorne und führen wiederum zu Fehlern. Es gibt bessere Python-Funktionen zur Ermittlung von Laufzeiten.

`time()`

In Zeile 32 von Quelltext 2.24 auf Seite 42 wird mit Hilfe von `time.time()` die Anzahl der Sekunden nach dem 01.01.1970 als eine Fließkommazahl (float) zurückgeliefert. Die Nachkommastellen geben dabei die Sekundenbruchteile an. Diese Messung ist genauer als die durch `pygame.clock.tick()` und liefert mir je nach Zeitmessungsmöglichkeiten der Rechnerarchitektur und des Betriebssystems mehr Nachkommastellen – bis in den Nanosekundenbereich.

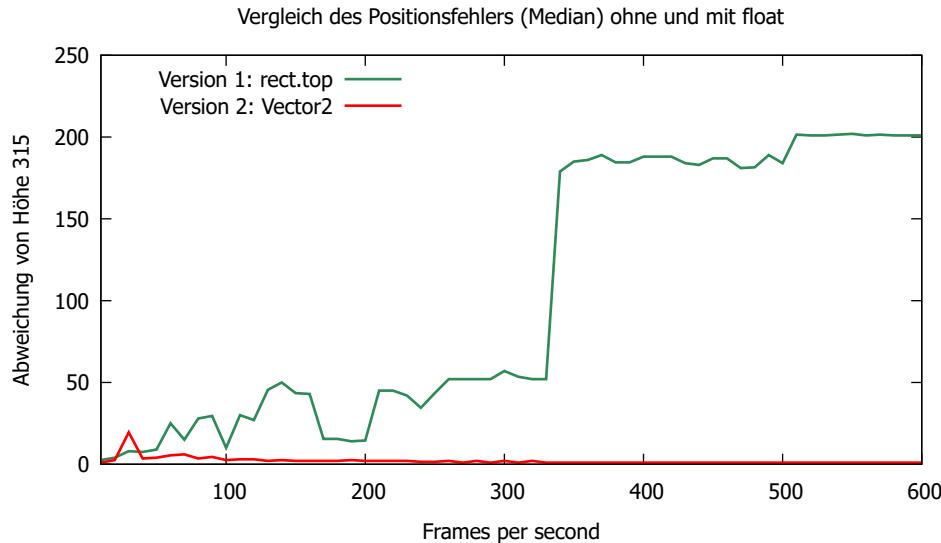


Abbildung 2.20: Vergleich des Positionsfehlers von Rect und FRect

In Zeile 55 wird nach Ablauf eines Frame die aktuelle Zeit gemessen und in der Zeile danach der Zeitverbrauch ermittelt. Dieser ist dann die tatsächliche *deltatime* des Frames, nun mit mehr Nachkommastellen. Anschließend wird in Zeile 57 die neue Startzeit des nächsten Frames festgehalten, um nach dem nächsten Frame wieder den Zeitverbrauch ermitteln zu können.

Abbildung 2.25 auf Seite 44 zeigt uns, dass die Zielpositionen bei allen Frameraten nahezu perfekt getroffen wurden. Der Vergleich der Positionsfehler in Abbildung 2.21 auf der nächsten Seite lässt aber keine eindeutige Bewertung zu. Ich denke mir aber, dass hier Experimente mit deutlich längeren Laufzeiten einen Unterschied erkennbar machen würden. Mit dem Restfehler müssen – und können – wir leben.

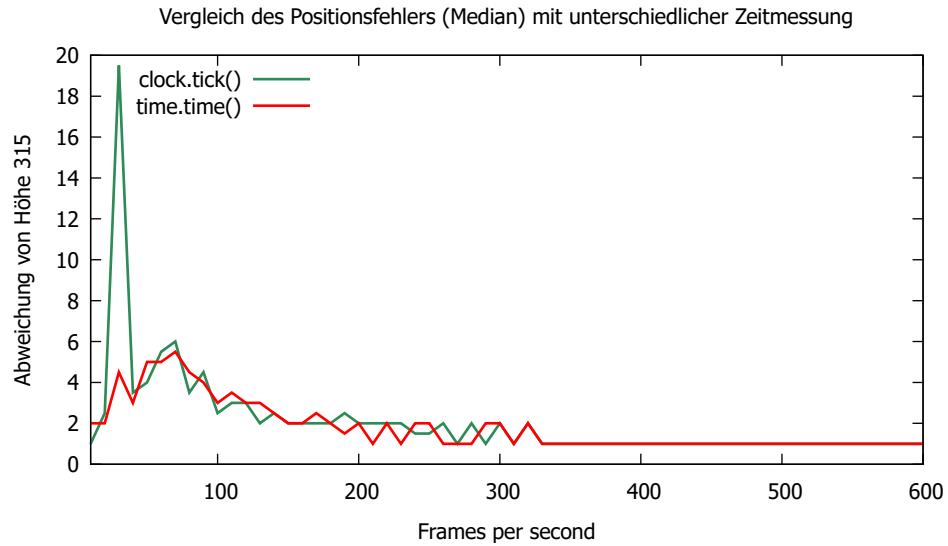


Abbildung 2.21: Vergleich der Positionsfehler mit unterschiedlichen Genauigkeiten

Quelltext 2.24: Normalisierte Bewegung mit `time.time()`

```

15  def main():
16      os.environ['SDL_VIDEO_WINDOW_POS'] = "10, 50"
17      pygame.init()
18
19      screen = pygame.display.set_mode(Settings.WINDOW.size)
20      pygame.display.set_caption("Bewegung")
21      clock = pygame.time.Clock()
22
23      defender_image = pygame.image.load("images/defender01.png").convert_alpha()
24      defender_image = pygame.transform.scale(defender_image, (30, 30))
25      defender_rect = pygame.Rect(defender_image.get_rect())
26      defender_rect.centerx = Settings.WINDOW.centerx
27      defender_rect.bottom = Settings.WINDOW.height - 5
28      defender_speed = 600
29      defender_direction_v = -1
30
31      start_time = pygame.time.get_ticks()
32      time_previous = time()                                     # Startzeit festhalten
33      running = True
34      while running:
35          if pygame.time.get_ticks() > start_time + Settings.LIMIT:
36              defender_speed = 0
37          # Events
38          for event in pygame.event.get():
39              if event.type == pygame.QUIT:
40                  running = False
41
42          # Update
43          defender_rect.top += defender_direction_v * defender_speed * Settings.DELTATIME
44          if defender_rect.bottom >= Settings.WINDOW.height:
45              defender_direction_v *= -1
46          elif defender_rect.top <= 0:
47              defender_direction_v *= -1
48
49      # Draw

```

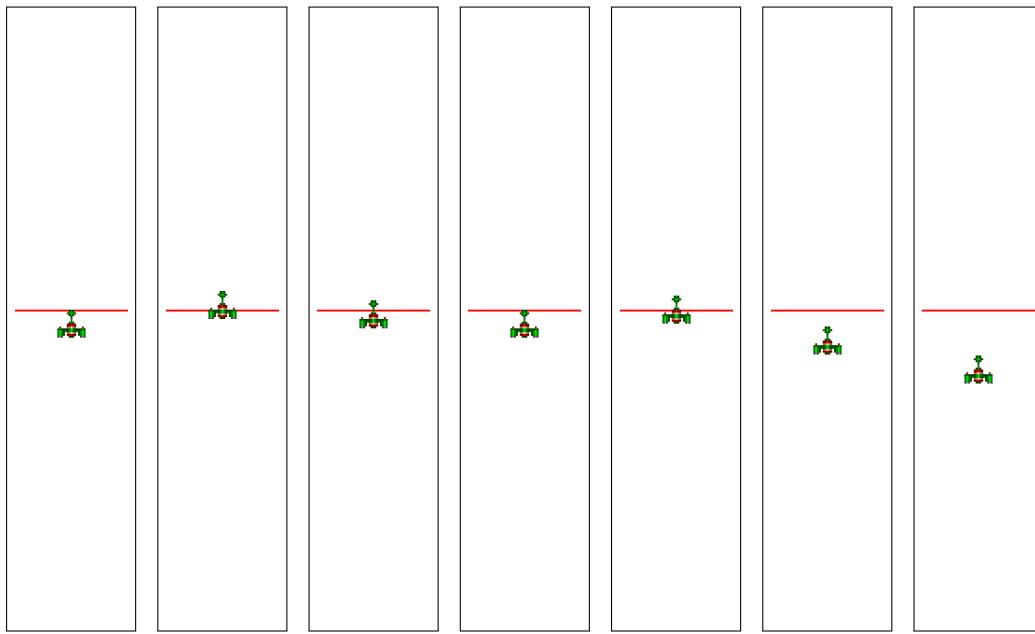


Abbildung 2.22: Normalisierte Bewegung mit $1/fps$ bei gleicher Geschwindigkeit aber unterschiedlichen Frameraten (von links nach rechts: 10, 30, 60, 120, 240, 300, 600)

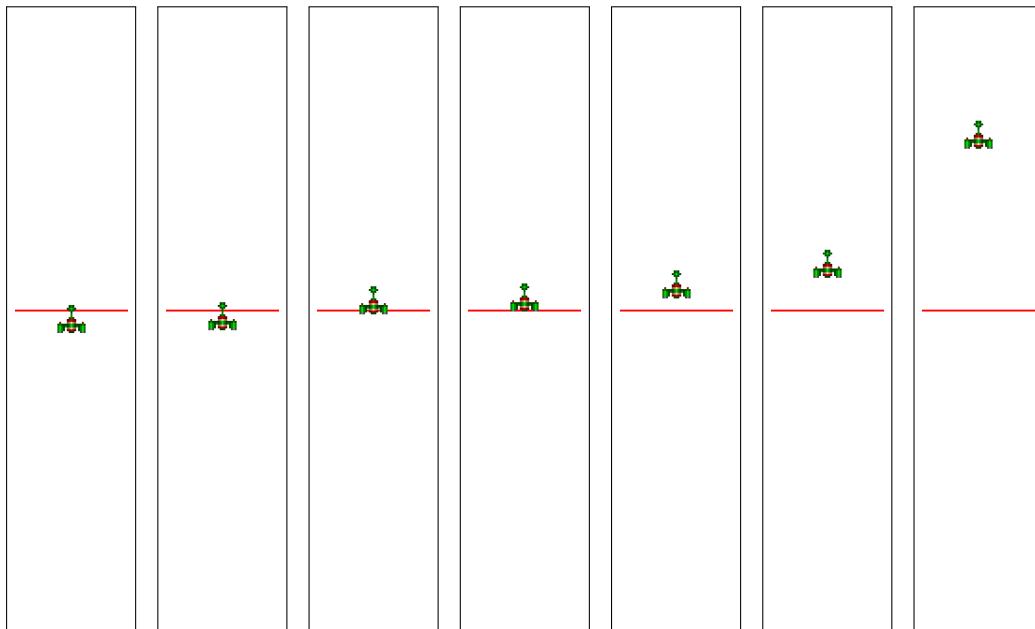


Abbildung 2.23: Normalisierte Bewegung mit `pygame.clock.tick()` bei gleicher Geschwindigkeit aber unterschiedlichen Frameraten (von links nach rechts: 10, 30, 60, 120, 240, 300, 600)

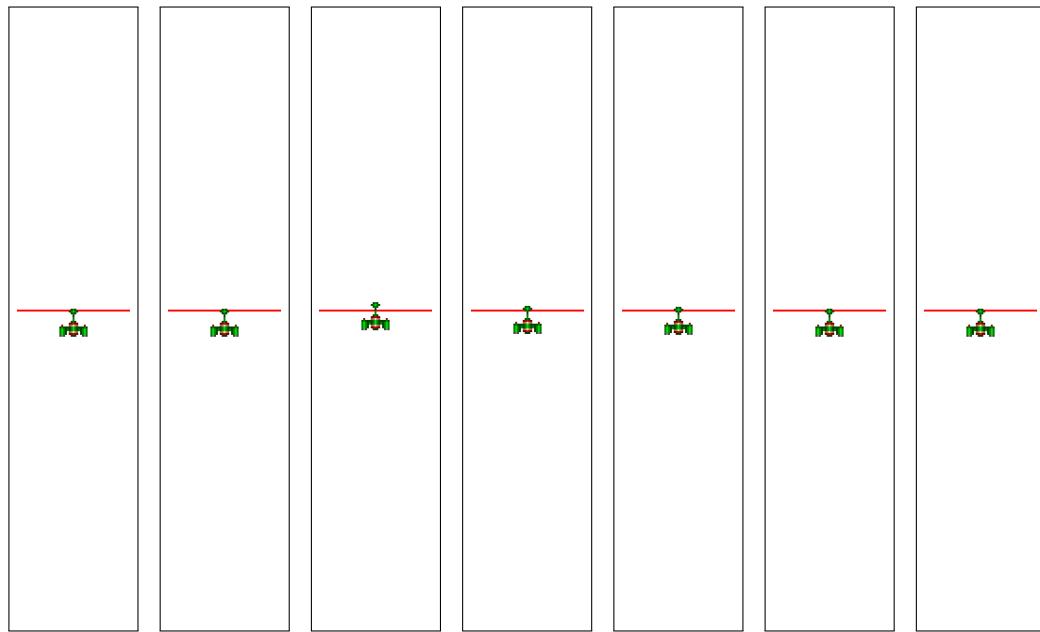


Abbildung 2.24: Normalisierte Bewegung mit `pygame.clock.tick()` (float) bei gleicher Geschwindigkeit aber unterschiedlichen Frameraten (von links nach rechts: 10, 30, 60, 120, 240, 300, 600)

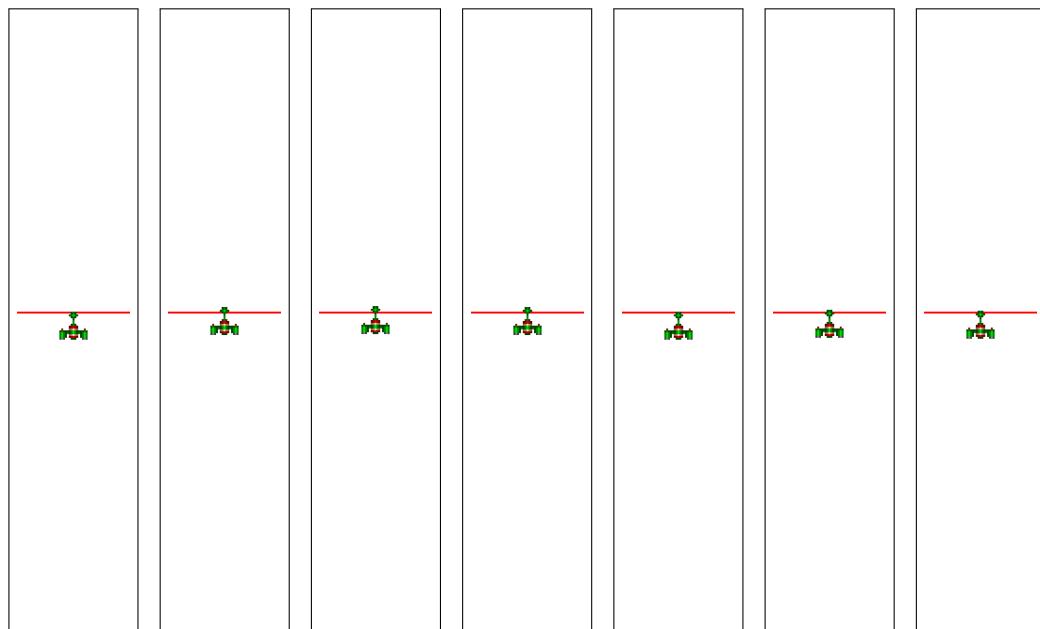


Abbildung 2.25: Normalisierte Bewegung mit `time.time()` bei gleicher Geschwindigkeit aber unterschiedlichen Frameraten (von links nach rechts: 10, 30, 60, 120, 240, 300, 600), Version 3

```
50     screen.fill("white")
51     pygame.draw.line(screen, "red", (0, 315), (Settings.WINDOW.width, 315), 2)
52     screen.blit(defender_image, defender_rect)
53     pygame.display.flip()
54     clock.tick(Settings.FPS)
55     time_current = time()                      # Aktuelle Zeit festhalten
56     Settings.DELTATIME = time_current - time_previous  # Zeitverbrauch
57     time_previous = time_current                # Neue Startzeit
58     pygame.quit()
```

Was war neu?

Die Position eines Objektes wird in einem `Rect`- oder `FRect`-Objekt abgelegt. In jedem Frame wird die Position überprüft und ggf. verändert. Bei einer Bildschirmausgabe entsteht dadurch der Eindruck einer Bewegung. Das Ergebnis einer Bewegung wird in der Regel zunächst in einer Variablen zwischengespeichert und überprüft, bevor es zur Positionsänderung verwendet wird.

Die Bewegungsrichtung wird durch das Vorzeichen und die Geschwindigkeit durch den Wert der Geschwindigkeitsvariablen kodiert. Dabei werden die horizontale und vertikale Richtung getrennt verarbeitet.

Um unabhängig von der tatsächlichen Framerate zu werden, muss bei der Berechnung der neuen Position ein Korrekturwert (Deltatime) verwendet werden. Dieser kann selbst berechnet werden oder aus dem Aufruf von `pygame.time.Clock.tick()` verwendet werden.

Es wurden folgende Pygame-Elemente eingeführt:

- `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>

2.5 Sprite-Klasse

Im letzten Beispiel fiel auf, dass viele Variablen mit `defender_` beginnen. Mit anderen Worten, es sind Attribute einer Sache und schreien förmlich nach einer Formulierung als Klasse.

Diese Klasse soll alle Informationen bzgl. der Aktualisierung und Darstellung des Bitmaps enthalten. Einige Elemente wie `defender_image` und `defender_rect` scheinen aber doch bei jeder Bitmap-Verarbeitung eine Rolle zu spielen. Auch wird es bei jedem Bitmap einen Bedarf für Zustandsänderungen und für die Bildschirmausgabe geben. Tatsächlich gibt es in Pygame schon eine Klasse, die mir genau dazu ein [Framework](#) bietet: `pygame.sprite.Sprite`.

Sprite

Formulieren wir also die Klasse `Defender` als eine Kindklasse von `Sprite` (Zeile 13).

Quelltext 2.25: Sprites (1), Version 1.0

```

13  class Defender(pygame.sprite.Sprite):           # Kindklasse von Sprite
14
15      def __init__(self) -> None:                 # Konstruktor
16          super().__init__()
17          self.image = pygame.image.load("images/defender01.png").convert_alpha()
18          self.image = pygame.transform.scale(self.image, (30, 30))
19          self.rect = pygame.Rect(self.image.get_rect())
20          self.rect.centerx = Settings.WINDOW.centerx
21          self.rect.bottom = Settings.WINDOW.bottom - 5
22          self.speed = 300
23
24      def update(self) -> None:                   # Zustandsberechnung
25          newpos = self.rect.move(self.speed * Settings.DELTATIME, 0)
26          if newpos.right >= Settings.WINDOW.right:
27              self.change_direction()
28          elif newpos.left <= Settings.WINDOW.left:
29              self.change_direction()
30          else:
31              self.rect = newpos
32
33      def draw(self, screen: pygame.surface.Surface) -> None: # Malen
34          screen.blit(self.image, self.rect)
35
36      def change_direction(self) -> None:           # OO style
37          self.speed *= -1

```

Die Zeilen des Konstruktors (Zeile 15ff.) entsprechen denen der vorherigen Version. Lediglich der Präfix `defender_` wird durch `self.` ersetzt, wodurch die Variablen zu Attributen der Klasse werden. Sie sollten keine Schwierigkeiten haben, diese zu verstehen.

`self.rect`
`self.image`

Jede Kindklasse von `Sprite` muss zwei Attribute haben: `rect` und `image`. Auf diese beiden Attribute greifen nämlich die schon vorformulierten Lösungen zur Kollisionserkennung, Bildschirmausgabe etc. zu. Wir werden später noch den Nutzen sehen.

In Zeile 24ff. werden die Kollisionserkennungen und die Zustandsänderungen formuliert. Hier fällt besonders die Berechnung der neuen Position mit `move()` auf.

Neu ist der Aufruf der Methode `change_direction()`. Diese Methode (Zeile 36) ist mehr *OO-like* also die vorherige Version. In der objektorientierten Programmierung werden

Algorithmen nicht direkt programmiert, sondern man sendet an das Objekt Nachrichten, und diese werden dann intern – und von außen nicht sichtbar wie – umgesetzt. Hier bedeutet dies, dass ich an der entsprechenden Stelle nicht den Richtungswechsel direkt durchführe, sondern mir selbst die Nachricht zusende, dass die Richtung geändert werden muss.

Mit der Methode `draw()` in Zeile 33 wird die Bildschirmausgabe gekapselt.

Quelltext 2.26: Sprites (2), Version 1.0

```

40 def main():
41     os.environ['SDL_VIDEO_WINDOW_POS'] = "10, 50"
42     pygame.init()
43
44     screen = pygame.display.set_mode(Settings.WINDOW.size)
45     pygame.display.set_caption("Sprite")
46     clock = pygame.time.Clock()
47     defender = Defender()                      # Objekt anlegen
48
49     time_previous = time()
50     running = True
51     while running:
52         # Events
53         for event in pygame.event.get():
54             if event.type == pygame.QUIT:
55                 running = False
56
57         # Update
58         defender.update()                      # Aufruf
59
60         # Draw
61         screen.fill("white")
62         defender.draw(screen)                 # Aufruf
63         pygame.display.flip()
64
65         clock.tick(Settings.FPS)
66         time_current = time()
67         Settings.DELTATIME = time_current - time_previous
68         time_previous = time_current
69     pygame.quit()

```

Die Verwendung der Klasse `Defender` ist nun denkbar einfach geworden. In der Zeile 47 wird ein Objekt der Klasse erzeugt. In Zeile 58 wird `update()` aufgerufen und in Zeile 62 `draw()`.

Ein Vorteil der neuen Architektur ist die bessere Übersichtlichkeit und Verständlichkeit des Hauptprogrammes. Durch Namenskonvention (sprechende Klassen- und Funktionsnamen) wird der grundsätzliche Ablauf klarer und nicht mehr von Details überlagert.

Ich möchte nun die Möglichkeiten der `Sprite`-Klasse nutzen, um die Kollisionsprüfung mit dem Rand nicht mehr selbst durchzuführen.

Los geht's: Da wir die Kollisionsprüfung anders organisieren wollen, wird erstmal das `update()` wieder einfach. Wir berechnen lediglich die neue Position.

Dabei wird in Zeile 25 die Methode `pygame.Rect.move_ip()` eingeführt. Sie arbeitet wie `move()`, nur dass hier die Änderung direkt im Rechteck durchgeführt wird; `ip` steht hier für *in place*. Bei `move()` bleibt das ursprüngliche Rechteck unverändert.

`move_ip()`

Quelltext 2.27: Sprites (1), Version 1.1

```
24  def update(self) -> None:
25      self.rect.move_ip(self.speed * Settings.DELTATIME, 0)  #
```

Damit die Ränder sichtbar werden und ich die Kollision besser erkennen kann, werden die Ränder nun zu zwei Steinwänden rechts und links; auch diese Bitmaps werden als Kindklasse von `pygame.sprite.Sprite` implementiert. Da der Zustand der beiden Wände sich nie verändert, kann ich auf die Programmierung von `update()` verzichten.

Quelltext 2.28: Sprites (2), Version 1.1

```
34  class Border(pygame.sprite.Sprite):
35
36      def __init__(self, leftright: str) -> None:
37          super().__init__()
38          self.image = pygame.image.load("images/brick01.png").convert_alpha()
39          self.image = pygame.transform.scale(self.image, (35, Settings.WINDOW.width))
40          self.rect = self.image.get_rect()
41          if leftright == 'right':
42              self.rect.left = Settings.WINDOW.width - self.rect.width
43
44      def draw(self, screen: pygame.surface.Surface) -> None:
45          screen.blit(self.image, self.rect)
```

Nun erzeuge ich die beiden Ränder:

Quelltext 2.29: Sprites (3), Version 1.1

```
56  border_left = Border('left')
57  border_right = Border('right')
```

Bisher war alles easy.

Quelltext 2.30: Sprites (4), Version 1.1

```
68      if pygame.sprite.collide_rect(defender, border_left):
69          defender.change_direction()
70      elif pygame.sprite.collide_rect(defender, border_right):
71          defender.change_direction()
72      defender.update()
```

collide_rect()

Was passiert hier? Mit der Methode `pygame.sprite.collide_rect()` werden die Rechtecke zweier `Sprite`-Objekte auf Kollision untersucht. Eine eigene Abfrage der linken und rechten Grenzen bleibt mir damit erspart.

Für beide Ränder – allgemeiner gesprochen für viele `Sprite`-Objekte – wird hier die Kollision mit einem einzelnen Objekt überprüft. Grundsätzlich kommen Sprites selten einzeln daher, sondern oft in Gruppen. Auch dies ist schon in Pygame vorgesehen und führt zu weiteren Vereinfachungen.

Quelltext 2.31: Sprites (1), Version 1.2

```

42 def main():
43     os.environ['SDL_VIDEO_WINDOW_POS'] = "10, 50"
44     pygame.init()
45
46     screen = pygame.display.set_mode(Settings.WINDOW.size)
47     pygame.display.set_caption("Sprite")
48     clock = pygame.time.Clock()
49     defender = pygame.sprite.GroupSingle(Defender())
50     all_border = pygame.sprite.Group()
51     all_border.add(Border('left'))
52     all_border.add(Border('right'))
53
54     time_previous = time()
55     running = True
56     while running:
57         # Events
58         for event in pygame.event.get():
59             if event.type == pygame.QUIT:
60                 running = False
61
62         # Update
63         if pygame.sprite.spritecollide(defender.sprite, all_border, False): # !
64             defender.sprite.change_direction() #
65             defender.update()
66
67         # Draw
68         screen.fill((255, 255, 255))
69         defender.draw(screen)
70         all_border.draw(screen)           # Mit einem Rutsch
71         pygame.display.flip()
72
73         clock.tick(Settings.FPS)
74         time_current = time()
75         Settings.DELTATIME = time_current - time_previous
76         time_previous = time_current
77     pygame.quit()

```

Der Verteidiger wird nicht mehr direkt angesprochen, sondern in eine Luxuskiste ge-
packt. Ich komme später nochmal darauf zurück. Die beiden Border-Objekte werden
nicht mehr in zwei Objektvariablen abgelegt, sondern ebenfalls in eine Luxuskiste, der
pygame.sprite.Group. Hier könnte ich nun noch andere Grenzen oder Grenzwälle ab-
legen. Von der Spiellogik her würden diese nun immer mit einem Schlag gemeinsam
verarbeitet. Deutlich wird das bei diesem Minispiel an zwei Stellen.

Group

Die erste Stelle ist Zeile 63 und dort wird eine andere Version der Kollisionsprüfung ver-
wendet: pygame.sprite.spritecollide(). Der erste Parameter ist *ein* Sprite-Objekt.
In unserem Fall ist es der Verteidiger. Der zweite Parameter ist eine Spritegruppe mit
allen Border-Objekten. Also wird der Verteidiger mit allen Mitgliedern der Gruppe auf
Kollisionen überprüft. Dies funktioniert nur, wenn alle Sprites ein Rect-Objekt mit dem
Namen rect als Attribut haben. Der dritte Parameter – hier False – steuert, ob das
kollidierende Sprite aus der Liste entfernt werden soll. Dieses Feature ist in Spielen recht
interessant, will man doch beispielsweise Felsen, die von einem Raumschiff zerschossen
wurden, löschen.

spritecollide()

Die zweite Stelle ist Zeile 70. Hier wird nicht mehr für jedes Objekt einzeln draw() aufge-
rufen, sondern für die ganze Gruppe. Nutzt man diesen Service, kann man die Methode
draw() aus seiner eigenen Klasse (hier Border und Defender) entfernen, wodurch schon
wieder alles einfacher wird.

GroupSingle

Es scheint also eine gute Idee zu sein, die Sprites in solche Luxuskisten zu packen. Aber was war nochmal mit dem Defender? Um die Vorteile einer Spritegruppe nutzen zu können, kann man auch Gruppen anlegen, die nur ein Element enthalten. Damit diese Gruppen aber etwas effizienter arbeiten können – schließlich weiß man ja, dass nur ein Element in der Gruppe ist –, gibt es dafür den Spezialfall `pygame.sprite.GroupSingle`. Da man oft den Bedarf hat auf das einzige `Sprite`-Objekt der *Gruppe* zuzugreifen, hat diese Gruppe das zusätzliche Attribut `sprite` (siehe Zeile 63f.).

Am Ende möchte ich meinen OO-Ansatz noch weiterverfolgen und auch das Hauptprogramm in eine `Game`-Klasse umwandeln. Wichtig ist mir dabei, gleich von Beginn an eine Strukturdisziplin zu etablieren. Je länger Sie in der Softwareentwicklung tätig bleiben, desto mehr freunden Sie sich mit Begriffen wie *Ordnung* oder *Struktur* an. Sie helfen auch bei komplexeren Spielen, nicht den roten Faden zu verlieren. Besonders hilfreich ist dabei das **Single Responsibility Principle (SRP)**.

Quelltext 2.32: Game-Klasse

```

42 class Game(object):
43
44     def __init__(self) -> None:
45         os.environ['SDL_VIDEO_WINDOW_POS'] = "10,50"
46         pygame.init()
47         self.screen = pygame.display.set_mode(Settings.WINDOW.size)
48         pygame.display.set_caption("Sprite")
49         self.clock = pygame.time.Clock()
50         self.defender = pygame.sprite.GroupSingle(Defender())
51         self.all_border = pygame.sprite.Group()
52         self.all_border.add(Border('left'))
53         self.all_border.add(Border('right'))
54         self.running = False
55
56     def run(self) -> None:
57         time_previous = time()
58         self.running = True
59         while self.running:
60             self.watch_for_events()
61             self.update()
62             self.draw()
63             self.clock.tick(Settings.FPS)
64             time_current = time()
65             Settings.DELTATIME = time_current - time_previous
66             time_previous = time_current
67         pygame.quit()
68
69     def watch_for_events(self) -> None:
70         for event in pygame.event.get():
71             if event.type == pygame.QUIT:
72                 self.running = False
73
74     def update(self) -> None:
75         if pygame.sprite.spritecollide(self.defender.sprite, self.all_border, False):
76             self.defender.sprite.change_direction() # Gefällt mir nicht!
77         self.defender.update()
78
79     def draw(self) -> None:
80         self.screen.fill((255, 255, 255))
81         self.defender.draw(self.screen)
82         self.all_border.draw(self.screen)
83         pygame.display.flip()

```

Ein Beispiel für den letzten Punkt ist die Einrichtung der Klasse `Game`. Hier wird der Quelltext nicht einfach ins `__main__` gestellt, sondern gekapselt und geordnet und damit flexibel verfügbar gemacht. Ein Beispiel für das SRP sind die Methoden `watch_for_events()`, `update()` und `draw()`. Es ist eben nicht die Aufgabe von `run()` alles zu organisieren. Aus Sicht der Hauptprogrammschleife interessiert es mich nicht, welche Events abgefragt und wie sie verarbeitet werden. Ich will nur, dass die Events pro Frame einmal betrachtet werden. Auch will sich `run()` nicht um die Reihenfolge kümmern, wie die Sprites auf den Bildschirm gezeichnet werden. Das soll die Methode `draw()` erledigen. Die Methode `run()` stellt nur sicher, dass zuerst die Sprites ihre neuen Zustände berechnen und dann die Ausgabe erfolgt.

Verbleibt noch ein Aspekt, den ich hier umsetzen möchte: Der Aufruf von `change_direction()` in Zeile 76 gefällt mir nicht. Er ist eine Verletzung von OO-Regeln.

Die Spritegruppe ist eine Liste von `Sprite`-Objekten. Die Klasse `pygame.sprite.Sprite`

kennt aber keine Methode `change_direction()`. Deshalb ist das nicht ganz sauber, die hier aufzurufen. Python hat mit soetwas kein Problem, aber das sollte nicht der Maßstab sein.

Es bietet sich vielmehr an, die Methode `update()` anzupassen. Schaut man sich die **Signatur** der Methode `pygame.sprite.Sprite.update()` genauer an, so sehen Sie, dass hier eigentlich frei definierbare Übergabeparameter vorgesehen sind. Ich habe mir ange-
wöhnt, einen Parameter mit Namen `action` zu benutzen, um Methoden der Kindklasse aufzurufen. So wird `change_direction()` nach Zeile 29 durch `update()` aufgerufen und nicht mehr von außen.

Quelltext 2.33: `Defender.update()`

```
25  def update(self, *args: Any, **kwargs: Any) -> None:
26      if "action" in kwargs.keys():
27          if kwargs["action"] == "newpos":          # Neue Position berechnen
28              self.rect.move_ip(self.speed * Settings.DELTATIME, 0)
29          elif kwargs["action"] == "direction":    # Richtung wechseln
30              self.change_direction()
```

Der Aufruf erfolgt dann in Quelltext 2.34 in Zeile 81 indirekt durch die Verwendung des Übergabeparameters.

Quelltext 2.34: `Game.update()`

```
79  def update(self) -> None:
80      if pygame.sprite.spritecollide(self.defender.sprite, self.all_border, False):
81          self.defender.update(action="direction")  # Besser
82          self.defender.update(action="newpos")
```

Was war neu?

Von der Verhaltenslogik her: *gar nichts*. Die vorhandene Anwendung wurde nur in ein flexibles Framework eingebettet.

Es wurden folgende Pygame-Elemente eingeführt:

- `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.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>

2.6 Tastatur

Ich möchte hier die Tastatur nicht erschöpfend behandeln, sondern lediglich das Grundprinzip verdeutlichen. So soll die Bewegungsrichtung durch die Pfeiltasten gesteuert werden können. Ebenso soll das Raumschiffe stehen bleiben oder sich wieder in Bewegung setzen können. Auch kann das Spiel jetzt durch die Escape-Taste verlassen werden ([Boss-Taste](#)).

Im ersten Schritt wird ein Dictionary der möglichen Richtungen in `Settings` angelegt. Diese werden als `Vector2`-Objekte verwaltet, da diese einfacher für mathematische Operationen verwendet werden können.

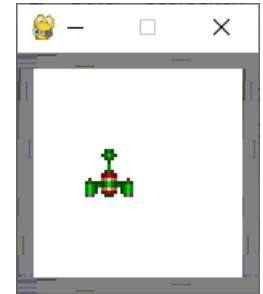


Abb. 2.26: Ränder

Quelltext 2.35: Bewegung durch Tastatur steuern (1), `Settings`

```
8 class Settings:
9     WINDOW = pygame.rect.Rect((0, 0), (600, 100))
10    FPS = 60
11    DELTATIME = 1.0 / FPS
12    DIRECTIONS = {"right": pygame.math.Vector2(1, 0),
13                  "left": pygame.math.Vector2(-1, 0),
14                  "up": pygame.math.Vector2(0, -1),
15                  "down": pygame.math.Vector2(0, 1)}
```

Danach bereiten wir die Verteidiger-Klasse vor bzw. wandeln sie ein wenig ab (Quelltext [2.36](#)). Das Sprite wird nun nicht mehr unten sondern mittig platziert (Zeile [25](#)), und das Raumschiff soll sich jetzt auch vertikal bewegen können. Dazu braucht es entweder zwei entsprechende Variablen oder aber ein `Vector2`-Objekt. Ich nehme ein `Vector2`-Objekt (Zeile [26](#)), wobei das erste Element der Richtungsvektor der horizontalen und das zweite der vertikalen Richtung ist. Der jeweilige Richtungsvektor wird dabei entsprechend der schon oben vorgestellten Semantik gesetzt. In der Methode `change_direction()` wird entweder der Richtungsvektor gesetzt oder die Geschwindigkeit angepasst. Bewegen und Stehenbleiben wird einfach dadurch erreicht, dass ich die Geschwindigkeit bei `start` auf 100 bzw. bei `stop` auf 0 setze.

Quelltext 2.36: Bewegung durch Tastatur steuern (1), `Defender`

```
18 class Defender(pygame.sprite.Sprite):
19
20     def __init__(self) -> None:
21         super().__init__()
22         self.image = pygame.image.load("images/defender01.png").convert_alpha()
23         self.image = pygame.transform.scale(self.image, (30, 30))
24         self.rect = pygame.rect.FRect(self.image.get_rect())
25         self.rect.center = Settings.WINDOW.center # 2 Dimensionen
26         self.direction = Settings.DIRECTIONS["right"] # 2 Dimensionen
27         self.change_direction("right")
28         self.change_direction("start")
29
30     def update(self, *args: Any, **kwargs: Any) -> None:
31         if "action" in kwargs.keys():
32             if kwargs["action"] == "move":
```

```

33         self.rect.move_ip(self.speed * Settings.DELTATIME * self.direction)
34     elif kwargs["action"] == "switch":
35         self.direction *= -1
36     elif "direction" in kwargs.keys():
37         self.change_direction(kwargs["direction"])
38
39     def change_direction(self, direction: str) -> None:
40         if direction in Settings.DIRECTIONS.keys():
41             self.direction = Settings.DIRECTIONS[direction]
42         elif direction == "stop":
43             self.speed = 0
44         elif direction == "start":
45             self.speed = 100

```

Die Klasse `Border` wird trivialerweise so erweitert, dass alle vier Seiten der Spielfläche nun durch eine Steinwand begrenzt werden (Quelltext 2.37). Dazu wird im Konstruktor abgefragt, auf welcher Seite die Wand hochgezogen werden soll. Rechts und links wird das Bitmap in die Höhe gestreckt und oben und unten in die Breite. Anschließend wird das `Rect`-Objekt ermittelt und die Position festgelegt.

Quelltext 2.37: Bewegung durch Tastatur steuern (2), `Border`

```

48 class Border(pygame.sprite.Sprite):
49
50     def __init__(self, whichone: str) -> None:
51         super().__init__()
52         self.image = pygame.image.load("images/brick1.png").convert_alpha()
53         if whichone == 'right':
54             self.image = pygame.transform.scale(self.image, (10, Settings.WINDOW.height))
55             self.rect = self.image.get_rect()
56             self.rect.right = Settings.WINDOW.right
57         elif whichone == 'left':
58             self.image = pygame.transform.scale(self.image, (10, Settings.WINDOW.height))
59             self.rect = self.image.get_rect()
60             self.rect.left = Settings.WINDOW.left
61         elif whichone == 'top':
62             self.image = pygame.transform.scale(self.image, (Settings.WINDOW.width, 10))
63             self.rect = self.image.get_rect()
64             self.rect.top = Settings.WINDOW.top
65         elif whichone == 'down':
66             self.image = pygame.transform.scale(self.image, (Settings.WINDOW.width, 10))
67             self.rect = self.image.get_rect()
68             self.rect.bottom = Settings.WINDOW.bottom

```

Es werden dann die vier Objekte der `Border`-Klasse erzeugt und der Spritegruppe hinzugefügt.

Quelltext 2.38: Bewegung durch Tastatur steuern (3), Game-Konstruktor

```

71 class Game(object):
72
73     def __init__(self) -> None:
74         os.environ['SDL_VIDEO_WINDOW_POS'] = "10,50"
75         pygame.init()
76         self.screen = pygame.display.set_mode(Settings.WINDOW.size)
77         pygame.display.set_caption("Sprite")
78         self.clock = pygame.time.Clock()
79         self.defender = pygame.sprite.GroupSingle(Defender())
80         self.all_border = pygame.sprite.Group()

```

```

81         self.all_border.add(Border('left'))
82         self.all_border.add(Border('right'))
83         self.all_border.add(Border('top'))
84         self.all_border.add(Border('down'))
85         self.running = False

```

Kommen wir jetzt zur eigentlichen Tastaturverarbeitung: Das Verwenden einer Taste kann die Ereignistypen `pygame.KEYDOWN` oder `pygame.KEYUP` auslösen. In unserem Beispiel (Zeile 104) wollen wir wissen, welche Taste *gedrückt* wurde, also verwenden wir `KEYDOWN`. Anschließend können wir über `pygame.event.key` ermitteln, welche Taste gedrückt wurde. Dazu stellt uns Pygame in `pygame.key` eine Liste von vordefinierten Konstanten zur Verfügung (siehe Tabelle 2.5 auf der nächsten Seite und Tabelle 2.6 auf Seite 60).

Quelltext 2.39: Bewegung durch Tastatur steuern (4), `Game.watch_for_events()`

```

100     def watch_for_events(self) -> None:
101         for event in pygame.event.get():
102             if event.type == pygame.QUIT:
103                 self.running = False
104             elif event.type == pygame.KEYDOWN:           # Taste drücken
105                 if event.key == pygame.K_ESCAPE:          # Boss-Taste
106                     self.running = False
107                 elif event.key == pygame.K_RIGHT:         # Pfeiltasten
108                     self.defender.update(direction="right")
109                 elif event.key == pygame.K_LEFT:
110                     self.defender.update(direction="left")
111                 elif event.key == pygame.K_UP:
112                     self.defender.update(direction="up")
113                 elif event.key == pygame.K_DOWN:
114                     self.defender.update(direction="down")
115                 elif event.key == pygame.K_SPACE:          # Leerzeichen-Taste
116                     self.defender.update(direction="stop")
117                 elif event.key == pygame.K_r:
118                     if event.mod & pygame.KMOD_LSHIFT:  # Shift-Taste
119                         self.defender.update(direction="stop")
120                     else:
121                         self.defender.update(direction="start")

```

`K_ESCAPE` Fangen wir mit der Boss-Taste an. In Zeile 105 wird über die Konstante `K_ESCAPE` abgefragt, ob die gedrückte Taste die Escape-Taste ist. Wie beim *Weg-Xen* wird danach einfach das Flag der Hauptprogrammschleife auf `False` gesetzt. Probieren Sie es aus!

`K_LEFT` `K_RIGHT` `K_UP` `K_DOWN` Danach werden mit Hilfe von `K_LEFT`, `K_RIGHT`, `K_UP` und `K_DOWN` ab Zeile 107ff. die vier Pfeiltasten abgefragt und die entsprechende Nachricht an den Verteidiger gesendet.

`K_SPACE` Mit Hilfe der Leerzeichen-Taste `K_SPACE` wird das Raumschiff in Zeile 115 gestoppt.

Um den Einsatz der Shift-Taste (Umschalttaste) mal zu demonstrieren, habe ich hier das `r` doppelt belegt (Zeile 118). Das große `R` stoppt das Raumschiff und das kleine `r` startet es wieder. Dabei wird die Variable `event.mod` mit Hilfe einer bitweisen Und-Verknüpfung dahingehend überprüft, ob das entsprechende Bit `KMOD_LSHIFT` für die linke Shift-Taste gedrückt wurde.

Dies soll ersteinmal ausreichen. Die Tastatur ist nur eine Möglichkeit der Spielsteuerung. Maus, Game-Controller oder Joystick sind ebenfalls in Pygame möglich.

Was war neu?

Die Tastatur sendet Ereignisnachrichten, die man abfangen und auswerten kann. Dabei wird zum einen unterschieden, was mit der Tastatur gemacht wurde (`event.type`) und dann mit welcher Taste (`event.key`). Über `event.mod` kann bitweise abgefragt werden, welche Steuertasten auf der Tastatur verwendet wurden.

Es wurden folgende Pygame-Elemente eingeführt:

- `pygame.key`:
<https://pyga.me/docs/ref/key.html>
- `pygame.KEYDOWN`, `pygame.KEYUP`:
<https://pyga.me/docs/ref/event.html>

Tabelle 2.5: Liste von vordefinierten Tastaturkonstanten

Konstante	Bedeutung	Beschreibung
<code>K_BACKSPACE</code>	<code>\b</code>	Löschen (backspace)
<code>K_TAB</code>	<code>\t</code>	Tabulator
<code>K_CLEAR</code>		Leeren
<code>K_RETURN</code>	<code>\r</code>	Eingabe (return, enter)
<code>K_PAUSE</code>		Pause
<code>K_ESCAPE</code>	<code>^[</code>	Abbruch (escape)
<code>K_SPACE</code>	<code>□</code>	Leerzeichen (space)
<code>K_EXCLAIM</code>	<code>!</code>	Ausrufezeichen
<code>K_QUOTEDBL</code>	<code>"</code>	Gänsefüßchen
<code>K_HASH</code>	<code>#</code>	Doppelkreuz (hash)
<code>K_DOLLAR</code>	<code>\$</code>	Dollar
<code>K_AMPERSAND</code>	<code>&</code>	Kaufmannsund
<code>K_QUOTE</code>	<code>'</code>	Hochkomma
<code>K_LEFTPAREN</code>	<code>(</code>	Linke runde Klammer
<code>K_RIGHTPAREN</code>	<code>)</code>	Rechte runde Klammer
<code>K_ASTERISK</code>	<code>*</code>	Sternchen
<code>K_PLUS</code>	<code>+</code>	Plus
<code>K_COMMMA</code>	<code>,</code>	Komma
<code>K_MINUS</code>	<code>-</code>	Minus
<code>K_PERIOD</code>	<code>.</code>	Punkt
<code>K_SLASH</code>	<code>/</code>	Schrägstrich
<code>K_0</code>	<code>0</code>	0
<code>K_1</code>	<code>1</code>	1
<code>K_2</code>	<code>2</code>	2
<code>K_3</code>	<code>3</code>	3
<code>K_4</code>	<code>4</code>	4
<code>K_5</code>	<code>5</code>	5

Tabelle 2.5: Liste von vordefinierten Tastaturkonstanten (Fortsetzung)

Konstante	Bedeutung	Beschreibung
K_6	6	6
K_7	7	7
K_8	8	8
K_9	9	9
K_COLON	:	Doppelpunkt
K_SEMICOLON	;	Semicolon
K_LESS	<	Kleiner
K_EQUALS	=	Gleich
K_GREATER	>	Größer
K_QUESTION	?	Fragezeichen
K_AT	@	Klammeraffe
K_LEFTBRACKET	[Linke eckige Klammer
K_BACKSLASH	\	Umgekehrter Schrägstrich
K_RIGHTBRACKET]	Rechte eckige Klammer
K_CARET	^	Hütchen
K_UNDERSCORE	_	Unterstrich
K_BACKQUOTE	`	Akzent Grvis
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

Tabelle 2.5: Liste von vordefinierten Tastaturkonstanten (Fortsetzung)

Konstante	Bedeutung	Beschreibung
K_x	x	x
K_y	y	y
K_z	z	z
K_DELETE	□	Löschen (delete)
K_KP0	□	Nummernfeld 0
K_KP1	□	Nummernfeld 1
K_KP2	□	Nummernfeld 2
K_KP3	□	Nummernfeld 3
K_KP4	□	Nummernfeld 4
K_KP5	□	Nummernfeld 5
K_KP6	□	Nummernfeld 6
K_KP7	□	Nummernfeld 7
K_KP8	□	Nummernfeld 8
K_KP9	□	Nummernfeld 9
K_KP_PERIOD	.	Nummernfeld Punkt
K_KP_DIVIDE	/	Nummernfeld Geteilt/Schrägstrich
K_KP_MULTIPLY	*	Nummernfeld Mal/Sternchen
K_KP_MINUS	-	Nummernfeld Minus
K_KP_PLUS	+	Nummernfeld Plus
K_KP_ENTER	\r	Nummernfeld Eingabe (return, enter)
K_KP_EQUALS	=	Nummernfeld Gleich
K_UP	□	Pfeil nach oben
K_DOWN	□	Pfeil nach unten
K_RIGHT	□	Pfeil nach rechts
K_LEFT	□	Pfeil nach links
K_INSERT	□	Einfügen ein/aus
K_HOME	□	Pos1
K_END	□	Ende
K_PAGEUP	□	Hochblättern
K_PAGEDOWN	□	Runterblättern
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

Tabelle 2.5: Liste von vordefinierten Tastaturkonstanten (Fortsetzung)

Konstante	Bedeutung	Beschreibung
K_F11	□	F11
K_F12	□	F12
K_F13	□	F13
K_F14	□	F14
K_F15	□	F15
K_NUMLOCK	□	Umschalten Zahlen
K_CAPSLOCK	□	Umschalten Großbuchstaben
K_SCROLLLOCK	□	Umschalten auf scrollen
K_RSHIFT	□	Rechte Umschalttaste
K_LSHIFT	□	Linke Umschalttaste
K_RCTRL	□	Rechte Steuerungstaste
K_LCTRL	□	Linke Steuerungstaste
K_RALT	□	Rechte Alternativtaste
K_LALT	□	Linke Alternativtaste
K_RMETA	□	Rechte Metataste
K_LMETA	□	Linke Metataste
K_LSUPER	□	Linke Windowstaste
K_RSUPER	□	Rechte Windowstaste
K_MODE	□	AltGr Umschalter
K_HELP	□	Hilfe
K_PRINT	□	Bildschirmschuss/Screenshot
K_SYSREQ	□	Systemabfrage
K_BREAK	□	Abbruch/Unterbrechung
K_MENU	□	Menü
K_POWER	□	Ein-/Ausschalten
K_EURO	€	Euro-Währungszeichen
K_AC_BACK	□	Android Zurückschalter

Tabelle 2.6: Liste von vordefinierten Konstanten zur Tastaturschaltung

Konstante	Beschreibung
KMOD_NONE	Keine Belegungstaste gedrückt
KMOD_LSHIFT	Linke Umschalttaste
KMOD_RSHIFT	Rechte Umschalttaste
KMOD_SHIFT	Linke oder rechte Umschalttaste oder beide
KMOD_LCTRL	Linke Steuerungstaste
KMOD_RCTRL	Rechte Steuerungstaste
KMOD_CTRL	Linke oder rechte Steuerungstaste oder beide
KMOD_LALT	Linke Alternativtaste

Tabelle 2.6: Liste von vordefinierten Konstanten zur Tastaturschaltung (Fortsetzung)

Konstante	Beschreibung
KMOD_RALT	Rechte Alternativtaste
KMOD_ALT	Linke oder rechte Alternativtaste oder beide
KMOD_LMETA	Linke Metataste
KMOD_RMETA	Rechte Metataste
KMOD_META	Linke oder rechte Metataste oder beide
KMOD_CAPS	Umschalten Großbuchstaben
KMOD_NUM	Umschalten Zahlen
KMOD_MODE	AltGr Umschalter

2.7 Textausgabe mit Fonts

2.7.1 Default-Font



Abbildung 2.27: Textausgabe mit Fonts

Bei vielen Spielen werden Informationen nicht nur symbolisch auf die Spielfläche gebracht (z.B. drei Männchen für drei Leben), sondern auch in Schriftform. Eine Möglichkeit dies zu erreichen, ist die Textausgabe mit Hilfe installierter Fonts. Dabei wird zuerst ein **Font**-Objekt erstellt und durch ein **Surface**-Objekt mit dem Text erzeugt ([gerendert](#)). Ich habe dies für ein kleines Beispiel in eine Klasse gekapselt, die Sie ja nach Belieben aufbohren oder anpassen können.

Zuerst importieren wir ein paar Konstanten. Die Klasse **Settings** überspringe ich mal, die hat sich nicht verändert:

Quelltext 2.40: Text mit Fonts ausgeben (1), Präambel

```

1 import os
2 from typing import Tuple
3
4 import pygame
5 from pygame.constants import (K_ESCAPE, K_KP_MINUS, K_KP_PLUS, K_MINUS, K_PLUS,
6                               KEYDOWN, KMOD_SHIFT, QUIT, K_b, K_g, K_r)

```

Font
Und nun die Klasse **TextSprite**: Lassen Sie sich nicht vom [OO](#)-Ansatz verwirren. Eigentlich ist alles ganz einfach. Wir brauchen ein **pygame.font.Font**-Objekt. Dieses wiederum braucht zwei Infos: Welchen installierten **Font** es benutzen soll, und die Fontgröße in [pt](#). Eine Möglichkeit zu einem installierten Font zu kommen, ist die Methode **pygame.font.get_default_font()**. Ihr Aufruf in Zeile 33 liefert mir die vom Betriebs-

[get_de-fault_font\(\)](#)

system eingestellte Zeichsatzvorgabe. Die Schriftgröße (`fontsize`) legen wir nach Bedarf einfach fest.

Quelltext 2.41: Text mit Fonts ausgeben (2), `TextSprite`

```

14  class TextSprite(pygame.sprite.Sprite):
15      def __init__(self, fontsize: int, fontcolor: list[int], center: Tuple[int, int], text:
16          str = 'HelloWorld!') -> None:
17          super().__init__()
18          self.image = None
19          self.rect = None
20          self.fontsize = fontsize
21          self.fontcolor = fontcolor
22          self.fontsize_update(0)           # 0!
23          self.text = text
24          self.center = center
25          self.render()                  # Alle Infos zusammen
26
26  def render(self) -> None:
27      self.image = self.font.render(self.text, True, self.fontcolor)  # Bitmap
28      self.rect = self.image.get_rect()
29      self.rect.center = self.center
30
31  def fontsize_update(self, step: int = 1) -> None:
32      self.fontsize += step
33      self.font = pygame.font.Font(pygame.font.get_default_font(), self.fontsize)  #
34
35  def fontcolor_update(self, delta: Tuple[int, int, int]) -> None:
36      for i in range(3):
37          self.fontcolor[i] = (self.fontcolor[i] + delta[i]) % 256
38
39  def update(self) -> None:
40      self.render()

```

Schauen wir uns nun den Konstruktor etwas genauer an. Die Attribute `image` und `rect` werden hier einfach schonmal als Dummies angelegt; könnte man auch lassen. Nachdem ich die übergebenen Informationen über Textgröße und -farbe in Attribute abgespeichert habe, kann ich das Font-Objekt erstellen lassen. Dies erfolgt durch den Aufruf von `fontsize_update()` in Zeile 21. Durch die Angabe 0 wird klar, dass hier nicht die Größe verändert werden soll, sondern nur, dass die Objekterzeugung passiert.

Nun merke ich mir den eigentlichen Text, der zu einem Schriftzug gerendert werden soll und, wo das Zentrum des Schriftzugs platziert wird. Jetzt habe ich alle Infos zusammen und kann durch Aufruf von `render()` in Zeile 24 mit Hilfe von `pygame.font.render()` das Surface-Objekt erzeugen (Zeile 27). Anschließend wird vom Bitmap das Rechteck ermittelt und das Zentrum des Rechtecks auf die gewünschte Position verschoben.

`render()`

Jetzt noch die zwei Methoden `fontsize_update()` und `fontcolor_update()`: Beide ermöglichen es mir, zur Laufzeit die Schriftgröße und -farbe zu ändern. Die Semantik sollte selbsterklärend sein.

Wie kann man nun so eine Klasse nutzen? Hier ein Beispiel. In der Mitte soll ein Gruß erscheinen. Dazu verwende ich das Objekt `hello` (Zeile 49). Darunter soll durch `info` ausgegeben werden, mit welcher Schriftgröße und -farbe der Gruß erzeugt wurde (Zeile 49).

Quelltext 2.42: Text mit Fonts ausgeben (3), Hauptprogramm

```

43 def main():
44     pygame.init()
45     clock = pygame.time.Clock()
46     screen = pygame.display.set_mode(Settings.WINDOW.size)
47     pygame.display.set_caption("Textausgabe mit Fonts")
48
49     hello = TextSprite(24, [255, 255, 255], (Settings.WINDOW.center)) # Gruß
50     info = TextSprite(12, [255, 0, 0], (Settings.WINDOW.centerx, Settings.WINDOW.bottom-20))
51         # Fontinfo
52     all_sprites = pygame.sprite.Group()
53     all_sprites.add(hello, info)
54
55     running = True
56     while running:
57         for event in pygame.event.get():
58             if event.type == QUIT:
59                 running = False
60             elif event.type == KEYDOWN:
61                 if event.key == K_ESCAPE:
62                     running = False
63                 elif event.key == K_KP_PLUS or event.key == K_PLUS:      # Größer
64                     hello.fontsize_update(+1)
65                 elif event.key == K_KP_MINUS or event.key == K_MINUS: # Kleiner
66                     hello.fontsize_update(-1)
67                 elif event.key == K_r:
68                     if event.mod & KMOD_SHIFT:
69                         hello.fontcolor_update((-1, 0, 0))           # Weniger Rot
70                     else:
71                         hello.fontcolor_update((+1, 0, 0))           # Mehr Rot
72                 elif event.key == K_g:
73                     if event.mod & KMOD_SHIFT:
74                         hello.fontcolor_update((0, -1, 0))           # Weniger Grün
75                     else:
76                         hello.fontcolor_update((0, +1, 0))           # Mehr Grün
77                 elif event.key == K_b:
78                     if event.mod & KMOD_SHIFT:
79                         hello.fontcolor_update((0, 0, -1))           # Weniger Blau
80                     else:
81                         hello.fontcolor_update((0, 0, +1))           # Mehr Blau
82
83             info.text = f"size={hello.fontsize}, r={hello.fontcolor[0]}, g={hello.fontcolor[1]}, b={hello.fontcolor[2]}"
84             all_sprites.update()
85             screen.fill((200, 200, 200))
86             all_sprites.draw(screen)
87             pygame.display.flip()
88             clock.tick(Settings.FPS)
89
90     pygame.quit()

```

Dieser Gruß kann durch die Plus- und Minus-Tasten in seiner Größe verändert werden (Zeile 62ff.). Die Tasten **r**, **g** und **b** werden dazu verwendet, den jeweiligen Farbkanal zu manipulieren. Der Großbuchstabe erhöht den Wert (z.B. in Zeile 68), der Kleinbuchstabe reduziert ihn (z.B. in Zeile 70).

In Abbildung 2.27 auf Seite 62 können Sie eine mögliche Darstellung sehen.

2.7.2 Fontliste

Als weiteres Beispiel möchte ich Ihnen ein kleines Programm zeigen, welches alle installierten Fonts auflistet. Vielleicht kann man sich ja dabei Gestaltungsideen holen. Der erste Teil sollte keine Verständnisprobleme mehr bereiten.



Abbildung 2.28: Fontliste

Quelltext 2.43: Fontliste (1), Präambel, Settings und TextSprite

```

1 import os
2
3 import pygame
4 from pygame.constants import K_DOWN, K_ESCAPE, K_UP, KEYDOWN, QUIT
5
6
7 class Settings:
8     WINDOW = pygame.Rect((0, 0), (700, 300))
9     FPS = 60
10
11
12 class TextSprite(pygame.sprite.Sprite):
13     def __init__(self, fontname: str, fontsize: int = 24, fontcolor: list[int] = [255, 255,
14         255], text: str = '') -> None:
15         super().__init__()
16         self.image = None
17         self.fontname = fontname
18         self.fontsize = fontsize
19         self.fontcolor = fontcolor
20         self.fontsize_update(0)
21         self.text = f"{self.fontname}: abcdefghijklmnopqrstuvwxyzßöäü0123456789"
22         self.render()
23
24     def render(self) -> None:
25         self.image = self.font.render(self.text, True, self.fontcolor)
26         self.rect = self.image.get_rect()
27
28     def fontsize_update(self, step: int = 1) -> None:

```

```

28         self.fontsize += step
29         self.font = pygame.font.Font(pygame.font.match_font(self.fontname), self.fontsize)
30         #
31     def fontcolor_update(self, delta: list[int]) -> None:
32         for i in range(3):
33             self.fontcolor[i] = (self.fontcolor[i] + delta[i]) % 256
34
35     def update(self) -> None:
36         self.render()

```

Die Klasse `TextSprite` wurde nur wenig auf die Bedürfnisse angepasst. Die Klasse `BigImage` hat die Aufgabe, alle `FontSprite`-Images als großes Bild zu verwalteten. Später wird immer ein Ausschnitt aus dem Bitmap auf den Bildschirm gedruckt. Der Ausschnitt orientiert sich an der Position innerhalb der Liste und wird durch das Attribut `offset` gesteuert und in der Methode `update()` (Zeile 49) ermittelt. Zuerst wird ermittelt, ob ich das obere oder untere Ende des Bitmaps erreicht habe. Falls ja, wird `top` bzw. `bottom` entsprechend gesetzt, so dass immer der ganze Bildschirm gefüllt wird. Ansonsten wird das `offset`-Rechteck nach oben bzw. nach unten verschoben und mit `pygame.Surface.subsurface()` der Ausschnitt ermittelt.

`subsurface()`

Quelltext 2.44: Fontliste (2), BigImage

```

39 class BigImage(pygame.sprite.Sprite):
40     def __init__(self):
41         super().__init__()
42         self.offset = pygame.Rect((0, 0), Settings.WINDOW.size)
43
44     def create_image(self, width: int, height: int) -> None:
45         self.image_total = pygame.Surface((width, height))
46         self.image_total.fill((200, 200, 200))
47         self.update(0)
48
49     def update(self, delta: int) -> None:                                     # Ermittle der Ausschnitt
50         if self.offset.top + delta >= 0:
51             if self.offset.bottom + delta <= self.image_total.get_rect().height:
52                 self.offset.move_ip(0, delta)
53             else:
54                 self.offset.bottom = self.image_total.get_rect().height
55             else:
56                 self.offset.top = 0
57             self.image = self.image_total.subsurface(self.offset)
58             self.rect = self.image.get_rect()

```

`get_fonts()`
`match_font()`

Und jetzt das Hauptprogramm. Im ersten Teil wird über `pygame.font.get_fonts()` (Zeile 68) eine Liste aller installierten Fontnamen ermittelt. Dieser Name wird dem Konstruktor von `TestSprite` übergeben. Mit Hilfe der Methode `pygame.font.match_font()` (Zeile 29) wird nun der Font selbst im System gesucht, wobei sich diese Methode zunutze macht, dass der Name der Fontdatei sich aus dem Fontnamen und der Endung `ttf` herleiten lässt.

Quelltext 2.45: Fontliste (3), Hauptprogramm (1)

```

61 def main():
62
63     pygame.init()
64     clock = pygame.time.Clock()
65     screen = pygame.display.set_mode(Settings.WINDOW.size)
66     pygame.display.set_caption("Fontliste")
67
68     fonts = pygame.font.get_fonts()                      # Ermittle installierte Fonts
69
70     list_of_fontsprites = pygame.sprite.Group()
71     height = 0
72     width = 0
73     for name in fonts:
74         try:
75             t = TextSprite(name, 24, [0, 0, 255])
76             t.rect.top = height
77             height += t.rect.height
78             width = t.rect.width if t.rect.width > width else width
79             list_of_fontsprites.add(t)
80         except OSError as err:
81             print(f"OS\u201eerror\u201e{err}\u201e")
82         except pygame.error as perr:
83             print(f"Pygame\u201eerror:\u201e{perr}\u201ewith\u201efont\u201e{name}\u201e")
84
85     bigimage = pygame.sprite.GroupSingle(BigImage())
86     bigimage.sprite.create_image(width, height)
87     list_of_fontsprites.draw(bigimage.sprite.image_total)  #

```

In der `for`-Schleife_ werden nun für alle Fonts `TextSprite`-Objekte erzeugt und deren Höhe und Breite ermittelt. Diese vielen Bitmaps werden dann auf das große Bitmap gedruckt (Zeile 87).

Quelltext 2.46: Fontliste, Hauptprogramm (2)

```

89     running = True
90     while running:
91         for event in pygame.event.get():
92             if event.type == QUIT:
93                 running = False
94             elif event.type == KEYDOWN:
95                 if event.key == K_ESCAPE:
96                     running = False
97                 if event.key == K_UP:
98                     bigimage.update(-Settings.WINDOW.height//3)
99                 if event.key == K_DOWN:
100                     bigimage.update(Settings.WINDOW.height//3)
101
102             bigimage.draw(screen)
103             pygame.display.flip()
104             clock.tick(Settings.FPS)
105
106     pygame.quit()

```

Die Hauptprogrammschleife übernimmt nun nur noch das Blättern (jeweils um eine drittel Bildschirmhöhe) und das Programmende.

Was war neu?

Es wurden folgende Pygame-Elemente eingeführt:

- `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>

2.8 Textausgabe mit Bitmaps

Oft erfolgen Textausgaben nicht über Fonts, sondern über eine [Spritelib](#). In einer solchen befinden sich dann Schriftzeichen, Symbole oder Ziffern, die dann meist auch in einem besonderen dem Spiel angepassten Design sind. In Abbildung 2.29 finden Sie eine Spritelib, die Sprites für ein Kampfspiel des 2. Weltkriegs zur Verfügung stellt. Unter anderem sind dort die Sprites für die Ziffern 0 – 9 und die Buchstaben des lateinischen Alphabets zu finden. Ein Vorteil dieses Vorgehens ist, dass das Vorhandensein des Spielfonts nicht vorausgesetzt werden muss. Wenn Sie also die Textausgabe mit dem Font *Calibri* durchführen, muss dieser Font ja auf dem Zielrechner installiert sein. Nachteil ist, dass sich Bitmaps meist nur sehr schlecht skalieren lassen und dann kaum Schriften verschiedener Größen zur Verfügung stehen.

Die Idee ist nun, die einzelnen Buchstaben aus der Spritelib auszustanzen und in einer geschickten Datenstruktur abzulegen. Soll nun ein Text ausgegeben werden, wird der Text in seine Buchstaben zerlegt und die dazu passenden Buchstabensprites aus der Datenstruktur auf ein Zielbitmap – beispielsweise Screen – ausgegeben. Ich möchte das ganze hier an einem einfachen Beispiel aufzeigen. Basis ist eine Spritelib mit einem Zeichensatz in fünf verschiedenen Farben (siehe Abbildung 2.31 auf Seite [74](#)).

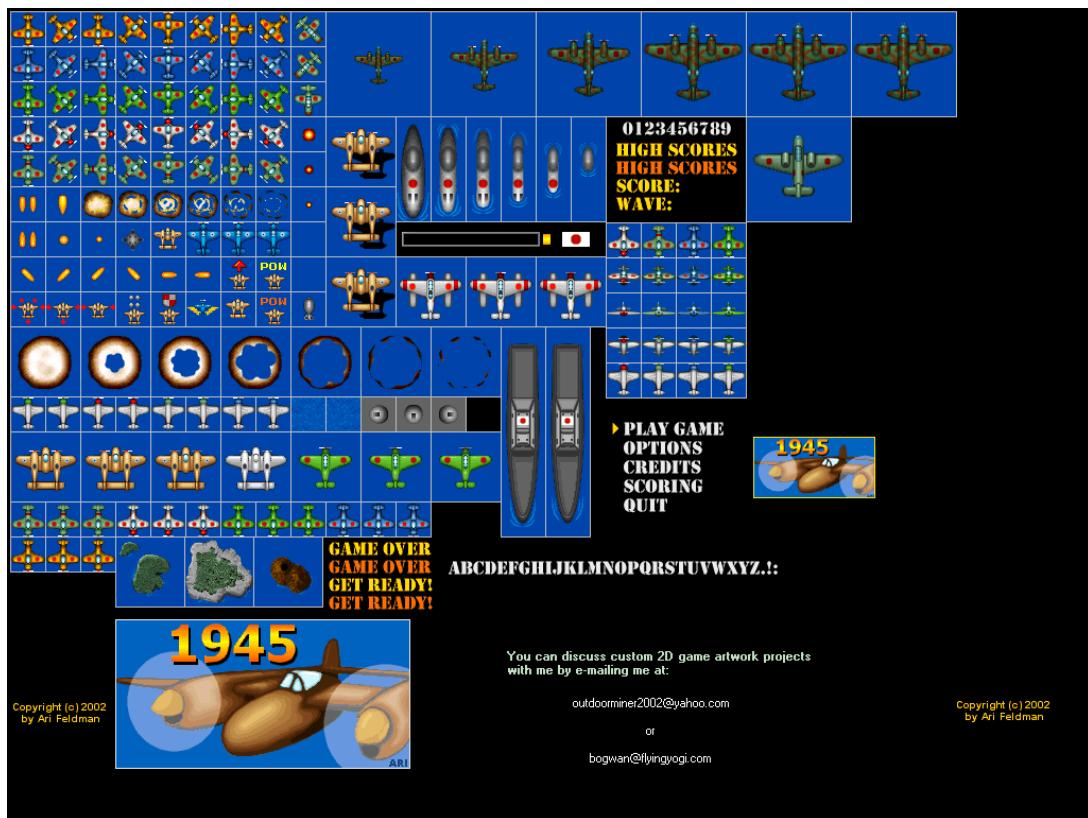


Abbildung 2.29: Beispiel für eine Spritelib

Der erste Teil von Quelltext 2.47 sollte bekannt vorkommen und ist nur um einige Bequemlichkeiten erweitert worden. Die Pfadangaben lasse ich mir nun in den statischen Methoden `filepath()` und `imagepath()` ermitteln.

Quelltext 2.47: Textbitmaps (1), Präambel und `Settings`

```

1 import os
2 from typing import Tuple
3
4 import pygame
5 from pygame import K_ESCAPE, KEYDOWN, QUIT
6
7
8 class Settings:
9
10    WINDOW = pygame.Rect((0, 0), (700, 650))
11    PATH: dict[str, str] = {}
12    PATH['file'] = os.path.dirname(os.path.abspath(__file__))
13    PATH['image'] = os.path.join(PATH['file'], "images")
14    FPS = 60
15
16    @staticmethod
17    def filepath(name: str) -> str:
18        return os.path.join(Settings.PATH['file'], name)
19
20    @staticmethod
21    def imagepath(name: str) -> str:
22        return os.path.join(Settings.PATH['image'], name)

```

Die Klasse `Spritelib` wird eigentlich nur als Container gebraucht. Sie lädt sich die Spritelib der Buchstaben und Symbole und enthält einige Angaben, die ich brauche, um ganz gezielt einzelne Buchstaben oder Symbole auszustanzen:

- **nof:** Enthält die Anzahl der Zeilen und Spalten. Unser Symbolsatz ist im Bitmap in 4 Zeilen und 10 Spalten angeordnet. Da ich mich immer nur für eine Farbe interessiere, reicht mir das.
- **letter:** Jedes Sprite hat eine Breite und eine Höhe. In unserem Fall kommt erleichternd hinzu, dass alle Sprites immer den gleichen Platzbedarf haben; schauen Sie sich dazu die drei Quadrate um die Buchstaben **N**, **W** und **X** in Abbildung 2.30 auf der nächsten Seite an. Unsere Sprites haben alle eine Breite und eine Höhe von 18 *px*.
- **offset:** Das erste Sprite oben links hat einen Abstand vom linken Rand und einen vom oberen Rand. Schauen Sie sich dazu das Sprite der Zahl 0 in Abbildung 2.30 an. Dort haben wir das Quadrat um das Bitmap und zwischen dem Quadrat und der oberen bzw. der linken Kante einen Abstand (markiert durch die grüne Linie). Beide Offsets haben in unserem Beispiel einen Wert von 6 *px*.
- **distance:** Jedes Sprite hat einen Abstand zum nächsten Sprite nach rechts und nach unten. Zum Glück sind unsere Sprites äquidistant in der Spritelib abgelegt, so dass ich es hier recht einfach habe. Am Beispiel des Sprites für **X** in Abbildung 2.30 können Sie die Abstände sehen. Hier sind es jeweils 14 *px*.

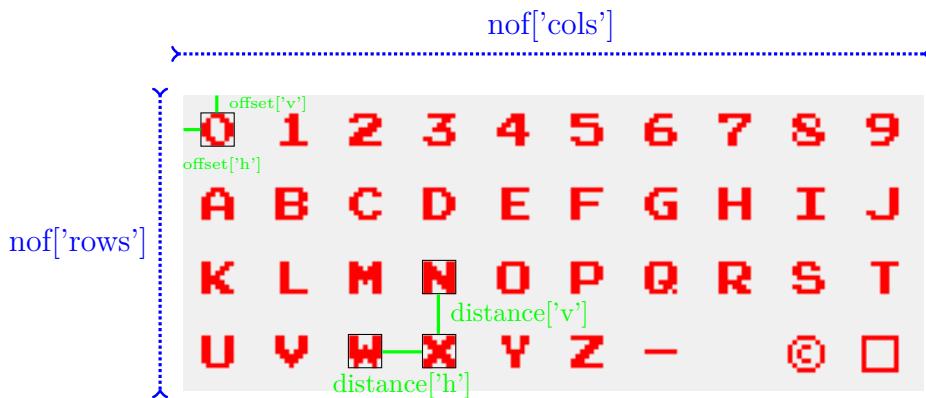


Abbildung 2.30: Bedeutung der Angaben in Spritelib

Quelltext 2.48: Textbitmaps (2), Spritelib

```

25 class Spritelib(pygame.sprite.Sprite):
26
27     def __init__(self, filename: str) -> None:
28         super().__init__()
29         self.image = pygame.image.load(Settings.imagepath(filename)).convert()
30         self.rect = self.image.get_rect()
31         self.nof = {'rows': 4, 'cols': 10}
32         self.letter = {'width': 18, 'height': 18}
33         self.offset = {'h': 6, 'v': 6}
34         self.distance = {'h': 14, 'v': 14}
35
36     def draw(self, screen: pygame.surface.Surface) -> None:
37         screen.blit(self.image, self.rect)

```

Kommen wir jetzt zur eigentlich interessanten Klasse: `Letters`. Diese stanzt aus der Spritelib alle Sprites einer Farbe aus und stellt sie in einem `Dictionary` als `Surface`-Objekte zur Verfügung. Dabei wird eine Menge rumgerechnet, was Sie aber nicht abschrecken sollte; es ist letztlich Grundschulmathematik. Fangen wir mit dem Konstruktor an. Der Konstruktor hat zwei Übergabeparameter: Der erste Parameter `spritelib` ist ein Verweis auf das Spritelib-Objekt, welches das originale Bitmap geladen hat und einige Abstandsinformationen enthält. Der zweite Parameter `colornumber` ermöglicht es mir später nur für eine Farbe den vollständigen Symbolsatz auszulesen: 0 steht für die weißen Sprites, 1 für die gelben usw..

Quelltext 2.49: Textbitmaps (3): Konstruktor von `Letters`

```

40 class Letters(object):
41
42     def __init__(self, spritelib: Spritelib, colornumber: int) -> None:
43         super().__init__()
44         self.spritelib = spritelib
45         self.letters: dict[str, pygame.surface.Surface] = {}
46         self.create_letter_bitmap(colornumber)

```

In der Methode `create_letter_bitmap()` werden nun die einzelnen Sprites ausgestanzt und in ein Dictionary abgelegt. Die Indizes des Dictionaries werden in Zeile 52 definiert. Hier muss die Reihenfolge natürlich der entsprechen, mit der man die Sprites ausstanzt. Die Variable `index` sorgt genau dafür, dass bei jedem Schleifendurchlauf der nächste `lettername` als Schlüssel für das Dictionary verwendet wird.

In Zeile 55 wird die Position, also die Pixelkoordinaten des ersten Sprites ausgerechnet. Versuchen Sie doch selbst anhand der Angaben in Abbildung 2.30 auf der vorherigen Seite die Arithmetik nachzuvollziehen! Nur Mut, sie ist nicht schwierig, sondern nur lang.

Ab Zeile 56 beginnt eine verschachtelte `for`-Schleife. Die äußere Schleife durchläuft alle Zeilen der Spritelib und die innere die Spalten. Ziel dieser Konstruktion ist es, für jedes Sprite ein `Rect`-Objekt zu erzeugen, in welchem ich die Position und die Größe des Sprites abspeichere. In Zeile 57 wird die obere Koordinate und in Zeile 59 die linke Koordinate der Position berechnet. Wenn Sie Zeile 55 verstanden haben, sollten diese beiden Berechnungen keine Schwierigkeiten mehr bereiten. Höhe und Breite in Zeile 59 sind einfach, da alle Sprites immer die gleichen Größen haben. Anschließend wird das `Rect`-Objekt erzeugt und zum Ausstanzen des Bitmap mit Hilfe von `subsurface()` verwendet. Dieses ausgestanzte Bitmap wird dann unter seinem Symbolnamen im Dictionary abgelegt.

Quelltext 2.50: Textbitmaps (4): `create_letter_bitmap()` von Letters

```

48     def create_letter_bitmap(self, colordnumber: int):
49         lettername = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
50                     'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
51                     'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
52                     'u', 'v', 'w', 'x', 'y', 'z', '—', '„', 'copy', 'square') #
53         index = 0
54         startpos = (self.spritelib.offset['h'], self.spritelib.offset['v'] + colordnumber *
55                     self.spritelib.nof['rows'] *
56                     * (self.spritelib.letter['height'] + self.spritelib.distance['v'])) #
57         for row in range(self.spritelib.nof['rows']):                      # Zeilen
58             for col in range(self.spritelib.nof['cols']):                      # Spalten
59                 left = startpos[0] + col * (self.spritelib.letter['width'] +
60                     self.spritelib.distance['h']) #
61                 top = startpos[1] + row * (self.spritelib.letter['height'] +
62                     self.spritelib.distance['v']) #
63                 width, height = self.spritelib.letter.values()      # Größe
64                 r = pygame.rect.Rect(left, top, width, height)
65                 self.letters[lettername[index]] = self.spritelib.image.subsurface(r) #
66                 index += 1

```

Die Methode `get_text()` liefert mir letztlich die passende Bitmap-Folge zu einem Text. Dabei bedient sie sich der Methode `get_letter()`, die notwendig ist, damit das Programm nicht bei undefinierten Buchstaben/Symbolen abstürzt. Wenn Sie jetzt beispielsweise ein ü eintippen, wird das Quadrat ausgegeben.

Quelltext 2.51: Textbitmaps (5): `get_letter()` und `get_text()` von Letters

```

65     def get_letter(self, letter: str) -> pygame.surface.Surface:
66         if letter in self.letters:
67             return self.letters[letter]

```

```

68     else:
69         return self.letters['square']
70
71     def get_text(self, text: str) -> pygame.surface.Surface:
72         l = len(text) * self.spritelib.letter['width']
73         h = self.spritelib.letter['height']
74         bitmap = pygame.Surface((l, h))
75         bitmap.set_colorkey((0, 0, 0))
76         for a in range(len(text)):
77             bitmap.blit(self.get_letter(text[a]), (a * self.spritelib.letter['width'], 0))
78

```

Das eigentliche Hauptprogramm ist in der Klasse `TextBitmaps` gekapselt. Da die Quelltexte hier nichts neues beinhalten, sollte der Quelltext verstanden werden. Nur zwei Zeilen möchte ich näher besprechen:

- Zeile 100: Hier wird das **Slicing** von **Arrays** verwendet. Die Angabe `-1` bewirkt, dass der Ende-Zeiger des Slice beim letzten Element startet und dann einen Schritt nach links geht. Das Ergebnis ist ein um das letzte Zeichen gekürzter neuer String.
- Zeile 102: Das Attribut `unicode` liefert mir, sofern dies sinnvoll ist, den Wert der gedrückten Tastatur im **Unicode**-Format. Somit werden sinnvolle Buchstaben, Ziffern usw. als Zeichen meinem String hinzugefügt.

`unicode`

Quelltext 2.52: Textbitmaps (6): `TextBitmaps`

```

81 class TextBitmaps(object):
82
83     def __init__(self) -> None:
84         pygame.init()
85         self.screen = pygame.display.set_mode(Settings.WINDOW.size)
86         pygame.display.set_caption('Textausgabe mit Bitmaps')
87         self.clock = pygame.time.Clock()
88         self.filename = "chars.png"
89         self.running = False
90         self.input = ""
91
92     def watch_for_events(self) -> None:
93         for event in pygame.event.get():
94             if event.type == QUIT:
95                 self.running = False
96             elif event.type == KEYDOWN:
97                 if event.key == K_ESCAPE:
98                     self.running = False
99                 elif event.key == pygame.K_BACKSPACE:
100                     self.input = self.input[:-1]           # Letztes Zeichen abschneiden
101                 else:
102                     self.input += event.unicode        # Tastaturwert als unicode-Zeichen
103
104     def run(self) -> None:
105         spritelib = Spritelib(self.filename)
106         letters = Letters(spritelib, 2)
107         self.running = True
108         while self.running:
109             self.watch_for_events()
110             self.screen.fill((200, 200, 200))
111             self.screen.blit(letters.get_text(self.input), (400, 200))
112             spritelib.draw(self.screen)
113             pygame.display.flip()
114             self.clock.tick(Settings.FPS)

```

115
116

```
    pygame.quit()
```

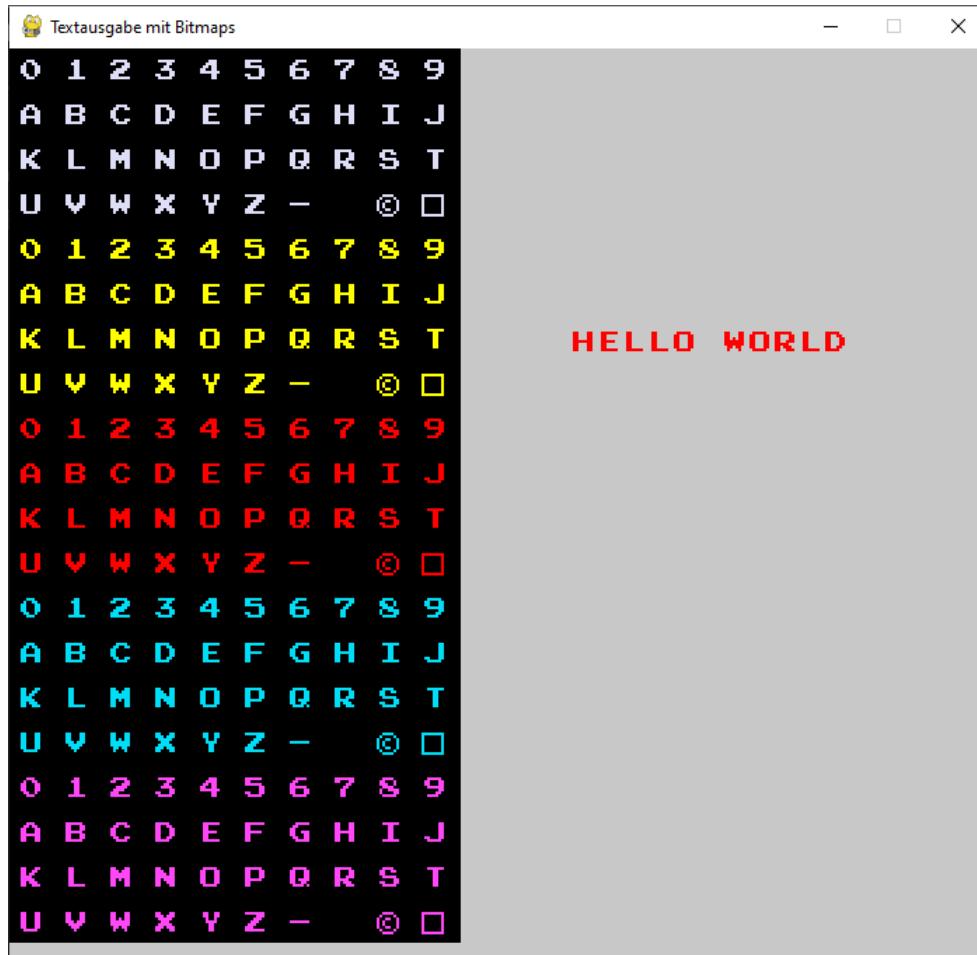


Abbildung 2.31: Textausgabe mit Bitmaps

Was war neu?

Textausgaben werden nicht nur über Fonts erzeugt, sondern auch über Spritlibs, die Zeichenbitmaps enthalten. Diese werden ausgestanzt und neuen Bitmaps zusammengesetzt.

Es wurden folgende Pygame-Elemente eingeführt:

- `pygame.event.Event.unicode`:
<https://pyga.me/docs/ref/event.html>
- `pygame.Surface.subsurface()`:
<https://pyga.me/docs/ref/surface.html#pygame.Surface.subsurface>

2.9 Kollisionserkennung

Kollisionserkennung wird in der Spieleprogrammierung oft gebraucht: Personen können nicht durch Hindernisse gehen, Geschosse treffen auf Ziele, Bälle prallen ab usw.. Deshalb stellt Pygame einen ganzen Blumenstrauß von Kollisionserkennungen zur Verfügung:

- **Rechtecküberschneidung:** Wir haben schon bei der Betrachtung der `Sprite`-Klasse gesehen, dass das Attribut `rect` notwendig ist. Dieses enthält die Positions- und Größenangaben des umgebenen Rechtecks. Treffen nun zwei Sprites aufeinander, wird überprüft, ob sich die beiden Rechtecke überschneiden. Dies ist eine sehr *billige* Erkennungsmethode, da mit wenigen Vergleichen entschieden werden kann, ob sich zwei Rechtecke treffen/überlappen. Hier eine beispielhafte Programmierung:

```

1 def rectangleCollision(rect1, rect2):
2     return rect1.left < rect2.right and
3             rect2.left < rect1.right and
4             rect1.top < rect2.bottom and
5             rect2.top < rect1.bottom

```

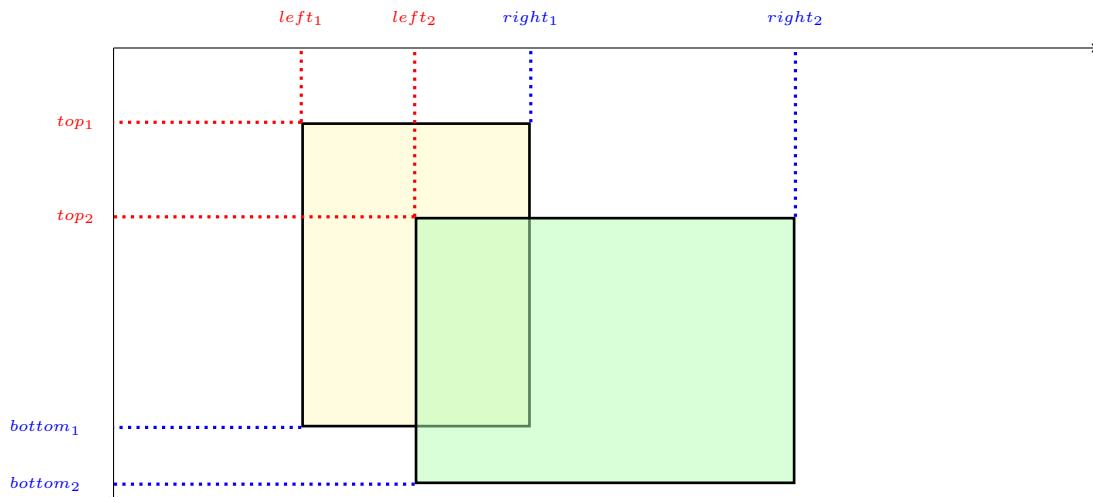


Abbildung 2.32: Kollisionserkennung mit Rechtecken

- **Kreisüberschneidung:** Bei eher runden Sprites empfiehlt es sich, nicht die Rechtecke zu überprüfen, sondern den Innenkreis zur Kollisionsprüfung zu verwenden. Auch diese Kollisionsprüfung ist recht schnell, da nur ein Vergleich auf den Abstand der Mittelpunkte erfolgen muss: $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} < r_1 + r_2$.

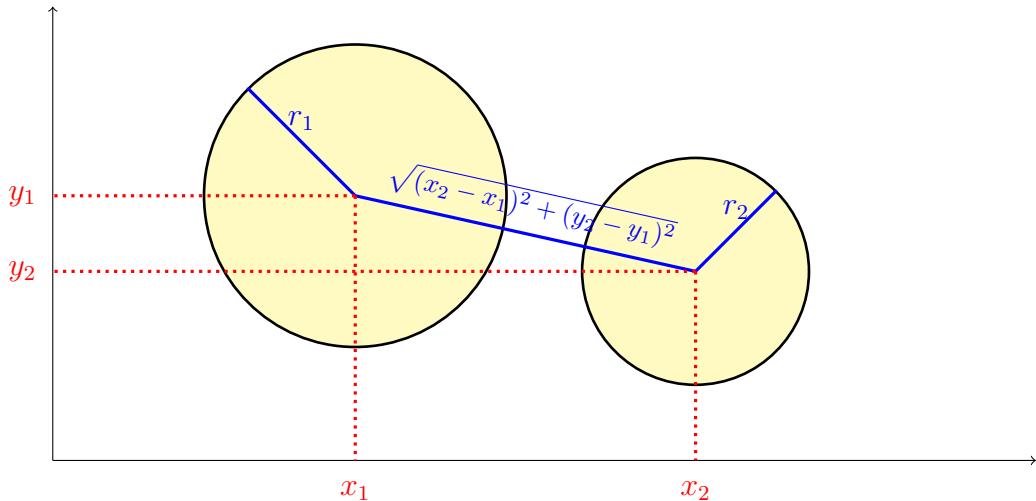


Abbildung 2.33: Kollisionserkennung mit Kreisen

- **Pixelüberschneidung:** Bei der pixelgenauen Überschneidung wird für jedes Pixel der beiden Sprites überprüft, ob sie die gleiche Position haben. Wenn *Ja* überschneiden sie sich, wenn *Nein* nicht. Dies ist die teuerste Kollisionsprüfung, aber auch die genaueste. Um den Aufwand zu reduzieren, wird zuerst das Schnittmengen-Rechteck der beiden Sprites ermittelt. Wie bei der Rechteckprüfung wird dabei erstmal gecheckt, ob die beiden Rechtecke sich überschneiden. Wenn nicht, bin ich sofort fertig. Wenn doch, muss die Schnittmenge der beiden Rechtecke wiederum ein Rechteck sein. Wenn nun zwei Pixel die gleiche Position haben, müssen diese innerhalb des Schnittmengen-Rechtecks liegen und die Pixel-Prüfung kann auf diesen in der Regel viel kleineren Bereich eingeschränkt werden. Ein weiteres Problem bei der Pixelprüfung ist, Hintergrund von Vordergrund zu unterscheiden. Woher soll die Pixelprüfung wissen, ob die Farbe Blau nun ein Teil des Objektes oder des Hintergrunds ist? Dazu gibt es mehrere Ansätze. Der einfachste ist, zu jedem Sprite ein schwarz/weiß-Bild zu erstellen (ein **Maske**); die weißen Pixel sind wichtig, die schwarzen können ignoriert werden. Nun wird die Pixelprüfung nur noch auf den Masken durchgeführt.

Maske

Schauen wir uns das Kollisionsverhalten mal im Detail an. In Abbildung 2.34 auf der nächsten Seite sehen wir vier Sprites: eine Mauer, ein Raumschiff, ein Monster und ein Geschoss. Keine der Sprites berühren sich.

In Abbildung 2.35 auf der nächsten Seite erkennen Sie gut den Effekt einer Kollisionserkennung durch die umgebenden Rechtecke. Bei der Mauer ist alles perfekt. Das Geschoss trifft die Mauer und durch die Farbgebung wird signalisiert, dass die Kollision vom Programm erkannt wurde. Den Nachteil sehen wir aber beim Raumschiff. Dort wird auch eine Kollision erkannt, obwohl sich die beiden Sprites nicht berühren. Aber das umgebende Rechteck des Raumschiffs umschließt die leeren Flächen in den Ecken, so dass eine Kollision erkannt wird. Beim Monster kann das ebenfalls beobachtet werden.

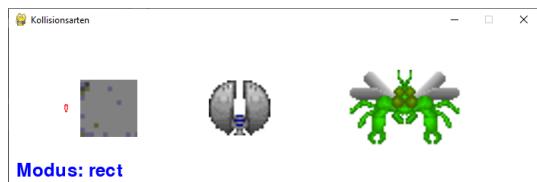
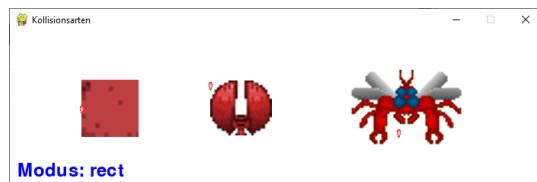


Abbildung 2.34: Vier Sprites

Abbildung 2.35: Rechtecksprüfung
(Montage)

Anders sieht es aus, wenn wir die Kollision durch die Innenkreise bestimmen lassen (Abbildung 2.36). Jetzt wird die Kollision bei der Mauer nicht mehr richtig erkannt, da die Ecken nicht mehr zum Innenkreis gehören. Beim Raumschiff hingegen liefert diese Methode genau das gewünschte Ergebnis, da die leeren Ecken nicht zum Innenkreis gehören. Würden wir nun etwas weiter nach rechts gehen, würde auch das Raumschiff rot werden, da eine Kollision erkannt wird. Das Monster liefert immer noch ein falsches Ergebnis.

Verbleibt noch die pixelgenaue Prüfung (Abbildung 2.37). Die Kollision mit der Mauer wird richtig erkannt. Erstaunlicher sind die beiden Ergebnisse beim Raumschiff und beim Monster. Beide erkennen richtig keine Kollision, da das Geschoss sich zwar innerhalb des Rechtecks und des Innenkreises befindet, aber nur auf transparenten Pixel. Probieren Sie es ruhig aus, das Geschoss mal nach rechts bzw. links zu bewegen, und Sie werden die pixelgenaue Kollisionserkennung anhand des Farbwechsels sofort sehen.

Abbildung 2.36: Kreisprüfung
(Montage)Abbildung 2.37: Maskenprüfung
(Montage)

Schauen wir uns jetzt den dazugehörigen Quelltext genauer an, wobei ich auf eine nochmalige Besprechung der Präambel und von **Settings** verzichten möchte.

Quelltext 2.53: Kollisionsarten (1): Präambel und **Settings**

```

1 import os
2 from typing import Any, Tuple
3
4 import pygame
5
6
7 class Settings(object):
8
9     WINDOW = pygame.rect.Rect((0, 0), (700, 200))
10    FPS = 60
11    TITLE = "Kollisionsarten"
12    PATH: dict[str, str] = {}
13    PATH['file'] = os.path.dirname(os.path.abspath(__file__))

```

```

14     PATH['image'] = os.path.join(PATH['file'], "images")
15     MODUS = "rect"
16
17     @staticmethod
18     def filepath(name: str) -> str:
19         return os.path.join(Settings.PATH['file'], name)
20
21     @staticmethod
22     def imagepath(name: str) -> str:
23         return os.path.join(Settings.PATH['image'], name)

```

Interessanter wird es beim `Obstacle`. Dies ist die Klasse für die Mauer, das Raumschiff und das Monster. Für die Rechteckprüfung wird das umgebende Rechteck benötigt, welches in Zeile 33 wir gewohnt mit Hilfe von `pygame.Surface.get_rect()` ermitteln und in das Attribut `rect` ablegen. Für Sprites mit impliziter oder einer durch `set_colorkey()` expliziten Transparenz kann die Maske sehr einfach mit `pygame.mask.from_surface()` bestimmt werden (Zeile 34). Damit die vordefinierten Funktionen zur Kollisionserkennung greifen können, muss diese Maske im `Sprite`-Objekt im Attribut `mask` abgelegt werden. In Zeile 35 wird der Innenradius berechnet. Dies ist etwas unsauber implementiert. Eigentlich müsste man das Minimum von Breite und Höhe ermitteln und dieses halbieren. Wie bei der Maske muss auch der Radius in einem Attribut abgelegt werden, damit die vordefinierten Kollisionsmethoden arbeiten können: `radius`.

Das Flag `hit` wird nur dafür gebraucht, damit je nach erkannter Kollision das richtige Image ausgegeben wird, denn – Sie haben es sicherlich schon gesehen – es werden für dieses Sprites zwei Bilder geladen: eines für den Zustand *nicht getroffen* und eines für *getroffen*.

Quelltext 2.54: Kollisionsarten (2): `Obstacle`

```

26 class Obstacle(pygame.sprite.Sprite):
27
28     def __init__(self, filename1: str, filename2: str) -> None:
29         super().__init__()
30         self.image_normal = pygame.image.load(Settings.imagepath(filename1)).convert_alpha()
31         self.image_hit = pygame.image.load(Settings.imagepath(filename2)).convert_alpha()
32         self.image = self.image_normal
33         self.rect = self.image.get_rect()                      # Rechteck
34         self.mask = pygame.mask.from_surface(self.image)      # Maske
35         self.radius = self.rect.centerx                      # Innenkreis
36         self.rect.centery = Settings.WINDOW.centery
37         self.hit = False
38
39     def update(self, *args: Any, **kwargs: Any) -> None:
40         if "hit" in kwargs.keys():
41             self.hit = kwargs["hit"]
42             self.image = self.image_hit if (self.hit) else self.image_normal

```

Die Klasse `Bullet` ähnelt in Vielem der Klasse `Obstacle`. Da wir auch diese Klasse für die drei Kollisionsprüfungsarten verwenden wollen, brauchen wir auch hier die drei Attribute `rect`, `radius` und `mask`. Daneben ist die Klasse mit einigen Zeilen versehen, um das Bullet bewegen zu können; sollte auch selbsterklärend sein. Hinweis: Der Einfachheit halber habe ich keine Randprüfung mit eingebaut. Warum auch.

Quelltext 2.55: Kollisionsarten (3): Bullet

```

45 class Bullet(pygame.sprite.Sprite):
46
47     def __init__(self, picturefile: str) -> None:
48         super().__init__()
49         self.image = pygame.image.load(Settings.imagepath(picturefile)).convert_alpha()
50         self.rect = self.image.get_rect()
51         self.radius = self.rect.centery
52         self.mask = pygame.mask.from_surface(self.image)
53         self.rect.center = (10, 10)
54         self.directions = {'stop': (0, 0), 'down': (0, 1), 'up': (0, -1), 'left': (-1, 0),
55                           'right': (1, 0)}
56         self.set_direction('stop')
57
58     def update(self, *args: Any, **kwargs: Any) -> None:
59         if "action" in kwargs.keys():
60             if kwargs["action"] == "move":
61                 self.rect.move_ip(self.speed)
62             elif "direction" in kwargs.keys():
63                 self.set_direction(kwargs["direction"])
64
65     def set_direction(self, direction: str) -> None:
66         self.speed = self.directions[direction]

```

Und jetzt die Klasse `Game`. Im Konstruktor passieren die üblichen Dinge. Besonders erwähnenswert ist hier eigentlich nichts.

Quelltext 2.56: Kollisionsarten (4): Konstruktor von `Game`, Konstruktor

```

68 class Game(object):
69
70     def __init__(self) -> None:
71         super().__init__()
72         os.environ['SDL_VIDEO_WINDOW_POS'] = "10, 50"
73         pygame.init()
74         pygame.display.set_caption(Settings.TITLE)
75         self.font = pygame.font.Font(pygame.font.get_default_font(), 24)
76         self.screen = pygame.display.set_mode(Settings.WINDOW.size)
77         self.clock = pygame.time.Clock()
78         self.bullet = pygame.sprite.GroupSingle(Bullet("shoot.png"))
79         self.all_obstacles = pygame.sprite.Group()
80         self.all_obstacles.add(Obstacle("brick1.png", "brick2.png"))
81         self.all_obstacles.add(Obstacle("raumschiff1.png", "raumschiff2.png"))
82         self.all_obstacles.add(Obstacle("alienbig1.png", "alienbig2.png"))
83         self.running = False

```

Auch die Methoden `run()` und `watch_for_events()` folgen ausgetretenen Pfaden.

Quelltext 2.57: Kollisionsarten (5): `run()` und `watch_for_events()` von `Game`

```

85     def run(self) -> None:
86         self.resize()
87         self.running = True
88         while self.running:
89             self.watch_for_events()
90             self.update()
91             self.draw()
92             self.clock.tick(Settings.FPS)
93         pygame.quit()
94

```

```

95     def watch_for_events(self) -> None:
96         for event in pygame.event.get():
97             if event.type == pygame.QUIT:
98                 self.running = False
99             elif event.type == pygame.KEYDOWN:
100                 if event.key == pygame.K_ESCAPE:
101                     self.running = False
102                 elif event.key == pygame.K_DOWN:
103                     self.bullet.sprite.update(direction='down')
104                 elif event.key == pygame.K_UP:
105                     self.bullet.sprite.update(direction='up')
106                 elif event.key == pygame.K_LEFT:
107                     self.bullet.sprite.update(direction='left')
108                 elif event.key == pygame.K_RIGHT:
109                     self.bullet.sprite.update(direction='right')
110                 elif event.key == pygame.K_r:
111                     Settings.MODUS = "rect"
112                 elif event.key == pygame.K_c:
113                     Settings.MODUS = "circle"
114                 elif event.key == pygame.K_m:
115                     Settings.MODUS = "mask"
116             elif event.type == pygame.KEYUP:
117                 self.bullet.sprite.update(direction='stop')

```

Ebenso so `update()` und `draw()`;

Quelltext 2.58: Kollisionsarten (6): `update()` und `draw()` von Game

```

119     def update(self) -> None:
120         self.check_for_collision()
121         self.bullet.update(action="move")
122         self.all_obstacles.update()
123
124     def draw(self) -> None:
125         self.screen.fill("white")
126         self.all_obstacles.draw(self.screen)
127         self.bullet.draw(self.screen)
128         text_surface_modus = self.font.render(f"Modus: {Settings.MODUS}", True, "blue")
129         self.screen.blit(text_surface_modus, dest=(10, Settings.WINDOW.bottom-30))
130         pygame.display.flip()

```

Die Methode `resize()` hat nichts mit der eigentlichen Kollisionsprüfung zu tun, sondern soll nur sicherstellen, dass die `Obstacle`-Objekte äquidistant auf die Fensterbreite verteilt werden. Die erste `for`-Schleife ermittelt mir die Summe der Breiten der `Obstacle`-Objekte. Diese Info brauche ich, um in Zeile 136 den Abstand auszurechnen. Dazu ziehe ich von der Fensterbreite `total_width` ab. Diese Anzahl an Pixel kann nun auf die Zwischenräume verteilt werden. Und wie viele Zwischenräume haben wir? Zwei zwischen den drei `Obstacle`-Objekten, einen zum linken Rand und einen zum rechten; also sind es insgesamt vier Zwischenräume. Den Abstand merke ich mir in `padding`. Jetzt kann ich in der zweiten `for`-Schleife die linke Position der `Obstacle`-Objekte bestimmen und setzen.

Quelltext 2.59: Kollisionsarten (7): `resize()` von Game

```

132     def resize(self) -> None:
133         total_width = 0
134         for s in self.all_obstacles:

```

```

135         total_width += s.rect.width
136         padding = (Settings.WINDOW.width - total_width) // 4      # Abstand
137         for i in range(len(self.all_obstacles)):
138             if i == 0:
139                 self.all_obstacles.sprites()[i].rect.left = padding
140             else:
141                 self.all_obstacles.sprites()[i].rect.left =
142                     self.all_obstacles.sprites()[i-1].rect.right + padding

```

Und jetzt – Trommelwirbel – die eigentliche Kollisionsprüfung. Je nachdem welche Kollisionsprüfung wir eingestellt haben, wird innerhalb der `for`-Schleife die entsprechende Methode zur Kollisionsprüfung aufgerufen: `pygame.sprite.collide_circle()`, `pygame.sprite.collide_mask()` oder `pygame.sprite.collide_rect()`. Die Semantik ist eigentlich simpel. Den Methoden werden zwei `Sprite`-Objekte übergeben und sie liefern `True` falls eine Kollision vorliegt, ansonsten `False`. Dabei ist – wie oben schon erwähnt – darauf zu achten, dass die benutzte Methode auch die Infos im Sprite vorfindet, die sie braucht:

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

Quelltext 2.60: Kollisionsarten (8): `check_for_collision()` von Game

```

143     def check_for_collision(self) -> None:
144         if Settings.MODUS == "circle":
145             for s in self.all_obstacles:
146                 s.update(hit=pygame.sprite.collide_circle(self.bullet.sprite, s))
147         elif Settings.MODUS == "mask":
148             for s in self.all_obstacles:
149                 s.update(hit=pygame.sprite.collide_mask(self.bullet.sprite, s))
150         else:
151             for s in self.all_obstacles:
152                 s.update(hit=pygame.sprite.collide_rect(self.bullet.sprite, s))

```

Noch ein Hinweis: Die Kollisionsprüfung mit Rechtecken einer Liste – also kollidiert ein Sprite mit irgendeinem Sprite einer `SpriteGroup` – wird so oft gebraucht, dass es dafür eine eigene Methode gibt: `pygame.sprite.spritecollide()`. Der erste Parameter ist ein einzelnes `Sprite`-Objekt – hier unsere Feuerkugel. Der zweite Parameter ist die Liste von `Sprites`, in der nach einer Kollision gesucht werden soll. Der dritte Parameter regelt, ob die kollidierenden Objekte aus der Liste entfernt werden soll. Dies ist ganz nützlich, wenn beispielsweise das Hindernis durch Berührung verschwinden soll.

`spritecollide()`

Hinweis: Die Methode hat noch einen vierten Parameter. Diesem kann man einen Funktionszeiger auf eine andere Kollisionsprüfungsmethode mitgeben. Diese Funktion muss zwei `Sprite`-Objekte als Parameter akzeptieren. Man kann also etwas Selbsterstelltes oder eine der drei Methoden `collide_circle()`, `collide_mask()` oder `collide_rect()` verwenden. Wird hier nichts angegeben – so wie in unserem Quelltext – wird automatisch `collide_rect()` verwendet.

Quelltext 2.61: Kollisionsarten (9): Variante von `check_for_collision()` von Game

```

143     def check_for_collision(self) -> None:
144         match Settings.MODUS:
145             case "circle":
146                 func = pygame.sprite.collide_circle
147             case "mask":
148                 func = pygame.sprite.collide_mask
149             case _:
150                 func = pygame.sprite.collide_rect
151         hits = pygame.sprite.spritecollide(self.bullet.sprite, self.all_obstacles, False,
152                                           func)
153         for s in self.all_obstacles:
154             s.update(hit=s in hits)

```

Und zu guter letzt noch der Aufruf:

Quelltext 2.62: Kollisionsarten (10): Der Aufruf von Game

```

155     def main():
156         game = Game()
157         game.run()
158
159
160
161 if __name__ == '__main__':
162     main()

```

Was war neu?

Es gibt drei Standardarten die Kollision zweier Sprites zu testen. Ob sich Rechtecke schneiden, ob sich Innenkreise – oder allgemeiner Umkreise – schneiden oder ob sich Pixel des Objekts überschneiden.

Um diese Kollisionsprüfungen durchführen zu können, muss das Sprite mit entsprechenden Infos versorgt werden: `rect`, `radius` oder `mask`.

Es wurden folgende Pygame-Elemente eingeführt:

- `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>

2.10 Zeitsteuerung

In Spielen werden an vielen Stellen zeitgesteuerte Aktionen benötigt: jede halbe Sekunde fällt eine Bombe, das Schutzschild ist 10 Sekunden aktiv, nach 3 Sprüngen steht die Funktion *Sprung* 5 Minuten lang nicht zur Verfügung, bei einer Animation sollen die Teilbilder jede 1/30 Sekunde erscheinen usw..

Schauen wir uns zunächst die Bildschirmausgabe von Quelltext 2.63ff. in Abbildung 2.38 an. Die Feuerbälle werden offensichtlich in dichter Folge abgeworfen, so dass diese wie eine Kette erscheinen. Durch die horizontale Bewegung des Enemys bekommen wir eine schräge Linie; so soll es offensichtlich nicht sein.



Abbildung 2.38: Feuerball ohne Zeitsteuerung

Bevor wir die Zeitsteuerung selbst angehen, ein kurzer Blick ins Programm. Präambel und die Klasse `Settings` kommen mit nichts Neuem um die Ecke.

Quelltext 2.63: Zeitsteuerung (1), Version 1.0: Präambel und `Settings`

```

1 import os
2 from time import time
3 from typing import Any, Tuple
4
5 import pygame
6
7
8 class Settings(object):
9     WINDOW = pygame.rect.Rect((0, 0), (700, 200))
10    FPS = 60
11    DELTATIME = 1.0 / FPS
12    TITLE = "Zeitsteuerung"
13    PATH: dict[str, str] = {}
14    PATH['file'] = os.path.dirname(os.path.abspath(__file__))
15    PATH['image'] = os.path.join(PATH['file'], "images")
16
17    @staticmethod
18    def filepath(name: str) -> str:
19        return os.path.join(Settings.PATH['file'], name)
20
21    @staticmethod
22    def imagepath(name: str) -> str:
23        return os.path.join(Settings.PATH['image'], name)

```

Die Klasse `Enemy` liefert auch nichts Weltbewegendes. Mit 10 *px* Abstand pendelt der `Enemy` immer von links nach rechts bzw. umgekehrt.

Quelltext 2.64: Zeitsteuerung (2), Version 1.0: `Enemy`

```

26  class Enemy(pygame.sprite.Sprite):
27
28      def __init__(self, filename: str) -> None:
29          super().__init__()
30          self.image = pygame.image.load(Settings.imagepath(filename)).convert_alpha()
31          self.rect = pygame.Rect(self.image.get_rect())
32          self.rect.topleft = (10, 10)
33          self.direction = 1
34          self.speed = pygame.math.Vector2(150, 0)
35
36      def update(self, *args: Any, **kwargs: Any) -> None:
37          newpos = self.rect.move(self.speed * Settings.DELTATIME * self.direction)
38          if newpos.left < 10 or newpos.right >= Settings.WINDOW.right - 10:
39              self.direction *= -1
40          else:
41              self.rect = newpos

```

Auch `Bullet` ist in weiten Teilen eine Wiederholung. Interessant dürfte Zeile 57 sein. Die Methode `pygame.sprite.Sprite.kill()` ist nicht wirklich eine Selbstzerstörung. Vielmehr entfernt diese Methode das `Sprite`-Objekt aus allen `Spritegroups`. Wenn damit auch alle Referenzen verloren gehen, wird natürlich auch dieses Objekt zerstört, besteht aber noch irgendwo eine Referenz, bleibt das Objekt erhalten. In der Regel werden `Sprite`-Objekte aber in Gruppen (also in `pygame.sprite.Group`-Objekten) verwaltet und somit durch `kill()` zerstört. Sie können das in Abbildung 2.38 auf der vorherigen Seite dadurch erkennen, dass 30 *px* vor dem unteren Bildschirmrand der Feuerball verschwindet.

Quelltext 2.65: Zeitsteuerung (3), Version 1.0: `Bullet`

```

44  class Bullet(pygame.sprite.Sprite):
45
46      def __init__(self, picturefile: str, startpos: Tuple[int, int]) -> None:
47          super().__init__()
48          self.image = pygame.image.load(Settings.imagepath(picturefile)).convert_alpha()
49          self.rect = pygame.Rect(self.image.get_rect())
50          self.rect.center = startpos
51          self.direction = 1
52          self.speed = pygame.math.Vector2(0, 100)
53
54      def update(self, *args: Any, **kwargs: Any) -> None:
55          self.rect.move_ip(self.speed * Settings.DELTATIME * self.direction)
56          if self.rect.top > Settings.WINDOW.bottom - 30:
57              self.kill()                                # Selbstzerstörung

```

Im Konstruktor Der Klasse `Game` wird eine `Spritegroup` für die Feuerbälle angelegt und ein `GroupSingle`-Objekt für den `Enemy`. In `run()` erfolgt die übliche Abarbeitung der Teilaufgaben durch entsprechende Funktionsaufrufe. Ein kurzes Augenmerk möchte ich auf Zeile 80ff. lenken. Durch den Aufruf von `pygame.time.Clock.tick()` wird das Spiel getaktet – hier auf das 1/60 einer Sekunde und anschließend die *Deltatime* berechnet.

`tick()`
`deltatime`

Quelltext 2.66: Zeitsteuerung (4), Version 1.0: Konstruktor und `run()` von `Game`

```

60  class Game(object):
61
62      def __init__(self) -> None:
63          super().__init__()
64          os.environ['SDL_VIDEO_WINDOW_POS'] = "10, 50"
65          pygame.init()
66          pygame.display.set_caption(Settings.TITLE)
67          self.screen = pygame.display.set_mode(Settings.WINDOW.size)
68          self.clock = pygame.time.Clock()
69          self.enemy = pygame.sprite.GroupSingle(Enemy("alienbig1.png"))
70          self.all_bullets = pygame.sprite.Group()
71          self.running = False
72
73      def run(self) -> None:
74          time_previous = time()
75          self.running = True
76          while self.running:
77              self.watch_for_events()
78              self.update()
79              self.draw()
80              self.clock.tick(Settings.FPS)           # Taktung
81              time_current = time()
82              Settings.DELTATIME = time_current - time_previous
83              time_previous = time_current
84      pygame.quit()

```

Die Methoden `watch_for_events()` und `draw()` sind auch ohne Besonderheiten.

Quelltext 2.67: Zeitsteuerung (5), Version 1.0: `watch_for_events()` und `draw()` von `Game`

```

86  def watch_for_events(self) -> None:
87      for event in pygame.event.get():
88          if event.type == pygame.QUIT:
89              self.running = False
90          elif event.type == pygame.KEYDOWN:
91              if event.key == pygame.K_ESCAPE:
92                  self.running = False
93
94  def draw(self) -> None:
95      self.screen.fill((200, 200, 200))
96      self.all_bullets.draw(self.screen)
97      self.enemy.draw(self.screen)
98      pygame.display.flip()

```

Die Methode `update()` ist nur bzgl. Zeile 101 erwähnenswert, da dort ein neuer Feuerball erzeugt/abgeworfen wird, indem die Methode `new_bullet()` aufgerufen wird. Die Startposition ergibt sich aus der aktuellen Position des `Enemy`s. Das horizontale Zentrum von Feuerball und `Enemy` soll gleich sein. Das vertikale Zentrum ist etwas nach unten verschoben; sieht besser aus.

Quelltext 2.68: Zeitsteuerung (6), Version 1.0: `update()` und `new_bullet()` von `Game`

```

100  def update(self) -> None:
101      self.new_bullet()                                     # Feuerballabwurf
102      self.all_bullets.update()
103      self.enemy.update()
104
105  def new_bullet(self) -> None:
106      self.all_bullets.add(Bullet("shoot.png", self.enemy.sprite.rect.move(0, 20).center))

```

Zurück zum eigentlichen Problem. Wir haben oben festgestellt, dass durch `Settings.FPS` und dem Aufruf von `tick()` in Zeile 80 die Anwendung auf das 1/60 einer Sekunde getaktet ist. Mit anderen Worten: Derzeit werden maximal 60 Feuerbälle pro Sekunde erzeugt, was Schwachsinn ist. Eine naive Idee wäre nun, die Taktung zu verringern. Will ich also nur jede halbe Sekunde einen Feuerball erzeugen, müsste die Taktung auf 2 gesetzt werden. Probieren Sie es aus!

Das Ergebnis ist ernüchternd. Es wird ja damit das ganze Spiel verlangsamt. Das ist nicht Sinn der Sache. Eine nächste und gar nicht so schlechte Idee wäre die Einführung einer Zählers. Der Gedanke dabei ist, wenn die Taktung 1/60 ist, zähle ich bis 30 und werfe erst dann einen Feuerball ab.

Im ersten Schritt werden in `Game` dazu zwei Attribute angelegt (Zeile 71 und Zeile 72).

Quelltext 2.69: Zeitsteuerung (7), Version 1.1: Konstruktor von `Game`

```

70  self.all_bullets = pygame.sprite.Group()
71  self.time_counter = 0                                # Zähler
72  self.time_range = 30                                # Obergrenze
73  self.running = False

```

In der Methode `new_bullet()` werden diese beiden Werte nun dazu genutzt, um den zeitlichen Abstand zwischen zwei Abwürfen zu steuern. Zunächst wird bei jedem Aufruf der Zähler um 1 erhöht. Da die Methode bei jedem Schleifendurchlauf der Hauptprogrammschleife aufgerufen wird und jeder Durchlauf getaktet ist, wird dadurch die Anzahl der Takte mitgezählt.

Überschreitet der Zähler seine Obergrenze (in unserem Beispiel die 30), ist eine halbe Sekunde seit dem letzten Abwurf vergangen, und ein neuer Abwurf wird durchgeführt.

Zum Schluss muss der Zähler wieder auf 0 gesetzt werden, da wir ja wieder die nächsten 30 Takte warten müssen. Das Ergebnis sehen wir in Abbildung 2.39 auf der nächsten Seite: Es sind nur noch zwei Feuerbälle sichtbar.

Quelltext 2.70: Zeitsteuerung (8), Version 1.1: `new_bullet()` von `Game`

```

107 def new_bullet(self) -> None:
108     self.time_counter += 1                            # Erhöhe pro Takt um 1
109     if self.time_counter >= self.time_range:        # Wenn Obergrenze erreicht
110         self.all_bullets.add(Bullet("shoot.png", self.enemy.sprite.rect.move(0,
111                                         20).center))                                # Setze Zähler wieder zurück
         self.time_counter = 0

```

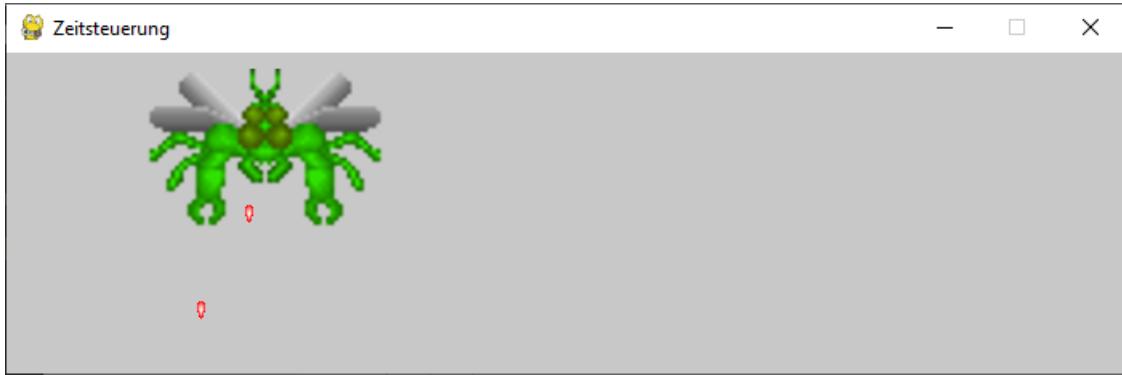


Abbildung 2.39: Feuerball mit Zeitsteuerung

Die Vorteile dieses Verfahrens sind: Es ist einfach zu implementieren, und die Geschwindigkeit des Spiels selbst wird nicht beeinflusst.

Es gibt aber einen entscheidenden Nachteil: Das ganze funktioniert nur, wenn die Taktung sich nicht ändert bzw. immer wie vorgesehen ist. Das ist aber nicht wirklich der Fall. Wir erinnern uns: Der Aufruf von `tick()` sorgt dafür, dass höchstens 60 mal pro Sekunde die Schleife durchwandert wird. Bei hoher Auslastung kann dies auch weniger sein. Auch wird die Anzahl der *frames per second* bei vielen Spielen dynamisch ermittelt, damit auf die unterschiedliche Leistungsfähigkeit der Hardware reagiert werden kann. Es ist also keine wirklich stabile Lösung, die Zeitsteuerung an die Taktung zu koppeln.

Besser ist es, die Zeitsteuerung an einen echten Zeitmesser zu koppeln. Hilfreich ist dabei die Methode `pygame.time.get_ticks()`. Diese Methode liefert mir die Zeitspanne seit Start des Spiels in [Millisekunden \(ms\)](#) und das ist unabhängig von der Arbeitsgeschwindigkeit der Hardware oder meines Programmes.

`get_ticks()`

Nun kann man den Quelltext umbauen. Zuerst wird in Zeile 71 die aktuelle Anzahl der *ms* seit Programmstart gemessen und in Zeile 72 wird festgehalten, wie viele *ms* ein Zeitintervall dauern soll; wir wollen alle halbe Sekunde einen Feuerball abwerfen, also 500.

Quelltext 2.71: Zeitsteuerung (9), Version 1.2: Konstruktor von Game

```

70     self.all_bullets = pygame.sprite.Group()
71     self.time_stamp = pygame.time.get_ticks()          # Zeitpunkt festhalten
72     self.time_duration = 500                          # Intervalldauer
73     self.running = False

```

Danach wird in `new_bullet()` abgeprüft, ob das Intervallende erreicht wurde. In Zeile 108 wird zuerst wieder mit `pygame.time.get_ticks()` die aktuelle Zeit gemessen. Ist diese größer als der alte Intervallbeginn plus Intervalldauer – was ja das gleiche wie das Intervallende ist –, so müssen 500 *ms* vergangen sein, und ein neuer Feuerball wird abgeworfen. Nun muss nur noch der neue Intervallstart ermittelt werden, und das erfolgt in Zeile 108.

Quelltext 2.72: Zeitsteuerung (10), Version 1.2: `new_bullet()` von Game

```

107     def new_bullet(self) -> None:
108         if pygame.time.get_ticks() >= self.time_stamp + self.time_duration: #
109             self.all_bullets.add(Bullet("shoot.png", self.enemy.sprite.rect.move(0,
110                                         20).center))
110             self.time_stamp = pygame.time.get_ticks()      # Intervallstart

```

Timer

Da wir diese Logik mehrfach brauchen, habe ich das ganze in der Klasse `Timer` gekapselt. Das Herzstück sind wieder die beiden Attribute, die sich die Intervalldauer (`duration`) und das Intervallende (`next`) merken. Anders als bisher wird sich also nicht der Intervallstart gemerkt, sondern das Intervallende – was ein wenig Rechenzeit spart. Interessant ist der optionale Übergabeparameter `with_start`. Über diesen kann ich steuern, ob schon beim ersten Durchlauf bis zum Intervallende gewartet werden soll, oder ob beim aller ersten Aufruf von `is_next_stop_reached()` schon `True` zurückgeliefert werden soll. Was würde das bei unserem Beispiel bedeuten? Würde `width_start` den Wert `True` haben, würde der erste Feuerball sofort beim ersten Schleifendurchlauf abgeworfen werden. Wäre der Wert `False`, würde der erste Feuerball erst nach 500 ms abgeworfen werden.

In `is_next_stop_reached()` wird das Erreichen des Intervallendes überprüft und ggf. das neue Intervallende festgelegt. Die Methode liefert ein `True`, wenn das Intervallende erreicht/überschritten wurde und ansonsten `False`.

Quelltext 2.73: Zeitsteuerung (11), Version 1.3: `Timer`

```

26 class Timer(object):
27
28     def __init__(self, duration: int, with_start: bool = True) -> None:
29         self.duration = duration
30         if with_start:
31             self.next = pygame.time.get_ticks()
32         else:
33             self.next = pygame.time.get_ticks() + self.duration
34
35     def is_next_stop_reached(self) -> bool:
36         if pygame.time.get_ticks() > self.next:
37             self.next = pygame.time.get_ticks() + self.duration
38         return True
39     return False

```

Wie wird dieser Timer nun verwendet? Zunächst wird im Konstruktor ein entsprechendes Objekt erzeugt (Zeile 87); die beiden Variablen von eben werden nicht mehr gebraucht.

Quelltext 2.74: Zeitsteuerung (12), Version 1.3: `Timer`-Objekt erzeugen

```

86     self.all_bullets = pygame.sprite.Group()
87     self.bullet_timer = Timer(500)                      # Timer ohne Verzögerung
88     self.running = False

```

Die Methode `new_bullet()` hat sich nun vereinfacht, da sie sich nicht mehr um die interne Timer-Logik kümmern muss. Es wird lediglich in Zeile 123 abgefragt, ob das Intervallende erreicht wurde und fertig!

Quelltext 2.75: Zeitsteuerung (13), Version 1.3: Timer-Objekt verwenden

```
122     def new_bullet(self) -> None:
123         if self.bullet_timer.is_next_stop_reached():      # Wenn Intervallgrenze erreicht
124             self.all_bullets.add(Bullet("shoot.png", self.enemy.sprite.rect.move(0,
20).center))
```

Hinweis: Eine Zeitsteuerung über Ereignisse wird in Kapitel [2.15.3](#) auf Seite [133](#) vorgestellt.

Was war neu?

Zeitliche Ereignisse oder Zeitspannen sollten von der Framerate unabhängig gemacht werden und sich an der tatsächlich verstrichenen Zeit orientieren. Da es sich um eine oft verwendete Logik handelt, wird diese in einer Klasse gekapselt.

Es wurden folgende Pygame-Elemente eingeführt:

- `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>

2.11 Animation

Eine Animation ist eigentlich eine Art *Filmchen* innerhalb eines Spiels. Beispiele für sinnvolle Animationen sind Bewegungen, Explosioen, Pulsieren, Übergänge von Aussehen usw.. Ich möchte hier zwei Beispiele vorstellen: ein kleine Bewegung und eine Explosion.

2.11.1 Die laufende Katze

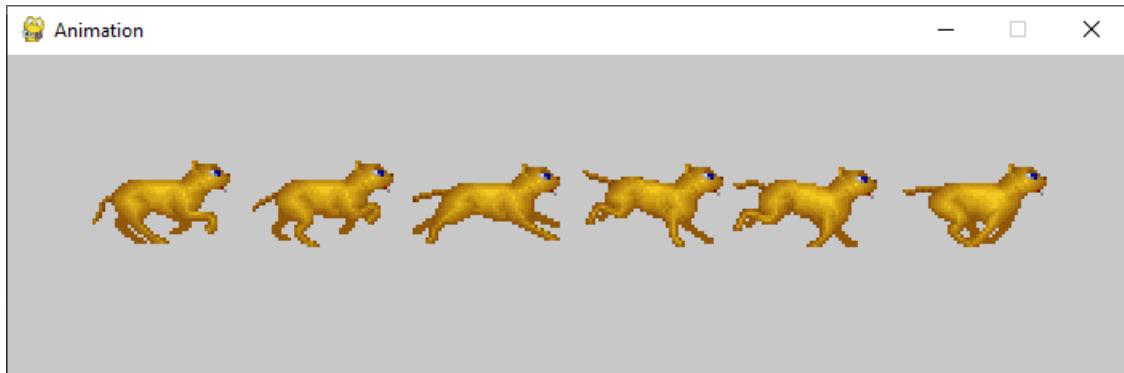


Abbildung 2.40: Animation einer Katze: Einzelsprites

Die Einzelbilder des Bewegungsbeispiels können Sie in Abbildung 2.40 sehen. Werden diese Einzelsprites in einer gewissen Geschwindigkeit hintereinander ausgegeben, so erscheinen sie wie eine flüssige Bewegung. Dabei gilt: Je mehr Einzelbilder, desto flüssiger die Bewegung.

Der Quelltext 2.76 unterscheidet sich nur um ein Feature zum letzten Kapitel. Die **Timer**-Klasse wurde um die Methode `change_duration()` erweitert. Diese Methode ermöglicht es, zur Laufzeit die Dauer des Zeitintervalls zu verändern, wobei die untere Grenze bei 0 ms festgelegt wird. Wir werden dieses Feature gleich dazu verwenden, die Animationsgeschwindigkeit manuell einzustellen.

Quelltext 2.76: Animation einer Katze (1), Version 1.0: Präambel, Timer und Settings

```

1 import os
2 from time import time
3 from typing import Any, Tuple
4
5 import pygame
6 from pygame.constants import K_ESCAPE, K_MINUS, K_PLUS, KEYDOWN, QUIT
7
8
9 class Settings():
10     WINDOW = pygame.rect.Rect((0, 0), (300, 200))
11     FPS = 60
12     DELTATIME = 1.0 / FPS
13     TITLE = "Animation"
14     PATH: dict[str, str] = {}

```

```

15     PATH['file'] = os.path.dirname(os.path.abspath(__file__))
16     PATH['image'] = os.path.join(PATH['file'], "images")
17
18     @staticmethod
19     def filepath(name: str) -> str:
20         return os.path.join(Settings.PATH['file'], name)
21
22     @staticmethod
23     def imagepath(name: str) -> str:
24         return os.path.join(Settings.PATH['image'], name)
25
26
27 class Timer():
28
29     def __init__(self, duration: int, with_start: bool = True):
30         self.duration = duration
31         if with_start:
32             self.next = pygame.time.get_ticks()
33         else:
34             self.next = pygame.time.get_ticks() + self.duration
35
36     def is_next_stop_reached(self) -> bool:
37         if pygame.time.get_ticks() > self.next:
38             self.next = pygame.time.get_ticks() + self.duration
39             return True
40         return False
41
42     def change_duration(self, delta: int = 10):
43         self.duration += delta
44         if self.duration < 0:
45             self.duration = 0

```

Wenn wir etwas animieren wollen, so benötigt diese Animation nicht nur ein Sprite zur Darstellung, sondern mehrere. Ich habe deshalb neben dem Attribut `image` ein weiteres: das Array `images`. In dieses lade ich nun mit Hilfe der `for`-Schleife ab Zeile 53 alle Bitmaps der Animation. Ich brauche nun ein Attribut, das sich merkt, welches der 6 Sprites nun eigentlich angezeigt werden soll: `imageindex`; Wenn die Bilder in der Reihenfolge in das Array `images` abgelegt werden, in welcher sie auch ausgegeben werden sollen, so muss `imageindex` nur noch hochgezählt werden. Auch brauchen wir ein `Timer`-Objekt, damit die Animation nicht absurd schnell abläuft – wir starten hier mit 100 ms.

In der Methode `update()` wird nun abhängig vom `Timer`-Objekt das Attribut `imageindex` immer um 1 erhöht und dieses Bitmap dann dem Attribut `image` zugewiesen, damit die schon bekannten `Sprite`-Features genutzt werden können. Die Methode `change_animation_time()` reicht seinen Übergabeparameter einfach nur an das `Timer`-Objekt weiter. Damit sind eigentlich alle vorbereitenden Aktivitäten abgeschlossen.

Quelltext 2.77: Animation einer Katze (2), Version 1.0: Cat

```

48 class Cat(pygame.sprite.Sprite):
49
50     def __init__(self) -> None:
51         super().__init__()
52         self.images: list[pygame.surface.Surface] = []
53         for i in range(6):
54             bitmap = pygame.image.load(Settings.imagepath(f"cat{i}.bmp")).convert()
55             bitmap.set_colorkey("black")
56             self.images.append(bitmap)

```

```

57     self.imageindex = 0
58     self.image: pygame.surface.Surface = self.images[self.imageindex]
59     self.rect: pygame.rect.Rect = self.image.get_rect()
60     self.rect.center = Settings.WINDOW.center
61     self.animation_time = Timer(100)
62
63     def update(self, *args: Any, **kwargs: Any) -> None:
64         if "animation_delta" in kwargs.keys():
65             self.change_animation_time(kwargs["animation_delta"])
66         if self.animation_time.is_next_stop_reached():
67             self.imageindex += 1
68             if self.imageindex >= len(self.images):
69                 self.imageindex = 0
70             self.image = self.images[self.imageindex]
71             # implement game logic here
72
73     def change_animation_time(self, delta: int) -> None:
74         self.animation_time.change_duration(delta)

```

Die Klasse `CatAnimation` ist nur die übliche Kapselung des Hauptprogramms. In Zeile 87 wird das `Cat`-Objekt erzeugt und in ein `GroupSingle` gestopft.

Quelltext 2.78: Animation einer Katze (3), Version 1.0: Konstruktor und `run()`

```

77 class CatAnimation():
78
79     def __init__(self) -> None:
80         super().__init__()
81         os.environ['SDL_VIDEO_WINDOW_POS'] = "10,50"
82         pygame.init()
83         self.screen = pygame.display.set_mode(Settings.WINDOW.size)
84         pygame.display.set_caption(Settings.TITLE)
85         self.clock = pygame.time.Clock()
86         self.font = pygame.font.Font(pygame.font.get_default_font(), 12)
87         self.cat = pygame.sprite.GroupSingle(Cat())           # Meine Katze
88         self.running = False
89
90     def run(self) -> None:
91         time_previous = time()
92         self.running = True
93         while self.running:
94             self.watch_for_events()
95             self.update()
96             self.draw()
97             self.clock.tick(Settings.FPS)
98             time_current = time()
99             Settings.DELTATIME = time_current - time_previous
100            time_previous = time_current
101        pygame.quit()

```

In `watch_for_events()` ist nur erwähnenswert, dass die `+`-Taste und die `--`-Taste für die Manipulation der Animationsgeschwindigkeit verwendet werden. Um die Animationsgeschwindigkeit zu erhöhen, muss das Zeitintervall des `Timer`-Objekts verkleinert werden, daher `-10`. Um die Animationsgeschwindigkeit zu verlangsamen, muss das Zeitintervall des `Timer`-Objekts verlängert werden, daher `+10`.

Quelltext 2.79: Animation einer Katze (4), Version 1.0: `watch_for_events()`

```

103     def watch_for_events(self) -> None:
104         for event in pygame.event.get():
105             if event.type == QUIT:
106                 self.running = False
107             elif event.type == KEYDOWN:
108                 if event.key == K_ESCAPE:
109                     self.running = False
110                 elif event.key == K_PLUS:
111                     self.cat.sprite.update(animation_delta=-10)
112                 elif event.key == K_MINUS:
113                     self.cat.sprite.update(animation_delta=10)

```

Der restliche Quelltext (Quelltext 2.80) sollte selbsterklärend sein. Wenn Sie das Programm nun starten, ist eine animierte Katzenbewegung zu sehen. Probieren Sie doch mal aus, die Animationsgeschwindigkeit zu verändern.

Quelltext 2.80: Animation einer Katze (5), Version 1.0: `update()` und `draw()`

```

115     def update(self) -> None:
116         self.cat.update()
117
118     def draw(self) -> None:
119         self.screen.fill("gray")
120         self.cat.draw(self.screen)
121         text_image = self.font.render(f"animation_time: {self.cat.animation_time.duration}", True, "white")
122         text_rect = text_image.get_rect()
123         text_rect.centerx = Settings.WINDOW.centerx
124         text_rect.bottom = Settings.WINDOW.bottom - 50
125         self.screen.blit(text_image, text_rect)
126         pygame.display.flip()

```

Wie bei der Zeitsteuerung stört mich, dass die Animationslogik über die Klasse `Cat` verteilt ist, was meiner Ansicht nach ein Verstoß gegen das SRP ist. Bauen wir doch einfach eine Animationsklasse (siehe Quelltext 2.81 auf der nächsten Seite).

Schauen wir uns die Übergabeparameter des Konstruktors an:

- **namelist**: Eine Liste von Dateinamen ohne Pfadangaben. Diese werden eigenständig anhand der Einträge in `Settings` ermittelt. Die Reihenfolge der Dateinamen muss der Animationsreihenfolge entsprechen.
- **endless**: Über dieses Flag wird gesteuert, ob die Animation sich immer wiederholt. `True` bedeutet, dass nach dem letztes Sprite wieder mit dem ersten begonnen wird. `False` lässt das letzte Sprite stehen.
- **animationtime**: Abstand der Einzelsprites in *ms*.
- **colorkey**: Mit diesem Parameter wird abgefangen, dass Sprites ggf. keine Transparenz besitzen und daher eine Angabe über Transparenzfarbe brauchen (siehe Seite 22). Wird keine Angabe gemacht, bleibt die Transparenz des geladenen Sprites erhalten. Wird eine Farbangabe gemacht, wird diese mit `set_colorkey()` in Zeile 38 verwendet.

In der Methode `next()` wird der nächste `imageindex` berechnet und das dazu passende Sprite zurückgeliefert. Dazu wird das interne `Timer`-Objekt verwendet, damit die Sprites in einem gewissen zeitlichen Abstand erscheinen. Das Attribut `imageindex` wird dabei um 1 erhöht und dahingehend überprüft, ob damit das Ende des Spritearrays erreicht wurde. Wurde die Animation auf *endlos* gesetzt, beginnt er wieder mit dem `imageindex` bei 0; falls nicht, wird immer das letzte Bild des Arrays ausgegeben.

Frage ins Plenum: Warum wurde im Konstruktor `imageindex` auf `-1` gesetzt?

Ein Feature, was man immer wieder mal braucht, wurde in der Methode `is-ended()` implementiert. Oft braucht derjenige, der die Animation aufgerufen hat, die Information darüber, ob die Animation beendet ist. Wir werden das später noch in Gebrauch sehen.

Quelltext 2.81: Animation (6), Version 1.1: Animation

```

27  class Animation():
28
29      def __init__(self, namelist: list[str], endless: bool, animationtime: int, colorkey:
30          tuple[int, int, int] | None = None) -> None:
31          self.images: list[pygame.surface.Surface] = []
32          self.endless = endless
33          self.timer = Timer(animationtime)
34          for filename in namelist:
35              if colorkey == None:
36                  bitmap = pygame.image.load(Settings.imagepath(filename)).convert_alpha()
37              else:
38                  bitmap = pygame.image.load(Settings.imagepath(filename)).convert()
39                  bitmap.set_colorkey(colorkey)           # Transparenz herstellen
40              self.images.append(bitmap)
41          self.imageindex = -1
42
43      def next(self) -> pygame.surface.Surface:
44          if self.timer.is_next_stop_reached():
45              self.imageindex += 1
46              if self.imageindex >= len(self.images):
47                  if self.endless:
48                      self.imageindex = 0
49                  else:
50                      self.imageindex = len(self.images) - 1
51
52      def is-ended(self) -> bool:
53          if self.endless:
54              return False
55          return self.imageindex >= len(self.images) - 1

```

Die Klasse `Cat` hat sich damit vereinfacht und kann sich wieder mehr auf ihre – hier natürlich noch nicht vorhandene – Spiellogik konzentrieren. Das Erzeugen des `Animation`-Objekts erfolgt hier in Zeile 83. Die Dateinamen lassen sich schön einfach generieren, da sie durchnummiert wurden. Die Katze soll endlos laufen und dabei 100 ms zeitlichen Abstand zwischen den Sprites haben. In `update()` wird dann einfach die Methode `next()` aufgerufen.

Quelltext 2.82: Animation einer Katze (7), Version 1.1: Cat

```

79  class Cat(pygame.sprite.Sprite):

```

```

80
81     def __init__(self) -> None:
82         super().__init__()
83         self.animation = Animation([f"cat{i}.bmp" for i in range(6)], True, 100, (0, 0, 0))
84         #
85         self.image: pygame.surface.Surface = self.animation.next()
86         self.rect: pygame.rect.Rect = self.image.get_rect()
87         self.rect.center = Settings.WINDOW.center
88
89     def update(self, *args: Any, **kwargs: Any) -> None:
90         if "animation_delta" in kwargs.keys():
91             self.change_animation_time(kwargs["animation_delta"])
92         self.image = self.animation.next()
93         # implement game logic here
94
95     def change_animation_time(self, delta: int) -> None:
96         self.animation.timer.change_duration(delta)

```

2.11.2 Der explodierende Felsen

Mein zweites Beispiel lässt an zufälliger Position in zufälligem zeitlichen Abstand Felsen (Meteoriten) erscheinen. Ihnen wird – ebenfalls zufällig – eine gewisse Lebensdauer mitgegeben. Danach explodieren sie. Diese Explosion ist animiert.

Schauen wir uns zuerst die Klasse `Rock` an. In Zeile 84 wird eine Zufallszahl ermittelt, die ich in der darauffolgenden Zeile brauche, um einen von vier möglichen Felsenbitmaps zu laden. Danach werden die Koordinaten des Mittelpunkts des Felsens per Zufallszahlen-generator geraten, wobei ein gewisser Abstand zu den Rändern gewahrt wird. In Zeile 89 wird das `Animation`-Objekt erzeugt. Dabei werden die Dateinamen der Animationsbitmaps wieder in der Reihenfolge der Animation eingelesen. Die Bitmaps können Sie in Abbildung 2.41 auf der nächsten Seite sehen.

Da die Animation sich nicht wiederholen soll, wird hier der entsprechende Übergabeparameter mit `False` angegeben. Nach der Explosion soll der Felsen ja verschwinden. Der Abstand zwischen den Einzelbildern wird auf 50 ms festgelegt. In Zeile 90 wird die Lebensdauer des Felsens wiederum per Zufall bestimmt und ein entsprechendes `Timer`-Objekt erzeugt – wie Sie sehen, kann man die Dinger recht oft gebrauchen. Das Flag `bumm` ist ein Marker darüber, ob ich gerade am explodieren bin².

Die Methode `update()` ist nun recht spannend geworden. Zuerst wird über das `Timer`-Objekt abgefragt, ob das Lebensende erreicht wurde. Wenn nicht, passiert hier garnichts, aber man könnte eine Bewegung oder irgendetwas anderes Sinnvolles im `else`-Zweig programmieren. Falls das Lebensende erreicht wurde, wird das entsprechende Flag gesetzt. Abhängig davon wird nun die Animation gestartet.

Was hat es mit den drei Zeilen ab Zeile 98 auf sich? Sie dienen rein optischen Zwecken. Die Abmaße der Explosionssprites sind nicht immer gleich und werden durch das `rect`-Objekt immer auf die linke, obere Koordinate ausgerichtet, was zu einem Ruckeln führen

² Was für eine Grammatik! Aber ich kann mich rausreden: Im westfälischen Dialekt gibt es ähnlich wie im Englischen eine Verlaufsform :-)

würde. So merke ich mir das alte Zentrum, berechne das neue Rechteck des nächsten Animationsprites und setze sein Zentrum auf die alte Position. So bleibt die Animation schön auf die alte Mitte des Felsen ausgerichtet.

Zum Schluss wird noch festgestellt, ob die Animation fertig ist. Wenn ja, dann brauche ich das Sprite nicht mehr und es kann aus der Spritegroup mit `kill()` entfernt werden.

`kill()`

Quelltext 2.83: Animation einer Explosion (1): Rock

```

80  class Rock(pygame.sprite.Sprite):
81
82      def __init__(self):
83          super().__init__()
84          rocknb = random.randint(6, 9)                                # Felsennummer
85          self.image =
86              pygame.image.load(Settings.imagepath(f"felsen{rocknb}.png")).convert_alpha()
87          self.rect = self.image.get_rect()
88          self.rect.centerx = random.randint(self.rect.width,
89                                              Settings.WINDOW.width-self.rect.width)
89          self.rect.centery = random.randint(self.rect.height,
90                                              Settings.WINDOW.height-self.rect.height)
90          self.anim = Animation([f"explosion0{i}.png" for i in range(1, 5)], False, 50)  #
91          self.timer_lifetime = Timer(random.randint(100, 2000), False)      # Lebenszeit
92          self.bumm = False
93
94      def update(self, *args: Any, **kwargs: Any) -> None:
95          if self.timer_lifetime.is_next_stop_reached():
96              self.bumm = True
97          if self.bumm:
98              self.image = self.anim.next()                                # Zentrum
99              c = self.rect.center
100             self.rect = self.image.get_rect()
101             self.rect.center = c
102         if self.anim.isEnded():
103             self.kill()

```



Abbildung 2.41: Animation einer Explosion: Einzelsprites

Die Klasse `ExplosionAnimation` sollte keine Schwierigkeit mehr für Sie sein. Es gibt nur wenige Stellen, die ich kurz ansprechen möchte. In Zeile 115 wird ein `Timer`-Objekt angelegt, welches zwei Felsen pro Sekunde erstellen soll und in Zeile 140 wird dieser abgefragt.

Quelltext 2.84: Animation einer Explosion (2): `ExplosionAnimation`

```

105  class ExplosionAnimation(object):
106
107      def __init__(self) -> None:
108          super().__init__()
109          os.environ['SDL_VIDEO_WINDOW_POS'] = "10, 50"
110          pygame.init()
111          self.screen = pygame.display.set_mode(Settings.WINDOW.size)
112          pygame.display.set_caption(Settings.TITLE)
113          self.clock = pygame.time.Clock()

```

```
114     self.all_rocks = pygame.sprite.Group()
115     self.timer_newrock = Timer(500)                                     # Timer
116     self.running = False
117
118     def run(self) -> None:
119         time_previous = time()
120         self.running = True
121         while self.running:
122             self.watch_for_events()
123             self.update()
124             self.draw()
125             self.clock.tick(Settings.FPS)
126             time_current = time()
127             Settings.DELTATIME = time_current - time_previous
128             time_previous = time_current
129             pygame.quit()
130
131     def watch_for_events(self) -> None:
132         for event in pygame.event.get():
133             if event.type == QUIT:
134                 self.running = False
135             elif event.type == KEYDOWN:
136                 if event.key == K_ESCAPE:
137                     self.running = False
138
139     def update(self) -> None:
140         if self.timer_newrock.is_next_stop_reached():                      # 500ms?
141             self.all_rocks.add(Rock())
142         self.all_rocks.update()
143
144     def draw(self) -> None:
145         self.screen.fill("black")
146         self.all_rocks.draw(self.screen)
147         pygame.display.flip()
```

Hinweis: Es gibt auch den Quelltext `animation03.py`. In dieser Variante bewegen sich die Felsen und explodieren, falls sie aufeinander treffen. Schauen Sie mal rein!

Was war neu?

Ups! Hier wurde überhaupt kein neues Pygame-Element vorgestellt. Alles wurde mit bereits bekannten Hilfsmitteln umgesetzt.

Die Animation besteht aus einer Abfolge von Einzelbildern, die in gewissen zeitlichen Abstand ausgegeben werden. Dabei wird zwischen einer endlosen Animation wie bei der Katze und einer endlichen wie bei der Explosion unterschieden.

2.12 Maus

Wie wohl viele Spiele durch Tastatur oder Controller gesteuert werden, wird auch oft die Maus verwendet. In diesem Skript werden die elementaren Mausaktionen wie *Klick* oder *Positionsabfrage* behandelt. Unser Beispiel bildet folgende Funktionalitäten ab:

- In der Mitte erscheint eine kleine transparente Blase.
- Bewegt sich die Maus innerhalb eines inneren Rechtecks, fungiert die Blase als Mauszeiger.
- Verlässt die Maus das innere Rechteck, erscheint der übliche Systemmauszeiger.
- Ein Linksklick lässt die Blase um 90° nach links rotieren.
- Ein Rechtsklick lässt die Blase um 90° nach rechts rotieren.
- Über das Mausrad wird die Größe der Blase skaliert.
- Ein Klick mit dem Mausrad beendet die Anwendung.

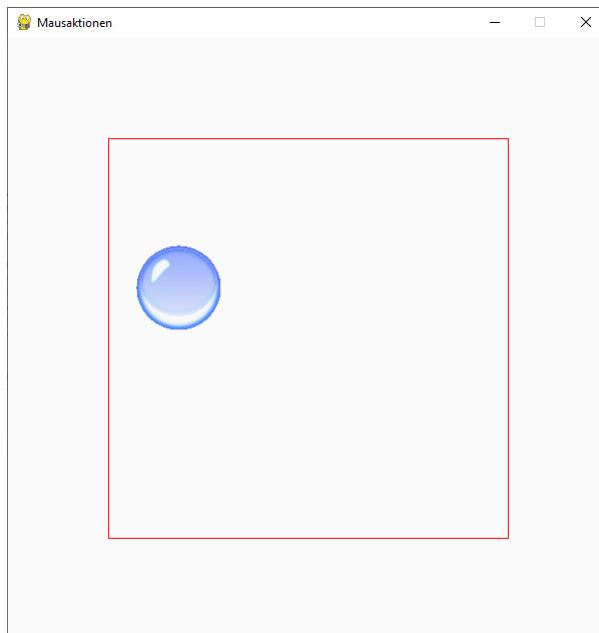


Abbildung 2.42: Mausaktionen

Die eigentliche Musik spielt in der Hauptklasse `Game`, da dort die Mausaktionen abgefragt werden. Anstelle einer Klasse `Settings`, habe ich die Einstellungen hier als statische Variablen und Methoden der Klasse `Game` implementiert – das geht auch. Im Konstruktor werden die üblichen Verdächtigen aufgerufen und in Zeile 69 wird das `Ball`-Objekt erzeugt.

Quelltext 2.85: Mausaktionen: Statics und Konstruktor von Game

```

55     WINDOW = pygame.rect.Rect((0, 0), (600, 600))
56     PATH: Dict[str, str] = {}
57     PATH["file"] = os.path.dirname(os.path.abspath(__file__))
58     PATH["image"] = os.path.join(PATH["file"], "images")
59     INNER_RECT = pygame.rect.Rect(100, 100, WINDOW.width - 200, WINDOW.height - 200)
60     FPS = 60
61     DELTATIME = 1.0 / FPS
62
63     def __init__(self) -> None:
64         os.environ["SDL_VIDEO_WINDOW_POS"] = "650, 70"
65         pygame.init()
66         self._clock = pygame.time.Clock()
67         self._screen = pygame.display.set_mode(Game.WINDOW.size)
68         pygame.display.set_caption("Mausaktionen")
69         self._ball = Ball() # Ball-Objekt
70         self._running = True

```

Auch die Methode `run()` birgt keine Überraschungen.

Quelltext 2.86: Mausaktionen: Game.run()

```

73     time_previous = time()
74     while self._running:
75         self.watch_for_events()
76         self.update()
77         self.draw()
78         self._clock.tick(Game.FPS)
79         time_current = time()
80         Game.DELTATIME = time_current - time_previous
81         time_previous = time_current
82     pygame.quit()

```

In `watch_for_events()` kommen uns die ersten interessanten Stellen unter. Wie bei den Tasten ein KEYUP und ein KEYDOWN das Drücken und Loslassen markieren, gibt es auch Entsprechungen bei der Maus: `MOUSEBUTTONDOWN` und `MOUSEBUTTONUP`. In Zeile 91 wird der `event.type` abgefragt und anschließend wird ermittelt, welche Maustaste denn gedrückt wurde.

Dazu liefern mir diese beiden Mausevents zwei Attribute: `event.button` und `event.pos`. In Tabelle 2.7 auf Seite 102 sind die Zahlenkodes von `event.button` abgebildet. Erstaunlicherweise gibt es hier keine vordefinierten Konstanten wie bei der Tastatur. Nach der Abfrage werden die entsprechenden Nachrichten an das Ball-Objekt versendet.

Wird also die linke Maustaste gedrückt (Zeile 92), wird an den Ball die Nachricht gesendet, sich um 90° nach links zu drehen und bei der rechten um 90° nach rechts (daher -90, siehe Zeile 96). Das Mausrad wird ebenfalls wie ein Mausbutton verarbeitet. Je nach Drehrichtung wird dabei ein anderer Zahlenkode zurückgeliefert (siehe Zeile 98 und Zeile 100). Wird das Mausrad gedrückt – also geklickt – soll ja das Spiel beendet werden. In Zeile 94 wird dies abgefragt und umgesetzt.

Mit `event.pos` könnte man jetzt sofort die Mausposition abfragen – was wir hier nicht tun.

MOUSE-
BUTTON-
DOWN
MOUSE-
BUTTONUP

event.button

event.pos

Quelltext 2.87: Mausaktionen: Game.watch_for_events()

```

85     for event in pygame.event.get():
86         if event.type == QUIT:
87             self._running = False
88         elif event.type == KEYDOWN:
89             if event.key == K_ESCAPE:
90                 self._running = False
91             elif event.type == MOUSEBUTTONDOWN: # Maustaste gedrückt
92                 if event.button == 1: # left
93                     self._ball.update(rotate=90)
94                 elif event.button == 2: # middle
95                     self._running = False
96                 elif event.button == 3: # right
97                     self._ball.update(rotate=-90)
98                 elif event.button == 4: # scroll up
99                     self._ball.update(scale=2)
100                elif event.button == 5: # scroll down
101                    self._ball.update(scale=-2)

```

Eine Anforderung war, dass der Systemmauszeiger nur außerhalb des inneren Rechtecks sichtbar ist. Innerhalb des Rechtecks soll ja der Ball als Mauszeiger herhalten. In Zeile 107 wird dies durch die Methode `pygame.mouse.set_visible()` erreicht. Diese steuert, ob der Systemmauszeiger – welcher Ausprägung auch immer – angezeigt werden soll oder nicht.

Als Entscheider dient dabei, ob die aktuelle Mausposition innerhalb des inneren Rechtecks liegt. Die Methode `pygame.mouse.get_pos()` liefert mir dazu die aktuelle Mausposition. Diese wird nun einfach in eine schon vorhandene Kollisionsprüfung gesteckt: `pygame.Rect.collidepoint()`. Ist die Mausposition innerhalb des Rechtecks, liefert diese den Wert `True`, ansonsten `False`; daher muss der Wahrheitswert noch mit `not` negiert werden.

Quelltext 2.88: Mausaktionen: `Game.update()` und `Game.draw()`

```

104     self._ball.update(center=pygame.mouse.get_pos())
105     pygame.mouse.set_visible(
106         not Game.INNER_RECT.collidepoint(pygame.mouse.get_pos()))
107     ) # Unsichtbar?
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        pygame.display.flip()

```

Verbleibt noch die Klasse `Ball`. Diese enthält zwar keine direkten Mausaktionen mehr, aber die Methode `update()` sieht nun ganz anders als bei den vorherigen Beispielen aus. In früheren Beispielen wurden Methoden wie `rotate()` oder `resize()` direkt aus `watch_for_events()` oder vergleichbaren Methoden von `Game` aufgerufen. Das ist auch soweit in Ordnung, aber wenn man diese Kindklassen von `pygame.sprite.Sprite` einer `pygame.sprite.Group` oder `pygame.sprite.GroupSingle` hinzufügt hat, kriegt man ein Problem. Diese Klassen erwarten nur `Sprite`-Objekt als Elemente. Deshalb kann man eigentlich im Sinne der objektorientierten Programmierung nur Methoden und Attribute verwenden, die der Elternklasse `pygame.sprite.Sprite` bekannt sind

– also beispielsweise `update()`. Methoden wie `rotate()` wären dann der Spritegruppe unbekannt.

Nehmen Sie beispielsweise Zeile 82 in Quelltext 2.32 auf Seite 51. Die Methode `change_direction()` ist dem `GroupSingle`-Objekt `defender` völlig unbekannt, da es ein `Sprite` und kein `Defender`-Objekt erwartet. Syntax-Checker wie `Pylance` werfen hier Fehlermeldungen raus. Eine Möglichkeit das Problem zu umgehen, ist die Verwendung von `update()` als Verteilstation. In der Klasse `pygame.sprite.Sprite` wird diese Methode mit folgender Signatur definiert:

```
update(self, *args: Any, **kwargs: Any) -> None
```

Mit anderen Worten, man kann der Methode beliebige frei definierbare Parameter übergeben. Genau das passiert in unserer `update()`-Methode. Bei der Rotation wird der Übergabeparameter `rotate` mit einem entsprechenden Winkel übergeben, bei der Skalierung der Parameter `scale` und in `update()` von `Game` der Parameter `go` mit dem Wert `True`. Jeder Aufrufer kann also seine Übergabeparameter spontan definieren und mit Werten versehen. Der `update()` in der Kindklasse – hier `Ball` – muss dies nur abfragen.

Dabei wird im ersten Schritt gefragt, ob der Parameter übergeben wurde wie in Zeile 19, Zeile 30, Zeile 33 oder Zeile 36. Anschließend wird der Parameterwert der entsprechenden Methode der Kindklasse übergeben. Somit muss die Spritegruppe nicht auf Methoden der Kindklasse zugreifen, sondern kann die Methode der Elternklasse verwenden.

Quelltext 2.89: Mausaktionen: Ball

```

9  class Ball(pygame.sprite.Sprite):
10     def __init__(self) -> None:
11         super().__init__()
12         fullfilename = os.path.join(Game.PATH["image"], "blue2.png")
13         self.image_orig = pygame.image.load(fullfilename).convert_alpha()
14         self._scale = 10
15         self.image = pygame.transform.scale(self.image_orig, (self._scale, self._scale))
16         self.rect = self.image.get_rect()
17
18     def update(self, *args: Any, **kwargs: Any) -> None:
19         if "go" in kwargs.keys(): # Parameter vorhanden?
20             if kwargs["go"]:
21                 self.rect.left = max(self.rect.left, Game.INNER_RECT.left)
22                 self.rect.right = min(self.rect.right, Game.INNER_RECT.right)
23                 self.rect.top = max(self.rect.top, Game.INNER_RECT.top)
24                 self.rect.bottom = min(self.rect.bottom, Game.INNER_RECT.bottom)
25                 c = self.rect.center # altes Zentrum merken
26                 self.image = pygame.transform.scale(self.image_orig, (self._scale,
27                                         self._scale))
28                 self.rect = self.image.get_rect()
29                 self.rect.center = c # Zentrum zurücksetzen
30
31         if "rotate" in kwargs.keys(): #
32             self.rotate(kwargs["rotate"])
33
34         if "scale" in kwargs.keys(): #
35             self.resize(kwargs["scale"])
36
37         if "center" in kwargs.keys(): #
38             self.set_center(kwargs["center"])

```

```

38
39     def draw(self, screen: pygame.surface.Surface) -> None:
40         screen.blit(self.image, self.rect)
41
42     def rotate(self, angle: float) -> None:
43         self.image_orig = pygame.transform.rotate(self.image_orig, angle)
44
45     def resize(self, delta: int) -> None:
46         if self.rect.width < Game.INNER_RECT.width:
47             if self._scale + delta > 0:
48                 self._scale += delta
49
50     def set_center(self, center: Tuple[int, int]) -> None:
51         self.rect.center = center

```

rotate()

Noch ein Hinweis zu `pygame.transform.rotate()`. Anders als bei vielen anderen Systemen, die Winkel verarbeiten, wird der Winkel hier in **Grad (°)** und nicht in **Radian (rad)** gemessen.

Was war neu?

Mausaktionen werden ähnlich wie Tastaturevents verarbeitet. Die Mausposition kann einfach abgefragt werden. Es ist einfacher den Mauszeiger unsichtbar zu setzen und ein Bitmap der Mausposition folgen zu lassen, als einen neuen Mauszeiger zu setzen.

Es wurden folgende Pygame-Elemente eingeführt:

- `pygame.MOUSEBUTTONDOWN`, `pygame.MOUSEBUTTONUP`:
<https://pyga.me/docs/ref/event.html>
- Liste der Mausbuttons: Tabelle 2.7
- `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>

Tabelle 2.7: Liste der Mausbuttons

Konstante	Beschreibung
0	nicht definiert
1	linke Maustaste
2	mittlere Maustaste/Mausrad
3	rechte Maustaste
4	Mausrad zu sich drehen (up)

Tabelle 2.7: Liste der Mausbuttons (Fortsetzung)

Konstante	Beschreibung
5	Mausrad von sich weg drehen (down)

2.13 Soundausgaben

So ohne Hintergrundgeräusche und/oder -musik wäre manches Spiel einfach nur langweilig. Ich möchte hier daher drei verschiedene Themen in zwei Beispielen vorstellen: Hintergrundmusik bzw. -geräusche, Soundereignisse und Stereoeffekte.

2.13.1 Hintergrundmusik und Soundereignisse

Das erste Beispiel deckt folgende Features ab:

- Eine Hintergrundmusik wird geladen und endlos wiederholend abgespielt.
- Die Lautstärke kann durch das Mausrad manipuliert werden.
- Mit der Taste P kann die Hintergrundmusik pausiert werden bzw. wieder anlaufen.
- Mit der Taste J kann man die Hintergrundmusik ausklingen lassen.
- Über die rechte und die linke Maustaste werden unterschiedliche Soundereignisse ausgegeben.

Der Import, die Klasse `Settings` und die anderen schon bekannten Bausteine möchte ich nicht mehr groß erklären. Sie sind so schon oft vorgekommen.

Quelltext 2.90: Sound: Präambel und `Settings`

```

1 import os
2 from time import time
3 from typing import Any, Tuple
4
5 import pygame
6 from pygame.constants import (
7     K_ESCAPE,
8     K_MINUS,
9     K_PLUS,
10    KEYDOWN,
11    KEYUP,
12    QUIT,
13    K_f,
14    K_j,
15    K_p,
16 )
17
18
19 class Settings:
20     WINDOW: pygame.rect.Rect = pygame.rect.Rect(
21         0, 0, 600, 800
22     ) # Rect
23     FPS = 60
24     PATH: dict[str, str] = {}
25     PATH["file"] = os.path.dirname(os.path.abspath(__file__))
26     PATH["image"] = os.path.join(PATH["file"], "images")
27     PATH["sound"] = os.path.join(PATH["file"], "sounds")
28     START_DISTANCE = 20

```

Bevor der Sound verwendet werden kann, muss das entsprechende Subsystem initialisiert werden. Dies geschieht entweder explizit durch `pygame.mixer.init()` oder wie im

`init()`

Quelltext in Zeile 45 implizit durch `pygame.init()`. In Zeile 52 wird die aktuelle Lautstärke in einem Attribut abgespeichert. Eigentlich ist dies nicht nötig, da man die aktuelle Lautstärke der Hintergrundmusik immer mit `pygame.mixer.music.get_volume()` und die eines Effekts mit `pygame.mixer.Sound.get_volume()` ermitteln kann.

In der Methode `sounds()` sind die vorbereitenden Aktionen zur Soundausgabe gekapselt. Eine Hintergrundmusik wird über `pygame.mixer.music.load()` in den internen Speicher des Mixers geladen. Dadurch wird die Hintergrundmusik aber noch nicht abgespielt. Dies geschieht, nachdem die Lautstärke in Zeile 57 mit `pygame.mixer.music.set_volume()` festgelegt wurde, in der Zeile 58. Die entsprechende Methode `pygame.mixer.music.play()` hat dazu drei Parameter: Der erste Parameter `loops` steuert die Anzahl der Wiederholungen; der Wert `-1` meint dabei, dass die Musik endlos wiederholt wird. Der zweite, `start`, gibt einen Position an, wo die Musik starten soll; der Default ist `0.0`. Soll die Musik leise starten und dann lauter werden (`fade`), kann dies mit dem dritten Parameter `fade` erfolgen; damit können Sie angeben, wie viele Millisekunden dem Lauterwerden zur Verfügung hat; wird nichts angegeben, wird sofort mit der Ziellautstärke gestartet.

Für Soundeffekte wird jeweils ein eigenes `Sound`-Objekt angelegt (Zeile 61f.). Dabei wird dem Konstruktor von `pygame.mixer.Sound` der Dateiname inkl. Pfad angegeben. Für den Fall, dass man eine geöffnete Dateireferenz hat, kann man auch diese übergeben; Sie sollten dann aber einen zweiten Parameter spendieren, der die Soundkodierung z.B. `.ogg` oder `.mp3` angibt. Wie bei der Hintergrundmusik ist auch hierbei *Laden* nicht gleichbedeutend mit *Abspielen*.

Quelltext 2.91: Sound: Konstruktor und `sounds()` von Game

```

31     def get_file(filename: str) -> str:
32         return os.path.join(Settings.PATH["file"], filename)
33
34     @staticmethod
35     def get_image(filename: str) -> str:
36         return os.path.join(Settings.PATH["image"], filename)
37
38     @staticmethod
39     def get_sound(filename: str) -> str:
40         return os.path.join(Settings.PATH["sound"], filename)
41
42
43 class Game:
44     def __init__(self) -> None:
45         pygame.init() # Auch mixer
46         self.screen = pygame.display.set_mode(Settings.WINDOW.size)
47         pygame.display.set_caption('Fingerübung "Sound"')
48         self.clock = pygame.time.Clock()

```

Die Methode `watch_for_events()` ist nur ein Verteiler. Je nachdem welche Taste gedrückt oder welches Mauselement verwendet wurde, werden entsprechende Hilfsmethoden aufgerufen.

Quelltext 2.92: Sound: `watch_for_events()` von Game

```

50     self.running = True
51     self.pause = False
52     self.volume: float = 0.1 # Lautstärke
53     self.sounds()
54
55     def sounds(self) -> None:
56         pygame.mixer.music.load(Settings.get_sound("Lucifer.mid"))
57         pygame.mixer.music.set_volume(self.volume) # Lautstärke
58         pygame.mixer.music.play(-1, 0.0) # Endlos abspielen
59         self.bubble: pygame.mixer.Sound = pygame.mixer.Sound(
60             Settings.get_sound("plopp1.mp3"))
61         ) #
62         self.clash: pygame.mixer.Sound = pygame.mixer.Sound(
63             Settings.get_sound("glas.wav"))
64     )
65
66     def watch_for_events(self) -> None:
67         for event in pygame.event.get():
68             if event.type == QUIT:
69                 self.running = False
70             elif event.type == KEYDOWN:
71                 if event.key == K_ESCAPE:
72                     self.running = False

```

Mit der Hilfsmethode `sound_play()` wird gesteuert, welcher Sound abgespielt werden soll. Das eigentliche Abspielen erfolgt über `pygame.mixer.Sound.play()`. Sie können sehen, dass für das jeweilige Sound-Objekt die Methode `play()` aufgerufen wird. Auch dieses `play` hat drei optionale Argumente: Über `loops` kann die Anzahl der Wiederholungen definiert werden; `-1` steht für endlos und ist die Vorbelegung. `maxtime` beendet nach der angegebenen Anzahl von Millisekunden die Wiedergabe; `0` steht für keine Beendigung und ist die Vorbelegung. `fade_ms` ist die Angabe wie viele Millisekunden das Fadein hat; die Vorbelegung ist `0`.

Werden – wie hier – keine Angaben gemacht, startet die Wiedergabe des Sounds unmittelbar und beendet sich nach dem Abspielen. Eventuell laufende Wiedergaben anderer Sound-Objekte werden dabei nicht abgebrochen.

Quelltext 2.93: Sound: `sound_play()` von Game

```

74     if event.key == K_f:
75         self.music_start_stop(fadeout=5000)
76     elif event.key == K_j:
77         self.music_start_stop(loop=-1)
78     elif event.key == K_p:

```

Die Hintergrundmusik will ich mal starten, mal ausklingen lassen. Dazu dient die Hilfsmethode `music_start_stop()`. Mit `pygame.mixer.music.fadeout()` wird die Musik gestoppt. Dabei muss man angegeben, über wie viele Millisekunden die Musik zum Ende hin leiser wird – in unserem Beispiel sind es `5000 ms`. Die Methode `pygame.mixer.music.play()` zum Starten der Hintergrundmusik wurde oben schon erläutert.

Quelltext 2.94: Sound: `music_start_stop()` von Game

```

80         elif event.key == K_PLUS:
81             self.volume_alter(0.05)
82         elif event.key == K_MINUS:
83             self.volume_alter(-0.05)
84     elif event.type == pygame.MOUSEBUTTONDOWN:

```

Über die Taste P wird die Hintergrundmusik pausiert bzw. wieder gestartet. Der aktuelle Zustand wird im `pause` abgelegt. Dieses Attribut steuert dann in der Methode `pause_alter()` welche der beiden `music`-Methoden – `pygame.mixer.music.pause()` oder `pygame.mixer.music.unpause()` ausgeführt wird. Am Ende wird in Zeile 111 das Flag `pause` umgelegt.

`pause()`
`unpause()`

Quelltext 2.95: Sound: `pause_alter()` von Game

```

86         self.sound_play(bubble=True)
87     elif event.button == 3: # right
88         self.sound_play(clash=True)
89     elif event.button == 4: # up
90         self.volume_alter(0.05)
91     elif event.button == 5: # down

```

Als letztes Feature soll die Lautstärkensteuerung noch vorgestellt werden. Diese ist in der Methode `volume_alter()` gekapselt. Als Übergabeparameter wird dieser Methode nicht eine absolute Lautstärke mitgegeben, sondern ein Veränderungswert.

Zunächst wird dieser Wert auf das Attribut `volume` addiert³. Anschließend wird der Wert auf das Intervall $[0, 1]$ begrenzt und abschließend die neu Lautstärke mit `pygame.mixer.music.set_volume()` gesetzt.

`set_volume()`

Quelltext 2.96: Sound: `volume_alter()` von Game

```

93     def sound_play(self, **kwargs: Any) -> None:
94         if "bubble" in kwargs.keys():
95             self.bubble.play()
96         if "clash" in kwargs.keys():
97             self.clash.play()

```

Und zum Schluss kommt der gute Rest:

Quelltext 2.97: Sound: `draw()`, `update()`, `run()` und Aufruf von Game

```

101    if "fadeout" in kwargs.keys():
102        pygame.mixer.music.fadeout(kwargs["fadeout"])
103    if "loop" in kwargs.keys():
104        pygame.mixer.music.play(kwargs["loop"], 0.0)
105
106    def pause_alter(self) -> None:
107        if self.pause:
108            pygame.mixer.music.unpause()
109        else:

```

³ Bedenken Sie, dass ein negativer Veränderungswert hier die Lautstärke reduziert.

```

110         pygame.mixer.music.pause()
111     self.pause = not self.pause #
112
113     def volume_alter(self, delta: float) -> None:
114         self.volume += delta
115         if self.volume > 1.0:
116             self.volume = 1.0
117         elif self.volume < 0.0:
118             self.volume = 0.0
119         pygame.mixer.music.set_volume(self.volume)
120
121     def draw(self) -> None:
122         self.screen.fill("black")
123
124         volume = self.font_bigsize.render(
125             f" Lautstärke: {self.volume:3.2f} ", True, "red"
126         )
127         rect = volume.get_rect()
128         rect.center = Settings.WINDOW.center
129         self.screen.blit(volume, rect)
130
131     pygame.display.flip()
132
133     def update(self):

```

2.13.2 Stereo

Das zweite Beispiel soll die Funktion von Kanälen und [Stereoeffekte](#) ausleuchten. Das Thema ist für eine vollständige Darstellung zu umfangreich, aber ich hoffe, dass dieses Kapitel einen hilfreichen Einstieg bietet.

In Abbildung 2.43 sehen Sie einen Panzer, der von links nach rechts bzw. von rechts nach links fährt. Während der Fahrt kann er bis zu 5 Schüsse abfeuern. Schön wäre es doch, wenn der Sound der Fahrbewegung akustisch untermauert, wo sich der Panzer gerade befindet. Also, ist der Panzer eher rechts, soll auf dem rechten Lautsprecher das Fahrgeräusch oder der Abschuss lauter sein, als auf dem linken. Bei einer Fahrt von rechts nach links würde also auch das Fahrgeräusch mitwandern.

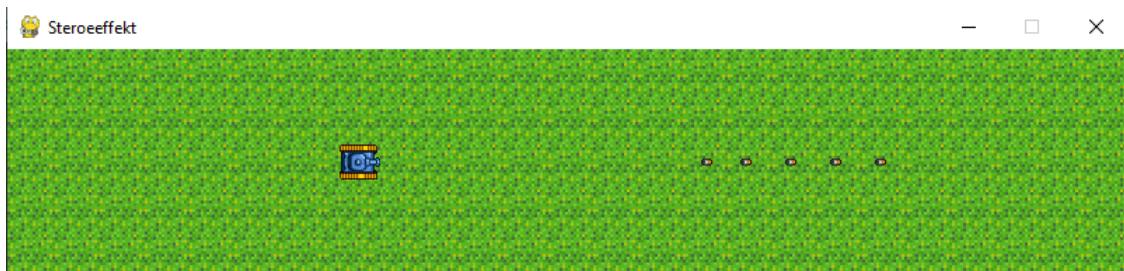


Abbildung 2.43: Sound: Stereoeffekt

Zunächst das notwendige Beiwerk, welches ich nicht weiter erklären müssen sollte:

Quelltext 2.98: Sound-Stereo: Präamble, `Settings` und `Ground`

```

1  import os
2  from time import time
3  from typing import Any, Tuple
4
5  import pygame
6  from pygame.constants import (K_DOWN, K_ESCAPE, K_LEFT, K_RIGHT, K_SPACE, K_UP,
7                                KEYDOWN, QUIT)
8
9
10 class Settings:
11     WINDOW: pygame.rect.Rect = pygame.rect.Rect(0, 0, 800, 160)
12     FPS = 60
13     DELTATIME = 1.0/FPS
14     PATH: dict[str, str] = {}
15     PATH["file"] = os.path.dirname(os.path.abspath(__file__))
16     PATH["image"] = os.path.join(PATH["file"], "images")
17     PATH["sound"] = os.path.join(PATH["file"], "sounds")
18
19     @staticmethod
20     def get_file(filename: str) -> str:
21         return os.path.join(Settings.PATH["file"], filename)
22
23     @staticmethod
24     def get_image(filename: str) -> str:
25         return os.path.join(Settings.PATH["image"], filename)
26
27     @staticmethod
28     def get_sound(filename: str) -> str:
29         return os.path.join(Settings.PATH["sound"], filename)
30
31
32 class Ground(pygame.sprite.Sprite):
33
34     def __init__(self) -> None:
35         super().__init__()
36         fullfilename = Settings.get_image("tankbrigade_part64.png")
37         tile = pygame.image.load(fullfilename).convert()
38         rect = tile.get_rect()
39         self.image = pygame.Surface(Settings.WINDOW.size)
40         for row in range(Settings.WINDOW.width // rect.width):
41             for col in range(Settings.WINDOW.height // rect.height):
42                 self.image.blit(tile, (row * rect.width, col * rect.height))
43         self.rect = self.image.get_rect()

```

In Zeile 65 wird ein Sound-Objekt erzeugt. Dieses wird abgespielt, um die Fahrt des Panzers mit entsprechenden Geräuschen hervorzuheben. In der Zeile danach (Zeile 65) wird die Hilfsmethode `stereo()` aufgerufen (s.u.) und anschließend beginnt die Wiedergabe des Fahrgeräuschs in einer Endlosschleife (Zeile 68). Dabei fällt auf, dass hier die Ausgabe nicht über `pygame.mixer.Sound.play()` erfolgt.

Sound-
Objekt

Normalerweise, wäre dies eine gute Idee gewesen, wählt dieser Befehl doch einen der acht verfügbaren Sound-Kanäle aus. Man kann aber auch einen Kanal direkt ansteuern und damit mehr Kontrolle über das Sound-Verhalten erlangen. In Zeile 66 wird dazu ein freies `pygame.mixer.Channel`-Objekt ermittelt. Die Methode `pygame.mixer.find_channel()` liefert mir nämlich den ersten freien Kanal und speichert diesen im Attribut

Kanal

find_channel()

channel ab. Das Abspielen erfolgt dann in Zeile 68 nicht mehr über eine Methode des Sound-Objektes, sondern mit Hilfe von `pygame.mixer.Channel.play()`.

Quelltext 2.99: Sound-Stereo: Konstruktor von Tank

```

46 class Tank(pygame.sprite.Sprite):
47
48     def __init__(self) -> None:
49         super().__init__()
50         self.image_filename = (209, 190, 202, 214, 226, 238, 250, 262)
51         self.images: dict[str, list[pygame.surface.Surface]] = {"up": [], "down": [],
52                     "left": [], "right": []}
53         for number in self.image_filename:
54             fullfilename = Settings.get_image(f"tankbrigade_part{number}.png")
55             picture = pygame.image.load(fullfilename).convert()
56             picture.set_colorkey("black")
57             self.images["up"].append(picture)
58             self.images["down"].append(pygame.transform.rotate(picture, 180))
59             self.images["left"].append(pygame.transform.rotate(picture, +90))
60             self.images["right"].append(pygame.transform.rotate(picture, -90))
61         self.direction = "right"
62         self.imageindex = 0
63         self.image = self.images[self.direction][self.imageindex]
64         self.rect = pygame.Rect(self.image.get_rect())
65         self.rect.left, self.rect.top = 3 * self.rect.width, 2 * self.rect.height
66         self.sound_drive = pygame.mixer.Sound(Settings.get_sound("tank_drive1.wav")) #
67         self.channel = pygame.mixer.find_channel()                      # Sound-Kanal finden
68         self.stereo()                                              #
69         self.channel.play(self.sound_drive, -1)                      #
70         self.speed = 50

```

Die Methode `update()` wird hier nur der Vollständigkeit halber abgedruckt. Bzgl. der Geräuschkulisse passiert hier nichts.

Quelltext 2.100: Sound-Stereo: Tank.update()

```

71     def update(self, *args: Any, **kwargs: Any) -> None:
72         if "go" in kwargs.keys():
73             if kwargs["go"]:
74                 self.update_imageindex()
75                 self.image = self.images[self.direction][self.imageindex]
76                 if self.direction == "up" or self.direction == "left":
77                     self.speed = -50
78                 elif self.direction == "down" or self.direction == "right":
79                     self.speed = 50
80                 if self.direction == "up" or self.direction == "down":
81                     self.rect.move_ip(0, self.speed * Settings.DELTATIME)
82                     if self.rect.top <= Settings.WINDOW.top:
83                         self.turn("down")
84                         if self.rect.bottom >= Settings.WINDOW.bottom:
85                             self.turn("up")
86                     elif self.direction == "left" or self.direction == "right":
87                         self.rect.move_ip(self.speed * Settings.DELTATIME, 0)
88                         if self.rect.left <= Settings.WINDOW.left:
89                             self.turn("right")
90                             if self.rect.right >= Settings.WINDOW.right:
91                                 self.turn("left")
92                         self.stereo()
93                     if "turn" in kwargs.keys():
94                         self.turn(kwargs["turn"])

```

Die Methode `stereo()` ist überraschend simpel. Die Methode `pygame.mixer.Channel.set_volume()` stellt nämlich zwei Übergabeparameter zur Verfügung: `left` und `right`. Beide haben einen Wertebereich von $[0, 1]$. Nun wollten wir ja, dass der rechte Lautsprecher das Motorengeräusch lauter wiedergibt je weiter rechts der Panzer steht und umgekehrt. Dazu berechne ich in Zeile 68 die relative Position des Panzerzentrums in der Waagerechten im Verhältnis zur Fensterbreite; gibt ja auch einen Wert im Intervall von $[0, 1]$. Habe ich diesen Wert, kann ich in der folgenden Zeile ebenfalls die relative linke Position ermitteln. Danach werden beide Werte der Methode `set_volume()` übergeben.

`set_volume()`

Hinweis: Der Methode `pygame.mixer.Channel.set_volume()` können unterschiedliche Lautstärken für Rechts und Links mitgegeben werden, den Methoden `pygame.mixer.Sound.set_volume()` und `pygame.mixer.music.set_volume()` nicht.

Quelltext 2.101: Sound-Stereo: `Tank.stereo()`

```
96     def stereo(self) -> None:
97         volume_rechts = self.rect.centerx / Settings.WINDOW.width  #
98         volume_links = 1 - volume_rechts
99         self.channel.set_volume(volume_links, volume_rechts)
```

Wozu könnte dieser Effekt noch genutzt werden? Denken wir beispielsweise an zwei Personen, die miteinander sprechen, Geräuschquellen in einem Raum, usw.. Immer dann, wenn durch die Akustik die Lokalisierung erleichtert werden soll, oder Einzelgeräusche abgehoben bzw. unterschieden werden sollen, bieten sich unterschiedliche Lautstärken – also Stereo – an.

In `turn()` und `update_imageindex()` passiert nichts bzgl. der Soundausgabe.

Quelltext 2.102: Sound-Stereo: `Tank.turn()` und `Tank.update_imageindex()`

```
101    def turn(self, direction: str) -> None:
102        self.direction = direction
103
104    def update_imageindex(self) -> None:
105        if self.speed == 0:
106            self.imageindex = 0
107        else:
108            self.imageindex = (self.imageindex + 1) % len(self.images[self.direction])
```

Die Soundausgabe des `Bullet` hätte man auch in der Klasse `Tank` programmieren können. Ich finde es aber organischer, diese in `Bullet` zu verorten. Vielleicht wollte man ja später auch noch einen Aufprall oder ein Explosion implementieren.

Vor dem Konstruktor wird in Zeile 113 die statische Variable `_sound_fire` definiert. Wir haben zwar viele Geschosse, aber alle nutzen den gleichen Abschusssound. Somit wäre es eine Speicherplatz- und Performanceverschwendung diesen Sound immer wieder neu zu lesen und ein entsprechendes Objekt zu erzeugen. Vielmehr erfolgt ab Zeile 133 eine Art `Singleton`-Prüfung. Dabei wird sicher gestellt, dass nur ein einiges mal die Sounddatei gelesen und das entsprechende Objekt erzeugt wird.

Anschließend wird wie beim Panzer ein freier Kanal gesucht und die Lautstärke des rechten und linken Lautsprechers abhängig von der Position bestimmt. Zum Schluss wird der Sound abgespielt.

Quelltext 2.103: Sound-Stereo: Die Klasse Bullet

```

111 class Bullet(pygame.sprite.Sprite):
112
113     _sound_fire = None                                # Es braucht nur einen
114
115     def __init__(self, tank: Tank) -> None:
116         super().__init__()
117         bulletspeed = 300
118         number: dict[str, int] = {"left": 49, "right": 61, "up": 37, "down": 73}
119         directions = {
120             "left": pygame.Vector2(-bulletspeed, 0),
121             "right": pygame.Vector2(bulletspeed, 0),
122             "up": pygame.Vector2(0, -bulletspeed),
123             "down": pygame.Vector2(0, bulletspeed),
124         }
125         fullfilename = os.path.join(Settings.PATH["image"],
126             f"tankbrigade_part{number[tank.direction]}.png")
127         self.image = pygame.image.load(fullfilename).convert()
128         self.image.set_colorkey("black")
129         self.rect = self.image.get_rect()
130         self.direction = tank.direction
131         self.rect.center = tank.rect.center
132         self.speed = directions[tank.direction]
133
133     if Bullet._sound_fire == None:                      # Es braucht nur einen
134         Bullet._sound_fire = pygame.mixer.Sound(Settings.get_sound("tank_fire1.wav"))
135     volume_rechts = self.rect.centerx / Settings.WINDOW.width
136     volume_links = 1 - volume_rechts
137     self.channel: pygame.mixer.Channel = pygame.mixer.find_channel()
138     self.channel.set_volume(volume_links, volume_rechts)
139     self.channel.play(Bullet._sound_fire)
140
141     def update(self, *args: Any, **kwargs: Any) -> None:
142         self.rect.move_ip(self.speed * Settings.DELTATIME)
143         if not Settings.WINDOW.contains(self.rect):
144             self.kill()

```

Der Rest des Quelltextes wird nur der Vollständigkeit wegen abgedruckt.

Quelltext 2.104: Sound-Stereo: Rest

```

147 class Game:
148
149     def __init__(self) -> None:
150         pygame.init()
151         self._screen = pygame.display.set_mode(Settings.WINDOW.size)
152         pygame.display.set_caption("Stereeeffekt")
153         self.clock = pygame.time.Clock()
154         self.ground = pygame.sprite.GroupSingle(Ground())
155         self.tankreference = Tank()
156         self.tank = pygame.sprite.GroupSingle(self.tankreference)
157         self.all_bullets = pygame.sprite.Group()
158         self.running = True
159
160     def watch_for_events(self) -> None:
161         for event in pygame.event.get():
162             if event.type == QUIT:

```

```

163         self.running = False
164     elif event.type == KEYDOWN:
165         if event.key == K_ESCAPE:
166             self.running = False
167         elif event.key == K_UP:
168             self.tank.update(turn="up")
169         elif event.key == K_DOWN:
170             self.tank.sprite.update(turn="down")
171         elif event.key == K_LEFT:
172             self.tank.sprite.update(turn="left")
173         elif event.key == K_RIGHT:
174             self.tank.sprite.update(turn="right")
175         elif event.key == K_SPACE:
176             self.fire()
177
178     def fire(self) -> None:
179         if len(self.all_bullets) < 5:
180             self.all_bullets.add(Bullet(self.tankreference))
181
182     def draw(self) -> None:
183         self.ground.draw(self._screen)
184         self.tank.draw(self._screen)
185         self.all_bullets.draw(self._screen)
186         pygame.display.flip()
187
188     def update(self) -> None:
189         self.tank.update(go=True)
190         self.all_bullets.update()
191
192     def run(self) -> None:
193         time_previous = time()
194         self.running = True
195         while self.running:
196             self.watch_for_events()
197             self.update()
198             self.draw()
199             self.clock.tick(Settings.FPS)
200             time_current = time()
201             Settings.DELTATIME = time_current - time_previous
202             time_previous = time_current
203         pygame.quit()

```

Was war neu?

Für die Soundunterstützung stehen zwei Möglichkeiten zur Verfügung. Einmal das Abspielen einer Hintergrundmusik und zum anderen einzelne Sounds über verschiedene Kanäle und, wenn möglich, auf den rechten und linken Lautsprecher verteilt.

Es wurden folgende Pygame-Elemente eingeführt:

- `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.music.fadeout()`:
<https://pyga.me/docs/ref/music.html#pygame.mixer.music.fadeout>
- `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

2.14 Dirty Sprites

Derzeit wird in jedem Frame der gesamte Bildschirm – also Hintergrund und Sprites – neu gezeichnet. Dies ist, besonders wenn sich eigentlich nur wenige Sprites bewegen oder ihr Aussehen verändern, Rechenzeitverschwendungen.

Pygame stellt dem das Konzept der *Dirty Sprites* entgegen. Dabei wird über das Flag `pygame.sprite.DirtySprite.dirty` gesteuert, ob das Sprite neu gezeichnet werden muss oder nicht. Auch muss der Sprite irgendwie mitgeteilt werden, was auf die alte Position gezeichnet werden soll, wenn sich sie bewegt oder verschwindet; wird doch der Hintergrund eben nicht in jedem Frame neu gezeichnet.

Dirty Sprite
`self.dirty`

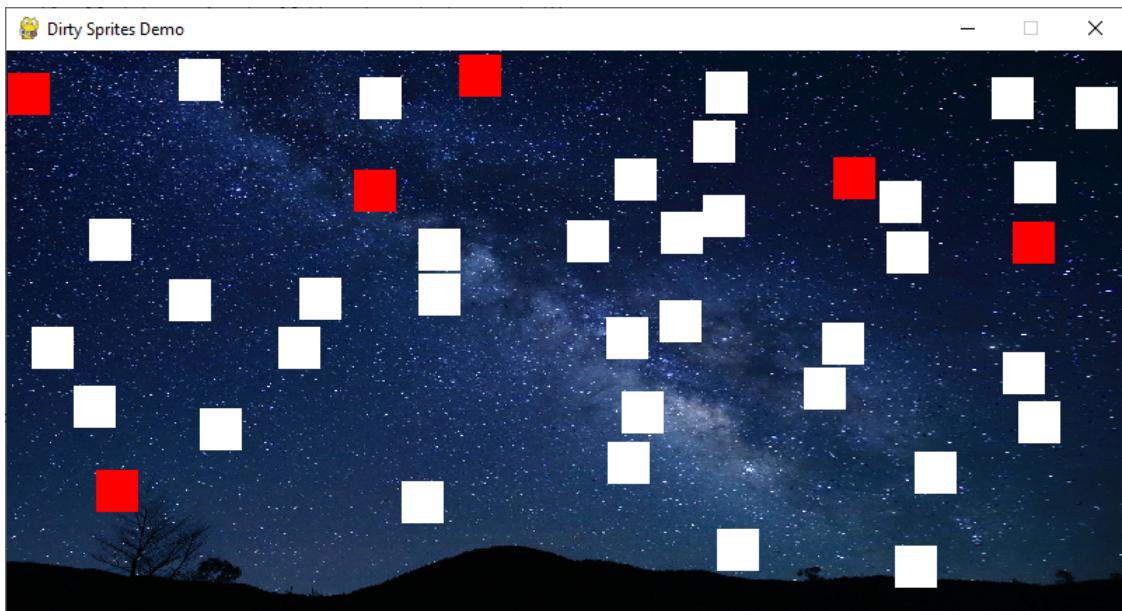


Abbildung 2.44: Dirty Sprite – Demo-Spiel

2.14.1 Einfaches Beispiel

Wir haben hier ein kleines, relativ simples Spiel (siehe Abbildung 2.44) ohne besonderen Anspruch an Logik oder Ästhetik mit folgenden Features:

- Der Bildschirmhintergrund ist ein Sternenhimmel.
- Vor dem Hintergrund erscheinen weiße Quadrate.
- Diese Quadrate werden per Zufall rot.
- Nach einer gewissen Zeitspanne werden die Quadrate wieder weiß.
- Mausklickt man ein rotes Quadrat, verschwindet es.
- Das Spiel beendet sich, wenn alle Quadrate verschwunden sind.

- Ziel ist es, die Quadrate in möglichst kurzer Zeit wegzu klicken.

Das eigentliche Spiel ist etwas anspruchsvoller. Dabei werden die Quadrate in einer gewissen Reihenfolge die Farbe ändern und die Quadrate müssen in dieser Reihenfolge (Kette) angeklickt werden. Mit jedem Level werden die Ketten länger und die Quadrate kleiner. Aber das wäre für unsere Einführung nur überflüssiger Ballast.

Zunächst ein paar Worte über das noch nicht umgebaute Spiel. Die Klassen `Settings` und `Timer` werde ich nicht mehr kommentieren, da die schon ausführlich besprochen wurden.

Die Klasse `Tile`: Eine sehr simple Klasse, die im Konstruktor ein farbiges Rechteck-Surface erzeugt. In `Tile.update()` werden die Zustandsänderungen definiert. Soll ein Farbwechsel erfolgen (`action='switch'`) so kann dies nur passieren, wenn der Status der Kachel den Wert 0 hat, was bedeutet, dass die Kachel noch weiß ist. Nur dann wird ein `Timer` mit einer Periodendauer von 500 ms gestartet, die Farbe gewechselt und der Status auf 1 gesetzt. Dadurch wird markiert, dass nun die Kachel durch Anklicken zerstört werden kann.

Wird ein Kill-Signal gesendet (`action='kill'`), wird die Kachel nur dann gelöscht, wenn der Status den Wert 1 hat, also rot eingefärbt ist. Ansonsten wird der Timer überprüft und ggf. Status und Farbe wieder zurückgesetzt.

Erwähnenswert ist noch, dass hier Farbnamen anstelle von RGB-Werten verwendet werden (siehe Zeile 18 und Zeile 26). Dies ist möglich, da in Pygame schon eine große Anzahl von Farben vordefiniert sind. Überall dort, wo ein RGB-Code oder ein Farbwert angegeben werden kann, z.B. im Konstruktor eines `Color`-Objekts, können diese vordefinierten Farbnamen als Strings angegeben werden.

Farbnamen

Quelltext 2.105: Dirty Sprite – `Tile` (unverändertes Demo)

```

1 import os
2 from random import randint
3 from typing import Any, Tuple
4
5 import pygame
6 from pygame import K_ESCAPE, KEYDOWN, MOUSEBUTTONDOWN, QUIT
7
8 from mytools import Timer
9 from settings import Settings
10
11
12 class Tile(pygame.sprite.Sprite):
13     def __init__(self, topleft: Tuple[int, int]) -> None:
14         super().__init__()
15         self.rect = pygame.Rect(0, 0, Settings.size, Settings.size)
16         self.rect.topleft = topleft
17         self.image = pygame.surface.Surface((self.rect.width, self.rect.height))
18         self.image.fill("white")                                     # Vordefinierte Farbnamen
19         self.timer: Timer
20         self.status = 0
21
22     def update(self, *args: Any, **kwargs: Any) -> None:
23         if "action" in kwargs.keys():
24             if kwargs["action"] == "switch" and self.status == 0:
25                 self.timer = Timer(500, False)

```

```

26         self.image.fill("red")                                # Vordefinierte Farbnamen
27         self.status = 1
28     if kwargs["action"] == "kill" and self.status == 1:
29         self.kill()
30     if self.status == 1 and self.timer.is_next_stop_reached():
31         self.image.fill("white")
32         self.status = 0

```

Die Klasse `Game` ist auch recht einfach. Herzstück ist die Methode `Game.update()`: Über den Timer `self.timer` wird in jeder Sekunde eine zufällige Kachel ausgewählt und die Farbe von weiß auf rot gedreht. Ist keine Kachel mehr vorhanden, weil diese nach und nach zerstört wurden, wird das Spiel beendet.

Für unser eigentliches Thema ist die Methode `Game.draw()` interessant. Hier können Sie sehen, dass in jedem Frame der komplette Hintergrund und alle Kacheln ausgegeben werden, obwohl nur wenige Kacheln ihr Aussehen zwischen zwei Frames verändern bzw. zerstört wurden. Eine offensichtliche Rechenzeitvernichtung. Nehmen wir beispielsweise nur das ständige Zeichnen des Hintergrundes. Bei einer Spielfeldgröße von $800\text{ px} \times 400\text{ px}$ sind hier *60mal* in der Sekunde 320.000 Pixel zu zeichnen, obwohl zwischen zwei Frames eher nur eine Kachel verschwindet, also bei einer Kachelgröße von $30\text{ px} \times 30\text{ px}$ nur 900 Pixel neu zu zeichnen wären.

Quelltext 2.106: Dirty Sprite – Game (unverändertes Demo)

```

35 class Game:
36     def __init__(self) -> None:
37         pygame.init()
38         self.screen = pygame.display.set_mode(Settings.WINDOW.size)
39         pygame.display.set_caption("DirtySpritesDemo")
40         self.clock = pygame.time.Clock()
41         self.all_tiles = pygame.sprite.Group()
42         self.create_playground()
43         self.background_image = pygame.image.load(Settings.get_image("background.png"))
44         self.background_image = pygame.transform.scale(self.background_image,
45             (Settings.WINDOW.size))
46         self.timer = Timer(1000, False)
47         self.running = True
48
49     def watch_for_events(self) -> None:
50         for event in pygame.event.get():
51             if event.type == QUIT:
52                 self.running = False
53             elif event.type == KEYDOWN:
54                 if event.key == K_ESCAPE:
55                     self.running = False
56             elif event.type == MOUSEBUTTONDOWN:
57                 if event.button == 1:
58                     self.klick(pygame.mouse.get_pos())
59
60     def draw(self) -> None:
61         self.screen.blit(self.background_image, (0, 0))
62         self.all_tiles.draw(self.screen)
63         pygame.display.flip()
64
65     def update(self):
66         if len(self.all_tiles) == 0:
67             self.running = False
68         if self.timer.is_next_stop_reached():
69             index = randint(0, len(self.all_tiles) - 1)

```

```

69         self.all_tiles.sprites()[index].update(action="switch")
70
71     def run(self):
72         self.running = True
73         while self.running:
74             self.clock.tick(Settings.FPS)
75             self.watch_for_events()
76             self.update()
77             self.draw()
78
79         pygame.quit()
80
81     def create_playground(self) -> None:
82         for _ in range(Settings.number):
83             tries = 10
84             while tries > 0:
85                 left = randint(0, Settings.WINDOW.width - Settings.size)
86                 top = randint(0, Settings.WINDOW.height - Settings.size)
87                 tile = Tile((left, top))
88                 collided = pygame.sprite.spritecollide(tile, self.all_tiles, False)
89                 if len(collided) == 0:
90                     self.all_tiles.add(tile)
91                     break
92                 tries -= 1
93
94     def klick(self, mousepos: Tuple[int, int]) -> None:
95         for tile in self.all_tiles.sprites():
96             if tile.rect.collidepoint(mousepos):
97                 tile.update(action="kill")

```

DirtySprite Wie oben schon erwähnt, wird uns von Pygame die Klasse `pygame.sprite.DirtySprite` zur Verfügung gestellt. Diese Klasse wird von `pygame.sprite.Sprite` abgeleitet und hat insbesondere ein zusätzliches Attribut, welches steuert, ob der Sprite neu gezeichnet werden muss oder nicht: `pygame.sprite.DirtySprite.dirty`. Dieses Attribut kann drei verschiedene Werte annehmen. In Tabelle 2.9 auf Seite 126 werden die Bedeutungen angegeben.

Fangen wir also mit dem Umbau an, und machen aus `Tile` eine Kindklasse von `DirtySprite`. In Zeile 12 wird der Name der Elternklasse angepasst. Im Konstruktor sollte man `self.dirty` auf 1 setzen, damit der Sprite auf jeden Fall beim ersten `draw()` ausgegeben wird (siehe Zeile 21)⁴. Anschließend müssen Sie die Stellen im Quelltext finden, die das Aussehen oder die Position ihres Sprites verändern. Wird eine solche Veränderung vorgenommen, muss `self.dirty` auf 1 gesetzt werden. Dies ist in der Regel die Methode `update()` oder solche, die von ihr aufgerufen werden.

In unserem Beispiel wird an zwei Stellen das Aussehen verändert. Zum einen, wenn das Signal 'switch' gesendet wird (siehe Zeile 29) und zum anderen, wenn der interne Timer die Farbe wieder von rot nach weiß abändert (siehe Zeile 35). Wie in Tabelle 2.9 auf Seite 126 beschrieben, wird der Wert automatisch nach der Ausgabe wieder auf 0 gesetzt.

Stellt sich noch die Frage, warum nicht auch beim Kill das `self.dirty` angepasst wird. Wird der Sprite entfernt, soll er ja gerade nicht neu gezeichnet werden; stattdessen soll

⁴ Der Wert 1 ist die Vorbelegung für dieses Attribut; ein Setzen ist daher nicht zwingend nötig, aber wegen der besseren Verständlichkeit angebracht.

ja der Hintergrund an dieser Stelle nachgezeichnet werden. Wie das geschieht, wird in Game geregelt.

Quelltext 2.107: Dirty Sprite – Tile (Umbau)

```

12 class Tile(pygame.sprite.DirtySprite):                      # Neue Super-Klasse
13     def __init__(self, topleft: tuple[int, int]) -> None:
14         super().__init__()
15         self.rect = pygame.rect.Rect(0, 0, Settings.size, Settings.size)
16         self.rect.topleft = topleft
17         self.image = pygame.surface.Surface((self.rect.width, self.rect.height))
18         self.image.fill("white")
19         self.timer: Timer
20         self.status = 0
21         self.dirty = 1                                         # Erstmaliges Zeichnen
22
23     def update(self, *args: Any, **kwargs: Any) -> None:
24         if "action" in kwargs.keys():
25             if kwargs["action"] == "switch" and self.status == 0:
26                 self.timer = Timer(500, False)
27                 self.image.fill("red")
28                 self.status = 1
29                 self.dirty = 1                                     # Muss neu gezeichnet werden
30             if kwargs["action"] == "kill" and self.status == 1:
31                 self.kill()
32             if self.status == 1 and self.timer.is_next_stop_reached():
33                 self.image.fill("white")
34                 self.status = 0
35                 self.dirty = 1                                     # Muss neu gezeichnet werden

```

Für die Liste der Kacheln, können wir nun kein `pygame.sprite.Group`-Objekt mehr nehmen, da diese Liste keine Attribute und Methoden von `pygame.sprite.DirtySprite` kennt. Die passende Alternative zur `Group`-Klasse ist `pygame.sprite.LayeredDirty`. Diese Klasse enthält alle Mechanismen, die wir für die Unterstützung von `DirtySprite` brauchen. Bauen wir daher erstmal das entsprechende Attribut in Zeile 46 um.

LayeredDirty

Quelltext 2.108: Dirty Sprite – Konstruktor von Game (Umbau)

```

38 class Game:
39     def __init__(self) -> None:
40         pygame.init()
41         self.screen = pygame.display.set_mode(Settings.WINDOW.size)
42         pygame.display.set_caption("DirtySpritesDemo")
43         self.clock = pygame.time.Clock()
44         self.background_image = pygame.image.load(Settings.get_image("background.png"))
45         self.background_image = pygame.transform.scale(self.background_image,
46                                         (Settings.WINDOW.size))
46         self.all_tiles = pygame.sprite.LayeredDirty()          # Gruppenklasse für DirtySprite
47         self.create_playground()
48         self.timer = Timer(1000, False)
49         self.running = True

```

Würden wir dieses Programm nun ausführen, bekämen wir ein sehr unbefriedigendes Ergebnis zu sehen. Einmal erscheinen ganz kurz (nur für ein Frame) alle Kacheln in weiß und danach ist nur noch der Hintergrund zu sehen. Ab und zu blitzen weiße und rote Kacheln auf, und zwar immer dann, wenn ein Farbwechsel erfolgt, also `dirty` auf 1 gesetzt wurde. Wir müssen hier also noch weitere Veränderungen vornehmen.

draw()

Zunächst müssen wir uns klar machen, dass nun in Zeile 63 nicht mehr das `draw()` von `Group`, sondern von `LayeredDirty` aufgerufen wird. Diese Methode kennt nämlich noch einen zweiten Parameter: das Hintergrundbitmap. Verschwindet nun eine Kachel, weil darauf geklickt wurde, wird der passende Ausschnitt aus dem Hintergrundbitmap an Stelle der Kachel ausgegeben. Auch merkt `LayeredDirty`, dass der Hintergrund vorher noch nicht ausgegeben wurde und blittet ihn beim ersten Aufruf von `draw()` einmalig komplett. Deshalb kann die Zeile, die in der vorherigen Version den Hintergrund immer blittet, entfallen.

update()

Ein weiterer Unterschied von `LayeredDirty.draw()` ist, dass es eine Liste von Rechtecken zurückliefert. Und zwar nur von den Rechtecken, die geänderte Bildschirmbereiche markieren. Verwenden wir nun nicht mehr `pygame.display.flip()`, sondern `pygame.display.update()` (siehe Zeile 64), so können wir diese veränderten Bildschirmbereiche als Parameter übergeben und nur diese Bereiche werden dann neu gezeichnet.

Quelltext 2.109: Dirty Sprite – `Game.draw()` (Umbau)

```
62     def draw(self) -> None:
63         rects = self.all_tiles.draw(self.screen, self.background_image)  #
64         pygame.display.update(rects)      # Kein flip() mehr
```

Wenn wir jetzt das Programm ausführen, sieht alles soweit ganz gut aus. Es verbleiben mir aber noch zwei Kleinigkeiten. In `draw()` wird jetzt bei jedem Aufruf in Zeile 63 das Backgroundimage mitgeliefert. Das passiert immerhin 60 mal pro Sekunde. Wäre es nicht schöner, wir würden einmal das den Hintergrund setzen und könnten dann auf den zweiten Parameter in der Zeile verzichten? Na klar wäre das schöner und deshalb geht das auch ;-)

clear()

In Zeile 47 wird der Hintergrund mit `pygame.sprite.LayeredDirty.clear()` gesetzt. Auch wird festgelegt, auf welches Surface der Hintergrund gezeichnet werden soll, hier eben auf `self.screen`.

Quelltext 2.110: Dirty Sprite – Konstruktor von `Game` (Umbau)

```
46         self.all_tiles = pygame.sprite.LayeredDirty()
47         self.all_tiles.clear(self.screen, self.background_image)  #
48         self.all_tiles.set_timing_threshold(1000.0/Settings.FPS)  #
49         self.create_playground()
```

Deshalb kann in `draw()` auf den zweiten Parameter verzichtet werden (Zeile 65).

Quelltext 2.111: Dirty Sprite – Game.draw() (Umbau)

```

64  def draw(self) -> None:
65      rects = self.all_tiles.draw(self.screen)  # 2ter Parameter entfällt
66      pygame.display.update(rects)

```

Ich sprach aber von zwei Kleinigkeiten. Für die zweite muss ich ein wenig was erklären. Die ganze Idee um den `DirtySprite` herum ist ja, Performance dadurch einzusparen, dass man nur noch die veränderten Bildschirmbereiche aktualisiert. Nun kostet aber das Ermitteln und Ausschneiden dieser Bereiche ebenfalls Rechenzeit. In der Informatik spricht man dabei von einem `Trade-off`. Diese Rechenzeit kann das Zeitfenster, welches dafür innerhalb eines Frames zur Verfügung steht, überschreiten und damit die Bildschirmausgabe verlangsamen bzw. qualitativ verschlechtern. Daher wird während der Ausführung von `draw()` in `LayeredDirty` die Ausführungszeit gemessen⁵. Wird das Zeitfenster überschritten, merkt sich das `LayeredDirty` und blättert beim nächsten mal Hintergründe und Sprites, als ob keine `DirtySprite`-Logik verwendet wird.

Aber woher soll nun `DirtyLayer` wissen, wie lang das verfügbare Zeitfenster ist? Eben dafür ist die Methode `pygame.sprite.LayeredDirty.set_timing_threshold()` zuständig (siehe Zeile 48). Im Handbuch wird vorgeschlagen, den Wert $1000.0/FPS$ zu nehmen. Warum? Eine Sekunde besteht aus 1000 Millisekunden. Teilt man diese 1000 durch die Anzahl der Frames pro Sekunde, erhält man die Anzahl der Millisekunden, die ein Frame zur Verfügung hat; bei uns sind es ca. 16 ms.

`set_timing_threshold()`

2.14.2 Performancemessungen

Der letzte Absatz im vorherigen Abschnitt hat mich misstrauisch gemacht. Sind `DirtySprite`-Objekte wirklich schneller als normale? Und ich muss gestehen, dass ich erst recht erschreckende Ergebnisse bekommen habe. Aber der Reihe nach.

Zunächst habe ich obiges Beispiel ein wenig umgebaut. Die Kacheln werden dabei nicht mit der Maus angeklickt, sondern nach zweimaligem Farbwechsel löschen die sich selbst. Sind dann keine Kacheln mehr da, beendet sich der Testlauf. An der entscheidenden Stelle habe ich dann Zeitwerte abgegriffen, um diese in eine Datei zu schreiben.

Quelltext 2.112: Performancevergleich – Messung

```

71  def run(self):
72      self.running = True
73      while self.running:
74          self.clock.tick(Settings.FPS)
75          start = time.perf_counter()
76          self.watch_for_events()
77          self.update()
78          self.draw()
79          duration = time.perf_counter() - start
80          self.performance.append(duration)
81      with open(Settings.get_file(f"perf0_{Settings.size}_{Settings.number}.txt"), "w") as
82          datei:

```

⁵ Siehe https://github.com/pygame-community/pygame-ce/blob/main/src_py/sprite.py.

```

82         for item in self.performance:
83             datei.write(f"{item}\n")
84         pygame.quit()

```

Dann habe ich Testläufe mit den Parametern aus Tabelle 2.8 auf einem Schul-PC laufen lassen. Die Werte waren unterschiedlich⁶, aber die Tendenz immer die gleiche. Ich habe mal ein Ergebnis in einer Grafik visualisiert (siehe Abbildung 2.45). Das Ergebnis ist eindeutig, DirtySprites sind signifikant schneller.

Tabelle 2.8: Testkonfiguration Performancemassung

Testlauf	Kachelgröße	Anzahl Kacheln
1	5 px × 5 px	100
2	5 px × 5 px	4000
3	30 px × 30 px	100
4	50 px × 50 px	100
5	100 px × 100 px	40

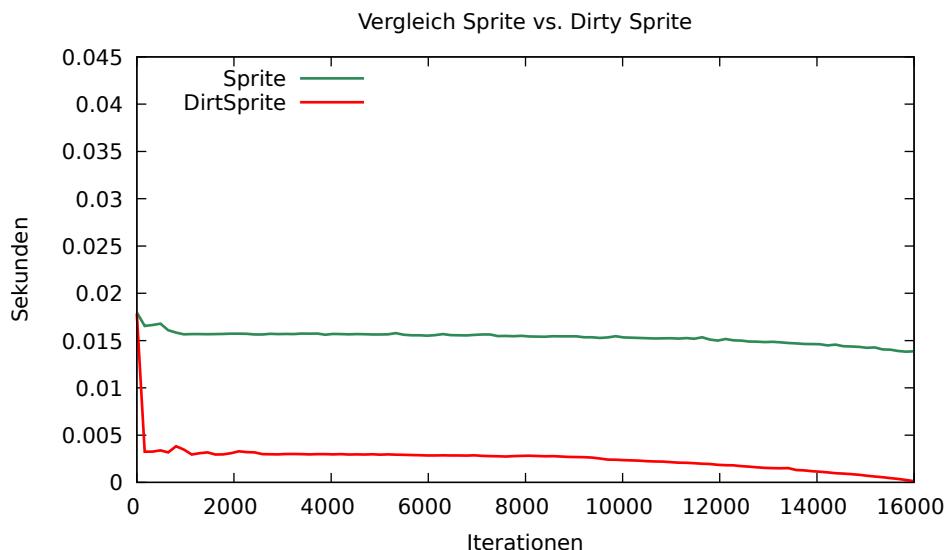


Abbildung 2.45: Performancevergleich mit Testkonfiguration 2

Doch leider meinte ich, das Experiment zu Hause wiederholen zu müssen, da das Programm noch ein wenig nachoptimiert wurde. Und dann – oh Schreck – kam Abbildung 2.46 auf der nächsten Seite raus. Natürlich fieberhaft nach einem Fehler gesucht, schließlich hatte ich ja Kleinigkeiten verändert. Test auf einem Surface Pro wiederholt

⁶ Die Werte werden hier bereinigt verwendet. So wurden durch Störungen wie beispielsweise eingehende E-Mails kurzfristige Spitzen erzeugt, die rausgerechnet wurden. Ebenso wurden die Anzahl der Messwerte angeglichen.

und schon wieder recht schlechte Ergebnisse (Abbildung 2.47). Hier war es sogar noch verwirrender, da die ersten rund 8.000 Iterationen einen Vorteil für DirtySprites ergaben und danach die Zeitverbräuche sich angleichten.

Nach vielen Wiederholungstest auf den drei Rechnern, kam mir der Gedanke, dass der PC zu Hause und der Surface Pro vielleicht nicht die Framerate von 60 *fps* schaffen und daher das Zeitfenster zu klein ist. Also den Test mit kleinerer Framerate wiederholt und siehe da, dann waren die Ergebnisse wieder eindeutig (siehe Abbildung 2.48). Ein Performancetest zwischen den einzelnen Rechnern bestätigte im Nachgang diese Vermutung. In Abbildung 2.49 sehen Sie, dass der Schul-PC definitiv die beste Grafikverarbeitungsgeschwindigkeit hat.

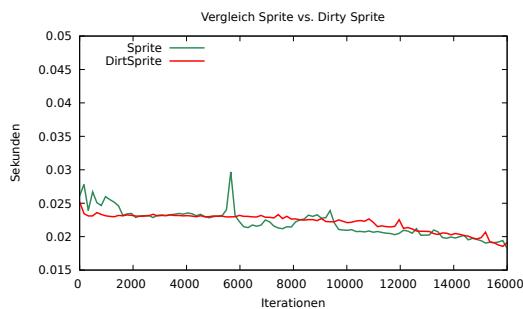


Abbildung 2.46: Testkonfiguration mit privatem PC

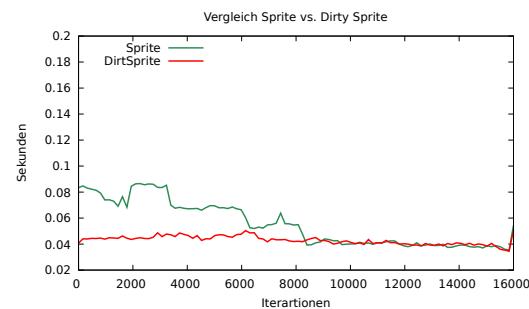


Abbildung 2.47: Testkonfiguration mit Surface Pro

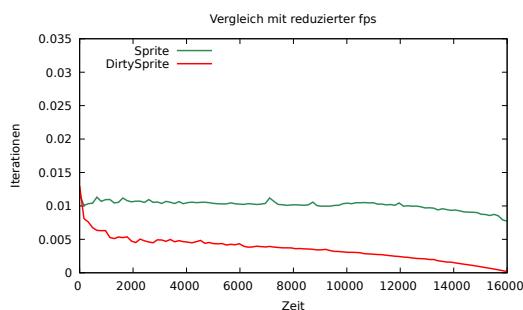


Abbildung 2.48: Testkonfiguration mit privatem PC und reduzierter fps

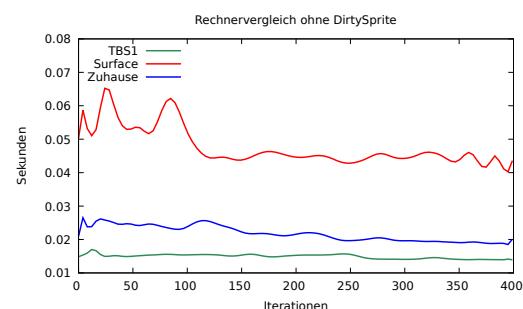


Abbildung 2.49: Bestätigung der Unterschiede

Es ist somit eine ernst zu nehmende Aufgabe, die maximale Framerate auf dem Rechner zu ermitteln. Nur dann können die Mechanismen des DirtySprite-Konzepts greifen. Auch ist nicht auszuschließen, dass die Rechner deshalb kein schönes ruckelfreies Bewegungsbild erzeugen, wenn die Framerate für den Rechner zu hoch eingestellt ist.

2.14.3 Fazit

Die obigen Messungen und Probleme bzgl. sich bewegender Sprites haben mich persönlich nicht von dieser DirtySprite-Implementierung überzeugt. Ein einfacher Ball, der an den Wänden abprallt (Quelltext 2.113), hinterlässt Artefakte oder flackert recht heftig, was ich nicht erwartet hätte.

Quelltext 2.113: Einfacher Ball mit DirtsSprite

```

15  class Ball(pygame.sprite.DirtySprite):
16
17      def __init__(self) -> None:
18          super().__init__()
19          self.image = pygame.surface.Surface((30, 30)).convert()
20          self.image.set_colorkey((0, 0, 0))
21          pygame.draw.circle(self.image, "blue", (15, 15), 15)
22          pygame.draw.rect(self.image, "red", (0, 0, 30, 30), 1)
23          self.rect = pygame.rect.FRect(self.image.get_rect())
24          self.speed = pygame.Vector2(-10, 5)
25          self.dirty = 1
26
27      def update(self) -> None:
28          self.rect.move_ip(self.speed)
29          if self.rect.left < 0 or self.rect.right > Settings.WINDOW.width:
30              self.speed.x *= -1
31          if self.rect.top < 0 or self.rect.bottom > Settings.WINDOW.height:
32              self.speed.y *= -1
33          self.dirty = 1
34
35  class Game(object):
36
37      def __init__(self) -> None:
38          pygame.init()
39          self.screen = pygame.display.set_mode(Settings.WINDOW.size)
40          self.background = pygame.surface.Surface(Settings.WINDOW.size)
41          self.background.fill("white")
42          pygame.display.set_caption("Ball mit DirtySprite")
43          self.clock = pygame.time.Clock()
44          self.ball = pygame.sprite.LayeredDirty(Ball())
45          self.ball.clear(self.screen, self.background)
46          self.ball.set_timing_threshold(1000.0/Settings.FPS)
47          self.running = True
48
49      def run(self) -> None:
50          time_previous = time()
51          while self.running:
52              self.watch_for_events()
53              self.update()
54              self.draw()
55              self.clock.tick(Settings.FPS)
56              time_current = time()
57              Settings.DELTATIME = time_current - time_previous
58              time_previous = time_current
59          pygame.quit()
60
61      def watch_for_events(self) -> None:
62          for event in pygame.event.get():
63              if event.type == pygame.QUIT:
64                  self.running = False
65
66      def update(self) -> None:
67          self.ball.update()
68

```

```
69
70     def draw(self) -> None:
71         rects = self.ball.draw(self.screen)
72         pygame.display.update(rects)
```

Falls jemand einen Programmierfehler im Beispiel findet, wäre ich um eine Korrektur sehr dankbar. In einem Pygame-Tutorial⁷ wird die Verwendung auch nicht wirklich empfohlen:

In the present day (2022) though, most modest desktop computers are powerful enough to refresh the entire display once per frame at 60 FPS and beyond. You can have a moving camera, or dynamic backgrounds and your game should run totally fine at 60 FPS. CPUs are more powerful nowadays, and you can use `display.flip()` without fear.

Ich werde deshalb das Thema in dieser Einführung nicht weiter vertiefen.

Was war neu?

Mit Hilfe von entsprechenden Klassen, kann die Zeichenausgabe pro Frame auf die Bereiche eingeschränkt werden, die sich tatsächlich verändert haben. Der Performanceverbrauch reduziert sich dabei erheblich.

Es muss allerdings darauf geachtet werden, dass die Framerate der Leistungsfähigkeit der Rechnerkonfiguration angepasst wird.

Es wurden folgende Pygame-Elemente eingeführt:

- Vordefinierte Farbnamen:
https://pyga.me/docs/ref/color_list.html
- `pygame.display.update()`:
<https://pyga.me/docs/ref/sprite.html#pygame.display.update>
- `pygame.sprite.DirtySprite`:
<https://pyga.me/docs/ref/sprite.html#pygame.sprite.DirtySprite>
- `pygame.sprite.DirtySprite.dirty`:
<https://pyga.me/docs/ref/sprite.html#pygame.sprite.DirtySprite>
Bedeutung siehe Tabelle 2.9 auf der nächsten Seite.
- `pygame.sprite.LayeredDirty`:
<https://pyga.me/docs/ref/sprite.html#pygame.sprite.LayeredDirty>
- `pygame.sprite.LayeredDirty.clear()`:
<https://pyga.me/docs/ref/sprite.html#pygame.sprite.LayeredDirty.clear>
- `pygame.sprite.LayeredDirty.draw()`:
<https://pyga.me/docs/ref/sprite.html#pygame.sprite.LayeredDirty.draw>

⁷ <https://pyga.me/docs/tutorials/en/newbie-guide.html>

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

Tabelle 2.9: Bedeutung von `dirty`

Konstante	Beschreibung
0	Der Sprite ist noch aktuell und muss nicht neu gezeichnet werden.
1	(Default) Der Sprite ist veraltet (Aussehen oder Position haben sich verändert) und muss neu gezeichnet werden. Nach dem Neuzeichnen wird <code>dirty</code> automatisch wieder auf 0 gesetzt.
2	Der Sprite wird immer neu gezeichnet. Sie wird nicht nach dem Zeichnen auf 0 gesetzt.

2.15 Events

Wir haben Ereignisse (Event) schon an zwei Stellen verwendet, ohne sie näher betrachtet zu haben. Zum einen als wir in Kapitel 2.6 auf Seite 54 über die Tastatur und zum anderen als wir in Kapitel 2.12 auf Seite 98 über die Maus gesprochen haben.

Wir werden hier drei Aspekte näher beleuchten:

- Welche Infos stecken eigentlich in einem Event?
- Wie kann ich selbst ein Event erzeugen?
- Wie kann ich periodisch Ereignisse erzeugen lassen?

2.15.1 Welche Infos stecken in einem Event?

Das Programm zu Quelltext 2.114 erstellt lediglich ein graues Fenster und gibt mit `print` in Zeile 40 das Event in der Konsole aus.

Quelltext 2.114: Events – Informationen ausgeben

```

38     def watch_for_events(self) -> None:
39         for event in pygame.event.get():
40             print(event)                                     # Eventinfo ausgeben
41             if event.type == QUIT:
42                 self._running = False
43             elif event.type == KEYDOWN:
44                 if event.key == K_ESCAPE:
45                     self._running = False

```

Wandert man nun mit der Maus hin und her, drückt ein paar Tasten oder beendet die Anwendung, erscheint ungefähr sowas in der Konsole, wobei ich viele redundante Zeilen gelöscht habe:

Quelltext 2.115: Events – Konsolenausgabe

```

1 <Event(769-KeyUp {'unicode': 'd', 'key': 100, 'mod': 4096, 'scancode': 7, 'window': None})>
2 <Event(768-KeyDown {'unicode': 'a', 'key': 97, 'mod': 4096, 'scancode': 4, 'window': None})>
3 <Event(771-TextInput {'text': 'a', 'window': None})>
4 <Event(768-KeyDown {'unicode': 'u', 'key': 32, 'mod': 4096, 'scancode': 44, 'window': None})>
5 <Event(771-TextInput {'text': 'u', 'window': None})>
6 <Event(1024-MouseMotion {'pos': (299, 143), 'rel': (-1, 0), 'buttons': (0, 0, 0), 'touch': False, 'window': None})>
7 <Event(1024-MouseMotion {'pos': (297, 143), 'rel': (-2, 0), 'buttons': (0, 0, 0), 'touch': False, 'window': None})>
8 <Event(1025-MouseButtonDown {'pos': (230, 118), 'button': 1, 'touch': False, 'window': None})>
9 <Event(1026-MouseButtonUp {'pos': (230, 118), 'button': 1, 'touch': False, 'window': None})>
10 <Event(1027-MouseWheel {'flipped': False, 'x': 0, 'y': 1, 'precise_x': 0.0, 'precise_y': 1.0, 'touch': False, 'window': None})>
11 <Event(1025-MouseButtonDown {'pos': (230, 118), 'button': 5, 'touch': False, 'window': None})>
12 <Event(1026-MouseButtonUp {'pos': (230, 118), 'button': 5, 'touch': False, 'window': None})>
13 <Event(1027-MouseWheel {'flipped': False, 'x': 0, 'y': -1, 'precise_x': 0.0, 'precise_y': -1.0, 'touch': False, 'window': None})>

```

```

14 <Event(1024-MouseMotion {'pos': (572, 0), 'rel': (3, -1), 'buttons': (0, 0, 0), 'touch': False, 'window': None})>
15 <Event(32768-ActiveEvent {'gain': 0, 'state': 1})>
16 <Event(32784-WindowLeave {'window': None})>
17 <Event(32787-WindowClose {'window': None})>
18 <Event(256-Quit {})>

```

Zunächst fällt auf, dass die Eventinformationen in Form eines Dictionarys zur Verfügung gestellt werden. Den ersten Eintrag (die Nummer mit dem Bindestrich und anschließendem Namen) kann über `event.type` abgefragt werden. Damit man sich diese Nummern nicht auswendig merken muss, werden von Pygame entsprechende Konstanten angeboten; für die Tastatur finden Sie in Tabelle 2.5 auf Seite 57 und für die Maus in Tabelle 2.7 auf Seite 102 eine Übersicht.

In den geschweiften Klammern stehen nun die Key/Value-Paare – also die dem Event mitgegebenen Informationen. Bei Tastaturereignissen sind dies beispielsweise die Darstellung als Unicode-Zeichen oder seine Unicodenummer. Mausereignissen wird sinnvollerweise die Position und die Tastennummer mitgegeben. Das Klicken auf dem *Fensterschließen*-Button oben rechts löst mehrere Ereignisse aus, hier die letzten vier der Liste.

Wir werden gleich sehen, dass bei selbsterstellten Ereignissen, diese Infos nach eigenem Bedarf definiert werden können.

2.15.2 Wie kann ich selbst ein Event erzeugen?

Als Beispiel will ich hier zwei primitive Buttons verwenden, die jeweils beim linken Mausklick ein Event erzeugen sollen. Innerhalb des Bildschirms flitzen NOFSTARTPARTICLES viele Partikel durch die Gegend. Mit den Buttons **Start** und **Stop** sollen die Partikel anhalten bzw. wieder flitzen.

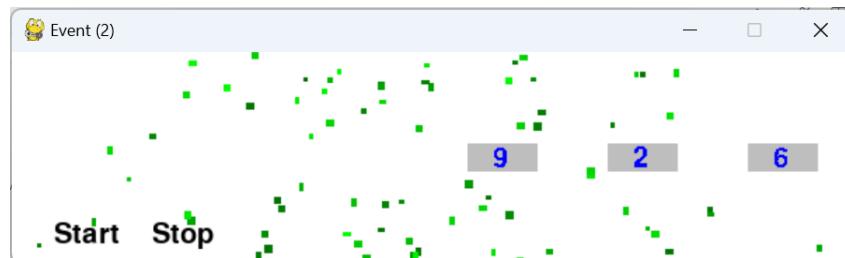


Abbildung 2.50: Selbst erstellte Events

Als weiteres Feature ist so eine Art Zählwerk implementiert. Die Kästchen in der Mitte absorbieren die Partikel und zählen Sie dabei. Die Logik ist wie folgt: Jedes mal, wenn ein Partikel eine Box trifft, wird ein Zählereignis ausgelöst. Dabei wird immer in der Box ganz rechts eine 1 aufaddiert.

Hat die ganz rechte Box den Wert 10 erreicht, erzeugt er einen Überlauf auf die nächste Ziffer links von ihm und setzt sich wieder auf 0; dies setzt sich von rechts nach links

fort. Somit wird in den Boxen die Gesamtanzahl von Partikeln angezeigt, die schon verschluckt wurden.

Das Ganze nun im Detail. In der Konsolenausgabe oben (siehe Seite 127) ist für jedes Event so eine eindeutige Nummer zu sehen, über die man das Event identifizieren kann. Pygame reserviert mir einen Nummernbereich für eigene Events zwischen den Konstanten `pygame.USEREVENT` und `pygame.NUMEVENTS - 1`. Für jedes selbst erstellte Event muss man nun eine solche eindeutige Nummer vergeben. Am einfachsten ist es, zentral diese durch `USEREVENT + n` zu definieren. In Zeile 11 und Zeile 12 finden Sie entsprechende Beispiele. Ich kapsle diese Definitionen in eine statische Klasse aus keinem anderen Grund, als dass ich dann die Autovervollständigung des Editors gut nutzen kann (Zeile 10). Die Klasse `Settings` sollte selbsterklärend sein.

USEREVENT
NUMEVENTS

Quelltext 2.116: Events (2) – Präambel

```

1 import os
2 from random import choice, randint
3 from time import time
4 from typing import Any, Tuple
5
6 import pygame
7 from pygame.constants import K_ESCAPE, KEYDOWN, MOUSEBUTTONDOWN, QUIT
8
9
10 class MyEvents:
11     BUTTONPRESSED = pygame.USEREVENT + 0           # Nur wegen Autovervollständigung
12     OVERFLOW = pygame.USEREVENT + 1                 # EventID für die Buttons
13
14
15 class Settings:
16     WINDOW = pygame.rect.Rect((0, 0), (600, 150))
17     FPS = 60
18     DELTATIME = 1.0/FPS
19     STARTNOFPARTICLES = 999
20     NOFBOXES = 3
21     BOXWIDTH = 50

```

Die Klasse `Button` sollte auch über weite Strecken verstanden werden. Die erste spannende Stelle finden Sie in Zeile 37. Hier wird ein neues `pygame.event.Event`-Objekt erzeugt. Als ersten Parameter muss diese eben erwähnte ID angegeben werden. Danach können Sie beliebig viele Angaben als Eventinfo mitgeben. In unserem Beispiel wird der Button-Text mitgegeben, damit man später feststellen kann, welcher Button gedrückt wurde.

Event

Danach wird in Zeile 38 über `pygame.event.post()` das Event abgefeuert.

post()

Quelltext 2.117: Events (2) – Klasse Button

```

24 class Button(pygame.sprite.Sprite):
25
26     def __init__(self, text: str, position: Tuple[int], *groups: Tuple[pygame.sprite.Group])
27         -> None:
28         super().__init__(*groups)
29         self.font = pygame.font.SysFont(None, 30)
30         self.centerxy = (Settings.WINDOW.centerx, self.font.get_height()//2)

```

```

30         self._text = text
31         self.image = self.font.render(self._text, True, "black")
32         self.rect = self.image.get_rect(topleft=(position))
33
34     def update(self, *args: Any, **kwargs: Any) -> None:
35         if "action" in kwargs.keys():
36             if kwargs["action"] == "pressed":
37                 evt = pygame.event.Event(MyEvents.BUTTONPRESSED, text=self._text) #
38                 pygame.event.post(evt) #
39
40     return super().update(*args, **kwargs)

```

Die Klasse `Particle` ist viel Quelltext mit wenig Neuem. Partikel zufälliger Größe, Farbe, Richtung und Geschwindigkeit sausen durch den Bildschirm und prallen ggf. von den Rändern ab. Sie enthalten keine event-spezifischen Funktionalitäten. Das Attribut `_halted` wird verwendet, um nach Drücken der Buttons den Partikel anzuhalten bzw. wieder loslaufen zu lassen.

Quelltext 2.118: Events (2) – Klasse Particle

```

42 class Particle(pygame.sprite.Sprite):
43
44     def __init__(self, *groups: Tuple[pygame.sprite.Group]) -> None:
45         super().__init__(*groups)
46         self.image = pygame.surface.Surface((randint(3, 6), randint(3, 6)))
47         self.image.fill((0, randint(100, 255), 0))
48         self.rect = pygame.rect.FRect(self.image.get_rect())
49         self.rect.topleft = (randint(30, Settings.WINDOW.right-30), randint(30,
50             Settings.WINDOW.bottom-30), )
51         self._speed = randint(100, 400)
52         self._direction = pygame.Vector2(choice((-1, 1)), choice((-1, 1)))
53         self._halted = False
54
55     def update(self, *args: Any, **kwargs: Any) -> None:
56         if "action" in kwargs.keys():
57             if kwargs["action"] == "move":
58                 if not self._halted:
59                     self._move()
60             elif kwargs["action"] == "Start":
61                 self._halted = False
62             elif kwargs["action"] == "Stop":
63                 self._halted = True
64
65     def _move(self) -> None:
66         self.rect.move_ip(self._speed*self._direction*Settings.DELTATIME)
67         if self.rect.left < 0:
68             self._direction[0] *= -1
69             self.rect.left = 0
70         if self.rect.right > Settings.WINDOW.right:
71             self._direction[0] *= -1
72             self.rect.right = Settings.WINDOW.right
73         if self.rect.top < 0:
74             self._direction[1] *= -1
75             self.rect.top = 0
76         if self.rect.bottom > Settings.WINDOW.bottom:
77             self._direction[1] *= -1
78             self.rect.bottom = Settings.WINDOW.bottom

```

Mit `Box` wird so eine Ziffernbox implementiert. Dem Konstruktor wird dabei eine Position und ein Index mitgegeben. Die Bedeutung des Paramters `position` sollte klar sein. Mit Hilfe von `index` kann später ermittelt werden, welche Box einen Überlauf zur nächst

höheren Zehnerpotenz hatte.

In `update()` wird der internen Zähler `_count` immer um 1 erhöht. Erreiche ich dabei die 10 (Zeile 95), wird das Event erzeugt und der Index als Eventinfo übergeben. Damit kann das Hauptprogramm ermitteln, welcher Box er nun ein `update()` verpassen muss.

Quelltext 2.119: Events (2) – Klasse Box

```

80  class Box(pygame.sprite.Sprite):
81
82      def __init__(self, index: int, position: Tuple[int], *groups:
83          Tuple[pygame.sprite.Group]) -> None:
84          super().__init__(*groups)
85          self.image = pygame.surface.Surface((Settings.BOXWIDTH, 20))
86          self.rect = self.image.get_rect(center=position)
87          self._font = pygame.font.SysFont(None, 30)
88          self._counter = 0
89          self._index = index
90          self._fill()
91
92      def update(self, *args: Any, **kwargs: Any) -> None:
93          if "counter" in kwargs.keys():
94              if kwargs["counter"] == "inc":
95                  self._counter += 1
96                  if self._counter == 10:                      # Überlauf
97                      evt = pygame.event.Event(MyEvents.OVERFLOW, index=self._index)
98                      pygame.event.post(evt)
99                      self._counter = 0
100                     self._fill()
101
102     def _fill(self) -> None:
103         self.image.fill("gray")
104         number = self._font.render(f"{self._counter}", False, "Blue")
105         self.image.blit(number, (18, 1))

```

Und jetzt das Hauptprogramm: Im Konstruktor werden die Buttons, Boxen und Partikel angelegt und Spritegroups zugeordnet.

Quelltext 2.120: Events (2) – Konstruktor von Game

```

108  class Game:
109
110      def __init__(self) -> None:
111          pygame.init()
112          self._clock = pygame.time.Clock()
113          self._screen = pygame.display.set_mode(Settings.WINDOW.size)
114          pygame.display.set_caption("Event(2)")
115          self._all_sprites = pygame.sprite.Group()
116          self._all_particles = pygame.sprite.Group()
117          self._generate_particles(Settings.STARTNOFPARTICLES)
118          self._all_buttons = pygame.sprite.Group()
119          self._all_buttons.add(Button("Start", (30, Settings.WINDOW.bottom - 30),
120                                      self._all_sprites))
121          self._all_buttons.add(Button("Stop", (100, Settings.WINDOW.bottom - 30),
122                                      self._all_sprites))
123          self._all_boxes = pygame.sprite.Group()
124          self._generate_boxes(Settings.NOFBOXES)
125          self._running = True

```

Die Methode `run()` ist nahezu langweilig.

Quelltext 2.121: Events (2) – Game.run()

```

125  def run(self) -> None:
126      time_previous = time()
127      while self._running:
128          self.watch_for_events()
129          self.update()
130          self.draw()
131          self._clock.tick(Settings.FPS)
132          time_current = time()
133          Settings.DELTATIME = time_current - time_previous
134          time_previous = time_current
135      pygame.quit()

```

Benutzerdefinierte Events werden genauso behandelt wie vordefiniert. Zuerst erfragt man den `type` und dann verarbeitet man die Eventinfos. In Zeile 147 wird nachgeschaut, ob einer der beiden Buttons gedrückt wird. Anschließend wird über die Eventinfo `text` an die Partikel die Nachricht weitergeleitet, ob sie stehenbleiben oder weiterlaufen sollen. Analoges ab Zeile 149. Zuerst wird überprüft, ob eine Box einen Überlauf hatte und dann wird mit Hilfe des Eventinfo `index` die nächste Box darüber informiert, dass sie sich um 1 erhöhen muss.

Quelltext 2.122: Events (2) – Game.watch_for_events()

```

137  def watch_for_events(self) -> None:
138      for event in pygame.event.get():
139          if event.type == QUIT:
140              self._running = False
141          elif event.type == KEYDOWN:
142              if event.key == K_ESCAPE:
143                  self._running = False
144          elif event.type == MOUSEBUTTONDOWN:
145              if event.button == 1:
146                  self._check_button_pressed(event.pos)
147          elif event.type == MyEvents.BUTTONPRESSED:      #
148              self._all_particles.update(action=event.text)
149          elif event.type == MyEvents.OVERFLOW:          #
150              if event.index < Settings.NOFBOXES-1:
151                  self._all_boxes.sprites()[event.index+1].update(counter="inc")

```

Der Rest wird hier der Vollständigkeit abgedruckt.

Quelltext 2.123: Events (2) – Der Rest von Game

```

153  def update(self):
154      self._all_buttons.update()
155      self._all_particles.update(action="move")
156      self._check_boxcollision()
157
158  def draw(self) -> None:
159      self._screen.fill("white")
160      self._all_sprites.draw(self._screen)
161      pygame.display.update()
162
163  def _generate_boxes(self, number: int) -> None:
164      for i in range(number):
165          self._all_boxes.add(Box(i, (Settings.WINDOW.right - 50 - i * 100,
166                               Settings.WINDOW.centery), self._all_sprites))
166

```

```

167     def _generate_particles(self, number: int) -> None:
168         for i in range(number):
169             self._all_particles.add(Particle(self._all_sprites))
170
171     def _check_button_pressed(self, position: Tuple[int]) -> None:
172         for b in self._all_buttons.sprites():
173             if b.rect.collidepoint(position):
174                 b.update(action="pressed")
175
176     def _check_boxcollision(self) -> None:
177         for p in self._all_particles.sprites():
178             for b in self._all_boxes.sprites():
179                 if p.rect.colliderect(b):
180                     self._all_boxes.sprites()[0].update(counter="inc")
181                     p.kill()

```

2.15.3 Wie kann ich periodisch Ereignisse erzeugen lassen?

Dies ist sogar recht einfach. Das vorherige Beispiel wird so erweitert, dass im Abstand von 500 ms immer neue Partikel erstellt werden.

Dazu wird zunächst die neue ID NEWPARTICLES für das Benutzerevent definiert.

Quelltext 2.124: Events (3) – Präambel

```

10 class MyEvents:
11     BUTTONPRESSED = pygame.USEREVENT + 0
12     OVERFLOW = pygame.USEREVENT + 1
13     NEWPARTICLES = pygame.USEREVENT + 2

```

Im Konstruktor von `Game` wird in Zeile 139 mit Hilfe von `pygame.time.set_timer()` ein periodischer Timer dazu gesetzt. Dieser schießt alle 500 ms die entsprechende EventID ab.

`set_timer()`

Quelltext 2.125: Events (3) – Timer

```

138     self._generate_boxes(Settings.NOFBOXES)
139     pygame.time.set_timer(MyEvents.NEWPARTICLES, 500)      # Periodischer Timer
140     self._running = True

```

Wie die anderen Events wird dieses nun in `watch_for_event()` abgefangen (Zeile 169) und verarbeitet. Hier indem die Methode `_generate_particles()` aufgerufen wird.

Quelltext 2.126: Events (3) – Event

```

169     elif event.type == MyEvents.NEWPARTICLES:      # Periodisches Event
170         self._generate_particles(Settings.NEWNOPARTICLES)

```

Was war neu?

Der Vorteil von benutzerdefinierten Ereignissen wird hier gut deutlich. Wollte man dies anders implementieren, müssen die Objekte von einander wissen. Alle Boxen müssten

ihren Vorgänger oder Nachfolger beispielsweise als Referenz kennen, um den Überlauf bekannt zu geben. Dies kann auch eine gute Methode sein, durch ein Event werden die Klassen aber entkoppelt und das Hauptprogramm kann die Informationsweiterleitung durch die Eventinfo gesteuert organisierten.

Besonders das Klicken auf die Buttons können durch die Events einfach implementiert werden.

Es wurden folgende Pygame-Elemente eingeführt:

- **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.post()** :
<https://pyga.me/docs/ref/event.html#pygame.event.post>
- **pygame.time.set_timer()** :
https://pyga.me/docs/ref/time.html#pygame.time.set_timer

3 Beispielprojekte

3.1 Pong

Der Anfängerklassiker überhaupt. Seit 1972 wird dieses Spiel in immer wieder neuen Varianten gespielt. Da die Regeln recht einfach sind, eignet es sich gut als Anfängerprojekt.

Wir werden dieses Spiel systematisch Schritt für Schritt entwickeln, wobei ich davon ausgehen werde, dass die Techniken aus Kapitel 2 bekannt sind. Ich werde auf Docstring-Kommentare im Quelltext verzichten, da hier im Text alles erklärt wird und die Listings sich dadurch unnötig verlängern. In der finalen Version sind sie eingetragen.

Hinweis: Ich habe mir mal zu Beginn per [ChatGPT](#) ein Pong-Spiel erzeugen lassen. Das war schon beeindruckend zu sehen, dass da ein funktionierendes Spiel erstellt wurde.

3.1.1 Requirement 1: Standards

Requirement 1 Standardfunktionalität

1. Fenster hat eine angemessene Größe.
 2. Hintergrund ist eine dunkelrote Spielfläche mit einer gestrichelten Mittellinie.
 3. Beendet wird mit der ESC-Taste oder per Mausklick auf rotes „X“.
 4. Das Spiel hat eine von der FPS unabhängige Ablaufgeschwindigkeit.
-

Und los geht's. Hier jetzt einmalig die Präambel. Ich gehe davon aus, dass Sie genügend Pythonkenntnisse besitzen, um diese jeweils zu erweitern.

Quelltext 3.1: Pong (Requirement 1) – Präambel und die Klasse `Settings`

```
1 from time import time
2 from typing import Any, Tuple
3
4 import pygame
5
6
7 class Settings:
8     WINDOW = pygame.Rect(0, 0, 1000, 600)
9     FPS = 60
10    DELTATIME = 1.0 / FPS
```

Der Background wird hier nicht als Bitmap geladen, sondern erzeugt. Es gibt dafür keinen besonderen Grund, außer zu demonstrieren, dass Bitmaps auch dynamisch erstellt werden können. Dafür wird als erstes ein Surface-Objekt in der Größe des Bildschirms erstellt. Dann wird es dunkelrot ausgefüllt, was einen Sandplatz simulieren soll. In `_paint_net()` ab Zeile 21 wird das Netz als eine Folge von weißen Rechtecken gemalt.



Abb. 3.1: Pong: der Hintergrund

Quelltext 3.2: Pong (Requirement 1) – die Klasse Background

```

13  class Background(pygame.sprite.Sprite):
14      def __init__(self, *groups: Tuple[pygame.sprite.Group]) -> None:
15          super().__init__(*groups)
16          self.image = pygame.surface.Surface(Settings.WINDOW.size).convert()
17          self.rect = self.image.get_rect()
18          self.image.fill("darkred")
19          self._paint_net()
20
21      def _paint_net(self) -> None: # Tennisnetz
22          net_rect = pygame.rect.Rect(0, 0, 0, 0)
23          net_rect.centerx = Settings.WINDOW.centerx
24          net_rect.top = 50
25          net_rect.size = (3, 30)
26          while net_rect.bottom < Settings.WINDOW.bottom:
27              pygame.draw.rect(self.image, "grey", net_rect, 0)
28              net_rect.move_ip(0, 40)

```

Die Klasse `Game` besteht aus den Grundelementen, die wir in Kapitel 2 schon gesehen haben. In `__init__()` wird Pygame gestartet, das Display und der Taktgeber erstellt und das Flag der Hauptprogrammschleife initialisiert. Der Hintergrund wird in einem `GroupSingle`-Objekt abgelegt. Die restlichen Methoden sollten selbsterklärend sein.

Quelltext 3.3: Pong (Requirement 1) – die Klasse Game

```

31  class Game:
32      def __init__(self):
33          pygame.init()
34          self._display = pygame.display.set_mode(Settings.WINDOW.size)
35          pygame.display.set_caption("My Kind of Pong")
36          self._clock = pygame.time.Clock()
37          self._background = pygame.sprite.GroupSingle(Background())
38          self._running = True
39
40      def run(self):
41          time_previous = time()
42          while self._running:
43              self.watch_for_events()
44              self.update()
45              self.draw()
46              self._clock.tick(Settings.FPS)
47              time_current = time()
48              Settings.DELTATIME = time_current - time_previous
49              time_previous = time_current
50          pygame.quit()

```

```

51
52     def update(self):
53         pass
54
55     def draw(self):
56         self._background.draw(self._display)
57         pygame.display.update()
58
59     def watch_for_events(self):
60         for event in pygame.event.get():
61             if event.type == pygame.QUIT:
62                 self._running = False
63             elif event.type == pygame.KEYDOWN:
64                 if event.key == pygame.K_ESCAPE:
65                     self._running = False

```

Der Vollständigkeit halber:

Quelltext 3.4: Pong (Requirement 1) – die Klasse Game

```

68 def main():
69     game = Game()
70     game.run()
71
72
73 if __name__ == "__main__":
74     main()

```

Die Anwendung ist derzeit noch funktionslos, stellt mir aber schon den Hintergrund dar (siehe Abbildung 3.1 auf der vorherigen Seite).

3.1.2 Requirement 2: Die Schläger

Requirement 2 Schläger

1. Auf der linken und der rechten Seite befindet sich jeweils ein rechteckiger Schläger.
 2. Die Schläger haben ein Breite von 15 px und ein Höhe von einem Zehntel der Bildschirmhöhe.
 3. Die Schläger haben eine Geschwindigkeit von $\frac{\text{Bildschirmhöhe}}{2}$ px/s.
 4. Die Schläger haben jeweils vom Mittelpunkt einen Abstand von 50 px zum linken bzw. rechten Rand.
 5. Der linke Schläger wird über die Taste  nach oben und mit der Taste  nach unten bewegt.
 6. Der rechte Schläger wird über die Taste  nach oben und mit der Taste  nach unten bewegt.
 7. Die Schläger können das Spielfeld nicht verlassen.
-

In Zeile 37 wird die Größe des Schlägers ermittelt (Requirements 2.1 und 2.2). Ab Zeile 38 wird die Position der Schläger festgelegt. Die vertikale Startposition ist dabei immer die Bildschirmmitte. Die horizontale Startposition ist davon abhängig, ob es der rechte oder linke Schläger ist. Beide werden entsprechend Requirement 2.4 etwas vom Rand abgesetzt.

Die Geschwindigkeit wird entsprechend Requirement 2.3 in Zeile 44 bestimmt. Auch dieses Bitmap wird nicht geladen, sondern selbst erstellt (Zeile 46) und gelb eingefärbt.

Quelltext 3.5: Pong (Requirement 2) – Der Konstruktor von Paddle

```

31  class Paddle(pygame.sprite.Sprite):
32      BORDERDISTANCE = {"horizontal": 50, "vertical": 10}
33      DIRECTION = {"up": -1, "down": 1, "halt": 0}
34
35      def __init__(self, player: str, *groups: Tuple[pygame.sprite.Group]) -> None:
36          super().__init__(*groups)
37          self.rect = pygame.rect.FRect(0, 0, 15, Settings.WINDOW.height // 10) # Größe
38          self.rect.centery = Settings.WINDOW.centery # Position
39          self._player = player
40          if self._player == "left":
41              self.rect.left = Paddle.BORDERDISTANCE["horizontal"]
42          else:
43              self.rect.right = Settings.WINDOW.right - Paddle.BORDERDISTANCE["horizontal"]
44          self._speed = Settings.WINDOW.height // 2 # Geschwindigkeit
45          self._direction = Paddle.DIRECTION["halt"] # Steht erstmal still
46          self.image = pygame.surface.Surface(self.rect.size) # Surface
47          self.image.fill("yellow")

```

Die Methode `update()` verteilt die Aufgaben. Dabei wird bzgl. der Bewegung das Attribut `self._direction` entsprechend manipuliert (ab Zeile 53). Soll der Schläger seine Position verändern, wird in Zeile 51 die Methode `_move()` aufgerufen.

Quelltext 3.6: Pong (Requirement 2) – Paddle.update()

```

49  def update(self, *args: Any, **kwargs: Any) -> None:
50      if "action" in kwargs.keys():
51          if kwargs["action"] == "move": # Positionsänderung
52              self._move()
53          elif kwargs["action"] in Paddle.DIRECTION.keys(): # Bewegungsrichtung
54              self._direction = Paddle.DIRECTION[kwargs["action"]]
55      return super().update(*args, **kwargs)

```

Verbleibt noch die Methode `_move()`. Sie sieht komplizierter aus, als sie ist. Nachdem überprüft wurde, ob überhaupt irgendwas getan werden muss, wird in Zeile 59 die neue vertikale Position berechnet (die horizontale bleibt ja unverändert). Anschließend wird überprüft, ob der Schläger das Spielfeld verlassen hat. Falls *Ja*, wird der Schläger an den oberen bzw. unteren Rand zurückversetzt.

Quelltext 3.7: Pong (Requirement 2) – Paddle._move()

```

57  def _move(self) -> None:
58      if self._direction != Paddle.DIRECTION["halt"]:
59          self.rect.move_ip(0, self._speed * self._direction * Settings.DELTATIME) #
60          if self._direction == Paddle.DIRECTION["up"]:

```

```

61         self.rect.top = max(self.rect.top, Paddle.BORDERDISTANCE["vertical"])
62     elif self._direction == Paddle.DIRECTION["down"]:
63         self.rect.bottom = min(self.rect.bottom, Settings.WINDOW.height -
64             Paddle.BORDERDISTANCE["vertical"])

```

Nun müssen die Schläger in `Game` eingepflegt werden. In Zeile 74 wird zunächst eine `Spritegroup` erstellt, welche alle Sprites außer dem Hintergrund aufnehmen wird. Danach werden die beiden Schläger erzeugt und per Übergabeparameter gleich der `Spritegroup` hinzugefügt.

Quelltext 3.8: Pong (Requirement 2) – Konstruktor von `Game`

```

66 class Game:
67     def __init__(self):
68         pygame.init()
69         self._display = pygame.display.set_mode(Settings.WINDOW.size)
70         pygame.display.set_caption("My Kind of Pong")
71         self._clock = pygame.time.Clock()
72         self._background = pygame.sprite.GroupSingle(Background())
73         self._all_sprites = pygame.sprite.Group()
74         self._paddle = {} # Schläger
75         self._paddle["left"] = Paddle("left", self._all_sprites)
76         self._paddle["right"] = Paddle("right", self._all_sprites)
77         self._running = True

```

In `update()` und `draw()` erfolgen lediglich der entsprechende Methodenaufruf der `Spritegroup`.

Quelltext 3.9: Pong (Requirement 2) – `Game.update()` und `Game.draw()`

```

91     def update(self):
92         self._all_sprites.update(action="move") # Bewegung
93
94     def draw(self):
95         self._background.draw(self._display)
96         self._all_sprites.draw(self._display) # Ausgabe
97         pygame.display.update()

```

Und jetzt werden die Tastaturevents verarbeitet. Das Drücken einer Taste löst eine Bewegung aus (ab Zeile 106) und das Loslassen führt zu einem Anhalten des entsprechenden Schlägers (ab Zeile 114).

Dabei wird die Methode `Paddle.update()` immer mit einem passenden Parameter aufgerufen; bei Bewegungen mit `action="up"` oder `action="down"` und zum Anhalten mit `action="halt"`.

Quelltext 3.10: Pong (Requirement 2) – `Game.watch_for_events()`

```

99     def watch_for_events(self):
100         for event in pygame.event.get():
101             if event.type == pygame.QUIT:
102                 self._running = False
103             elif event.type == pygame.KEYDOWN:
104                 if event.key == pygame.K_ESCAPE:
105                     self._running = False

```

```

106         elif event.key == pygame.K_UP: # Schlägerbewegung
107             self._paddle["right"].update(action="up")
108         elif event.key == pygame.K_DOWN:
109             self._paddle["right"].update(action="down")
110         elif event.key == pygame.K_w:
111             self._paddle["left"].update(action="up")
112         elif event.key == pygame.K_s:
113             self._paddle["left"].update(action="down")
114     elif event.type == pygame.KEYUP: # Schlägerstopp
115         if event.key == pygame.K_UP or event.key == pygame.K_DOWN:
116             self._paddle["right"].update(action="halt")
117         elif event.key == pygame.K_w or event.key == pygame.K_s:
118             self._paddle["left"].update(action="halt")

```

3.1.3 Requirement 3: Der Ball

Requirement 3 Ball

1. Der Ball ist ein Kreis mit einem Radius von 10 px.
2. Seine Geschwindigkeit beträgt $\frac{\text{Bildschirmbreite}}{3}$ px/s.
3. Er startet in der Bildschirmmitte und hat eine zufällige horizontale und vertikale Richtung.
4. Am oberen und unteren Bildschirmrand prallt er ab.
5. Berührt er den linken Rand, startet er in der Mitte neu. Analoges passiert, wenn er den rechten Rand berührt.
6. Wird der rechte Rand berührt, erhält Spieler 1 einen Punkt und beim Linken der Spieler 2.

Da wir laut Requirement 3.6 den Punktestand der Spieler brauchen, wird in `Settings` ein entsprechendes Array angelegt (Zeile 12).

Quelltext 3.11: Pong (Requirement 3) – `Settings`

```

8 class Settings:
9     WINDOW = pygame.Rect(0, 0, 1000, 600)
10    FPS = 60
11    DELTATIME = 1.0 / FPS
12    POINTS = [0, 0] # Punktestand

```

Passend zu Requirement 3.1 und 3.2 werden in Zeile 71 und Zeile 75 die Größe und die Geschwindigkeit festgelegt. Der Start des Balls erfolgt häufiger und wird daher in die Methode `_service()` (Zeile 77) ausgelagert.

Quelltext 3.12: Pong (Requirement 3) – Konstruktor von `Ball`

```

68 class Ball(pygame.sprite.Sprite):
69     def __init__(self, *groups: Tuple[pygame.sprite.Group]) -> None:
70         super().__init__(*groups)

```

```

71     self.rect = pygame.Rect(0, 0, 20, 20) # Größe
72     self.image = pygame.surface.Surface(self.rect.size).convert()
73     self.image.set_colorkey("black")
74     pygame.draw.circle(self.image, "green", self.rect.center, self.rect.width // 2)
75     self._speed = Settings.WINDOW.width // 3 # Geschwindigkeit
76     self._speedxy = pygame.Vector2()
77     self._service() # Aufschlag

```

In `update()` werden die Aufgaben verteilt.

Quelltext 3.13: Pong (Requirement 3) – `Ball.update()`

```

79 def update(self, *args: Any, **kwargs: Any) -> None:
80     if "action" in kwargs.keys():
81         if kwargs["action"] == "move":
82             self._move()
83         elif kwargs["action"] == "hflip":
84             self._horizontal_flip()
85         elif kwargs["action"] == "vflip":
86             self._vertical_flip()
87         elif kwargs["action"] == "service":
88             self._service()
89     return super().update(*args, **kwargs)

```

Schauen wir uns jetzt die Hilfsfunktionen im einzelnen an. Beginnen wir mit `_move()`. Wie zu erwarten, werden die Positionsangabe mit Hilfe der Geschwindigkeiten aktualisiert. Danach wird ab Zeile 93 überprüft, ob der Ball eine der vier Ränder erreicht hat.

Wird der obere oder untere Rand erreicht (Requirement 3.4), wechselt das Vorzeichen der vertikalen Geschwindigkeit durch den Aufruf von `_vertical_flip()` (Quelltext 3.16 auf der nächsten Seite). Nach dem Flip wird der obere bzw. der untere Rand gesetzt, da es ja sein kann, dass der Ball die Randgrenze schon überschritten hat.

Anders, wenn der Ball den rechten oder linken Rand erreicht. Dann soll nach Requirement 3.5 der Ball neu aufgeschlagen werden (siehe Quelltext 3.15 auf der nächsten Seite) und wie in Requirement 3.6 gefordert wird der entsprechende Punktestand angepasst.

Quelltext 3.14: Pong (Requirement 3) – `Ball._move()`

```

91 def _move(self) -> None:
92     self.rect.move_ip(self._speedxy * Settings.DELTATIME)
93     if self.rect.top <= 0: # Rändercheck
94         self._vertical_flip()
95         self.rect.top = 0
96     elif self.rect.bottom >= Settings.WINDOW.bottom:
97         self._vertical_flip()
98         self.rect.bottom = Settings.WINDOW.bottom
99     elif self.rect.right < 0:
100         Settings.POINTS[1] += 1
101         self._service()
102     elif self.rect.left > Settings.WINDOW.right:
103         Settings.POINTS[0] += 1
104         self._service()

```

Beim Aufschlag wird das Zentrum des Balls auf das Zentrum des Bildschirms gesetzt (Requirement 3.3). Danach werden für die beiden Richtungsgeschwindigkeiten per Zufall

die Vorzeichen und damit die Bewegungsrichtung (nach links oder rechts bzw. nach oben oder unten) bestimmt. Da wir noch keine Punkteausgabe haben, ist in Zeile 109 eine provisorische Ausgabe auf der Konsole programmiert.

Quelltext 3.15: Pong (Requirement 3) – Ball._service()

```
106     def _service(self) -> None:
107         self.rect.center = Settings.WINDOW.center
108         self._speedxy = pygame.Vector2(choice([-1, 1]), choice([-1, 1])) * self._speed
109         print(Settings.POINTS) # Provisorium
```

Der Richtungswechsel ist hier nur ein Vorzeichenwechsel. Die Methode `_flip_horizontal()` wird noch nicht verwendet, wird aber gebraucht, wenn wir den Ball vom Schläger abprallen lassen wollen.

Quelltext 3.16: Pong (Requirement 3) – Die Flip-Methoden von Ball

```
111     def _horizontal_flip(self) -> None:
112         self._speedxy.x *= -1
113
114     def _vertical_flip(self) -> None:
115         self._speedxy.y *= -1
```

3.1.4 Requirement 4: Punkte

Requirement 4 Punkte

1. *Der Punktestand wird mittig oben dargestellt.*
-

Zur Darstellung verwende ich die Klasse `Score`. Letztlich ist sie auch nur ein Sprite, welches allerdings von Zeit zu Zeit neu gebildet werden muss, nämlich jedes mal, wenn der Punktestand sich ändert. Da der Punktestand nun in Zeile 128 abgelegt wird, kann er aus `Settings` entfernt werden.

Quelltext 3.17: Pong (Requirement 4) – Konstruktor von Score

```
123 class Score(pygame.sprite.Sprite):
124
125     def __init__(self, *groups: Tuple[pygame.sprite.Group]):
126         super().__init__(*groups)
127         self._font = pygame.font.SysFont(None, 30)
128         self._score = {1: 0, 2: 0} # Nicht mehr in Settings!
129         self.image: pygame.surface.Surface = None
130         self.rect: pygame.rect.Rect = None
131         self._render()
```

In dieser Methode wird der aktuelle Punktestand mit Hilfe des Font-Objektes gerendert und positioniert.

Quelltext 3.18: Pong (Requirement 4) – Score._render()

```
139 def _render(self):
140     self.image = self._font.render(f"{self._score[1]}\u00b7{self._score[2]}", True, "white")
141     self.rect = self.image.get_rect(centerx=Settings.WINDOW.centerx, top=15)
```

In `update()` wird der passende Punktestand aktualisiert und `_render()` aufgerufen.

Quelltext 3.19: Pong (Requirement 4) – Score.update()

```
133 def update(self, *args: Any, **kwargs: Any) -> None:
134     if "player" in kwargs.keys():
135         self._score[kwargs["player"]] += 1
136         self._render()
137     return super().update(*args, **kwargs)
```

Was jetzt noch fehlt, ist das Anstoßen einer Punktestandausgabe. Dies ist eine gute Gelegenheit für ein Userevent. Ab Zeile 14 wird alles für ein Userevent benötigte implementiert. Zuerst ein eine Event-ID und dann das passende `pygame.event.Event`-Objekt.

Quelltext 3.20: Pong (Requirement 4) – MyEvent

```
14 class MyEvents: # Userevent
15     POINT_FOR = pygame.USEREVENT
16     MYEVENT = pygame.event.Event(POINT_FOR, player=0)
```

Nun muss `Ball` nur noch das passende Event auslösen und `Game` das Event verwalten. Hier die Anpassungen in `Ball`. In der Methode `_move()` werden die entsprechenden Stellen ersetzt. So wird beispielsweise in Zeile 104 die Nummer des Spielers in das Event gestopft, der den Punkt bekommt, und in Zeile 105 wird das Event abgeschickt.

Quelltext 3.21: Pong (Requirement 4) – Ball._move()

```
95 def _move(self) -> None:
96     self.rect.move_ip(self._speedxy * Settings.DELTATIME)
97     if self.rect.top <= 0:
98         self._vertical_flip()
99         self.rect.top = 0
100    elif self.rect.bottom >= Settings.WINDOW.bottom:
101        self._vertical_flip()
102        self.rect.bottom = Settings.WINDOW.bottom
103    elif self.rect.right < 0:
104        MyEvents.MYEVENT.player = 2 # Spielernummer
105        pygame.event.post(MyEvents.MYEVENT) # Abfeuern
106        self._service()
107    elif self.rect.left > Settings.WINDOW.right:
108        MyEvents.MYEVENT.player = 1
109        pygame.event.post(MyEvents.MYEVENT)
110        self._service()
```

Jetzt muss nur noch in `watch_for_events()` das Userevent abgegriffen werden (ab Zeile 199).

Quelltext 3.22: Pong (Requirement 4) – `Ball.watch_for_events()`

```

179  def watch_for_events(self):
180      for event in pygame.event.get():
181          if event.type == pygame.QUIT:
182              self._running = False
183          elif event.type == pygame.KEYDOWN:
184              if event.key == pygame.K_ESCAPE:
185                  self._running = False
186              elif event.key == pygame.K_UP:
187                  self._paddle["right"].update(action="up")
188              elif event.key == pygame.K_DOWN:
189                  self._paddle["right"].update(action="down")
190              elif event.key == pygame.K_w:
191                  self._paddle["left"].update(action="up")
192              elif event.key == pygame.K_s:
193                  self._paddle["left"].update(action="down")
194          elif event.type == pygame.KEYUP:
195              if event.key == pygame.K_UP or event.key == pygame.K_DOWN:
196                  self._paddle["right"].update(action="halt")
197              elif event.key == pygame.K_w or event.key == pygame.K_s:
198                  self._paddle["left"].update(action="halt")
199          elif event.type == MyEvents.POINT_FOR: # Userevent
200              self._score.update(player=event.player)

```

3.1.5 Requirement 5: Tennisschlag

Das Ergebnis sieht eigentlich fertig aus und kann aber so noch nicht gespielt werden, da die Schläger immer noch nutzlos sind.

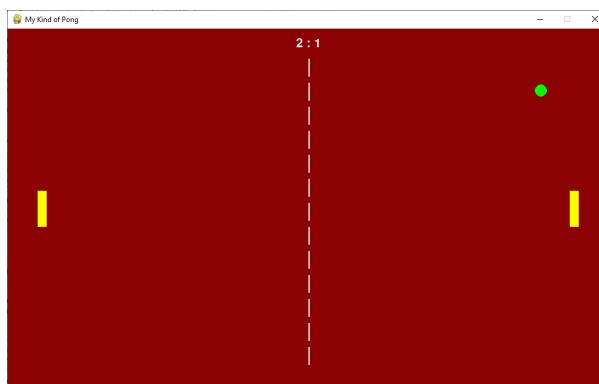


Abbildung 3.2: Pong: mit Schläger, Ball und Punktstand

Requirement 5 Punkte

1. *Berührt der Ball den Schläger, so prallt er von ihm ab und wird in das gegnerische Feld zurückgespielt.*
2. *Bei jeder Schlägerberührung werden die Richtungsgeschwindigkeiten per Zufall um einen kleinen Betrag erhöht.*

In Game bauen wir dazu die Methode `_check_collision()`, welche überprüft, ob der Ball einen Schläger getroffen hat. Es bietet sich an, dazu die Methode `pygame.sprite.collide_rect()` zu verwenden. Wenn eine Kollision vorliegt, wird die bisher noch nicht verwendete Methode `_horizontal_flip()` (siehe Quelltext 3.16 auf Seite 142) über `update()` ausgeführt. Danach werden die Ränder wieder so verschoben, dass Ball und Schläger sich nicht überlappen. Ebenso wird über `update()` die Methode `_respeed()` aufgerufen, so dass Requirement 5.2 erfüllt wird.

Quelltext 3.23: Pong (Requirement 5) – Game. `_check_collision()`

```
208     def _check_collision(self):
209         if pygame.sprite.collide_rect(self._ball, self._paddle["left"]):
210             self._ball.update(action="hflip")
211             self._ball.rect.left = self._paddle["left"].rect.right + 1
212         elif pygame.sprite.collide_rect(self._ball, self._paddle["right"]):
213             self._ball.update(action="hflip")
214             self._ball.rect.right = self._paddle["right"].rect.left - 1
```

In `_respeed()` werden den Geschwindigkeitsvektoren jeweils um Zufallswerte erhöht. Über `_speed` ist diese Schwankung indirekt von der Bildschirmgröße abhängig.

Quelltext 3.24: Pong (Requirement 5) – Ball. `_respeed()`

```
123     def _respeed(self) -> None:
124         self._speedxy.x += randrange(0, self._speed // 4)
125         self._speedxy.y += randrange(0, self._speed // 4)
```

3.1.6 Requirement 6: Computerspieler

Eigentlich wären wir jetzt fertig, aber ich möchte noch einen Computerspieler einbauen. Dadurch kann das Spiel auch gegen den Computer gespielt werden bzw. man den Computer stundenlang gegen sich selbst spielen lassen.

Requirement 6 Punkte

1. Über die Taste **[1]** wechselt die Steuerung vom linken Schläger zwischen Mensch und Computer.
2. Über die Taste **[2]** wechselt die Steuerung vom rechten Schläger zwischen Mensch und Computer.
3. Wird die Steuerung wieder auf manuell gestellt, soll der Schläger erstmal stehen bleiben.

In `Settings` habe ich in Zeile 12 ein Dictionary von Flags angelegt, welches mir für jeden Spieler steuert, ob er per Hand oder per Computer gespielt werden soll.

Quelltext 3.25: Pong (Requirement 6) – `Settings`

```
8 class Settings:
9     WINDOW = pygame.rect.Rect(0, 0, 1000, 600)
10    FPS = 60
11    DELTATIME = 1.0 / FPS
12    KI = {"left": False, "right": False} # Computerspielerflags
```

In der Methode `update()` wird ab Zeile 197 mit Hilfe der Flags überprüft, ob der Schläger vom Computer gespielt wird und wenn *Ja*, wird eine Controller-Methode aufgerufen.

Quelltext 3.26: Pong (Requirement 6) – `Game.update()`

```
195 def update(self):
196     self._check_collision()
197     for i in Settings.KI.keys(): # Computerbefehl
198         if Settings.KI[i]:
199             self._paddlecontroller(self._paddle[i])
200     self._all_sprites.update(action="move")
```

Schauen wir uns nun die Controller-Methode an. Die Grundidee ist, dass der Schläger solange nach oben wandert, wie die Ballmitte oberhalb der Schlägermitte liegt, bzw. nach unten, solange die Ballmitte unterhalb der Schlägermitte. Dabei muss nicht bis nach ganz oben oder unten gewandert werden, die letzten Pixel kann man sich sparen, da dann ggf. schon eine Kollision ausgelöst wird.

Quelltext 3.27: Pong (Requirement 6) – `Game._paddlecontroller()`

```
252 def _paddlecontroller(self, paddle: pygame.sprite.Sprite) -> None:
253     if paddle.rect.centery > self._ball.rect.centery and paddle.rect.top > 10:
254         paddle.update(action="up")
255     elif paddle.rect.centery < self._ball.rect.centery and paddle.rect.bottom <
256         Settings.WINDOW.bottom - 10:
257         paddle.update(action="down")
258     else:
259         paddle.update(action="halt")
```

In `watch_for_events()` sind umfangreiche Umbauten notwendig. Zunächst muss die manuelle Steuerung für die Schläger unterbunden werden, wenn die auf Computerspieler

stehen. Dazu wird vor Aufruf der entsprechenden `update()`-Methode zuerst gefragt, ob nicht der Computerspieler die Kontrolle hat. Ein Beispiel finden Sie in Zeile 215.

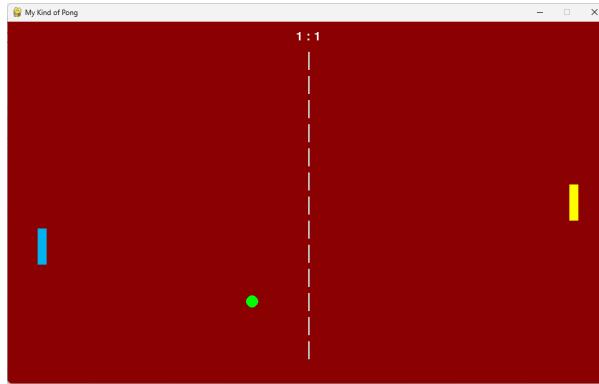


Abbildung 3.3: Pong: Schlägerfarbe markiert KI-Modus (links KI, rechts manuell)

Ein verbleibender Punkt ist noch Requirement 6.3. Dazu wird wie in Zeile 228 das entsprechende Flag abgefragt und dem Schläger das Halt-Signal gesendet.

Quelltext 3.28: Pong (Requirement 6) – `Game.watch_for_events()`

```

207     def watch_for_events(self):
208         for event in pygame.event.get():
209             if event.type == pygame.QUIT:
210                 self._running = False
211             elif event.type == pygame.KEYDOWN:
212                 if event.key == pygame.K_ESCAPE:
213                     self._running = False
214                 elif event.key == pygame.K_UP:
215                     if not Settings.KI["right"]:
216                         self._paddle["right"].update(action="up")
217                     elif event.key == pygame.K_DOWN:
218                         if not Settings.KI["right"]:
219                             self._paddle["right"].update(action="down")
220                     elif event.key == pygame.K_w:
221                         if not Settings.KI["left"]:
222                             self._paddle["left"].update(action="up")
223                     elif event.key == pygame.K_s:
224                         if not Settings.KI["left"]:
225                             self._paddle["left"].update(action="down")
226                     elif event.key == pygame.K_1:
227                         Settings.KI["left"] = not Settings.KI["left"]
228                         if not Settings.KI["left"]:
229                             self._paddle["left"].update(action="halt")
230                     elif event.key == pygame.K_2:
231                         Settings.KI["right"] = not Settings.KI["right"]
232                         if not Settings.KI["right"]:
233                             self._paddle["right"].update(action="halt")
234             elif event.type == pygame.KEYUP:
235                 if event.key == pygame.K_UP or event.key == pygame.K_DOWN:
236                     if not Settings.KI["right"]:
237                         self._paddle["right"].update(action="halt")
238                     elif event.key == pygame.K_w or event.key == pygame.K_s:
239                         if not Settings.KI["left"]:
240                             self._paddle["left"].update(action="halt")
241             elif event.type == MyEvents.POINT_FOR:

```

3.1.7 Requirement 7: Sound

Ein wenig Sound könnte das Spiel noch aufpeppen.

Requirement 7 Punkte

1. *Der Schlag mit dem Tennisschläger soll mit einem passendem Sound untermauert werden.*
2. *Das Abprallen vom oberen und unterem Rand soll mit einem passendem Sound untermauert werden.*
3. *Der Sound soll über  ein- bzw. ausgeschaltet werden können.*

Als ersten Schritt erweitern wir `Settings` um das Flag `SOUNDFLAG` in Zeile 14, welches steuert, ob der Sound abgespielt werden soll oder nicht und Zugriffe auf das Soundfile.

Quelltext 3.29: Pong (Requirement 7) – `Settings`

```

14  SOUNDFLAG = True  # Soundflags
15  PATH = {}
16  PATH["file"] = os.path.dirname(os.path.abspath(__file__))
17  PATH["sound"] = os.path.join(PATH["file"], "sounds")
18
19  @staticmethod
20  def get_sound(filename: str) -> str:
21      return os.path.join(Settings.PATH["sound"], filename)

```

Die eigentliche Soundausgabe wird in `Ball` implementiert. Im Konstruktor werden ab Zeile 103 die Geräusche geladen und der Kanal ermittelt, über welchen der Sound abgespielt werden soll.

Quelltext 3.30: Pong (Requirement 7) – Konstruktor von `Ball`

```

103  self._sounds: dict[str, pygame.mixer.Sound] = {}  # Geräusche speichern
104  self._sounds["left"] = pygame.mixer.Sound(Settings.get_sound("player1.mp3"))
105  self._sounds["right"] = pygame.mixer.Sound(Settings.get_sound("player2.mp3"))
106  self._sounds["bounce"] = pygame.mixer.Sound(Settings.get_sound("bounce.mp3"))
107  self._channel = pygame.mixer.find_channel()

```

Den ersten Sound programmieren wir für das Abprallen am Schläger in `_horizontal_flip()`. Nachdem abgefragt wurde, ob überhaupt eine Soundausgabe erfolgen soll, wird ermittelt, ob der Ball vom rechten oder vom linken Schläger abprallt. Dies geschieht indirekt durch die Abfrage, in welche Richtung der Ball aktuell fliegt (Zeile 151). Passend dazu wird die Lautstärke so angepasst, dass der Eindruck entsteht, dass der Abprall links bzw. rechts vom Zuschauer erfolgt.

Quelltext 3.31: Pong (Requirement 7) – Ball._horizontal_flip()

```

149  def _horizontal_flip(self) -> None:
150      if Settings.SOUNDFLAG:
151          if self._speedxy.x < 0: # Flugrichtung nach links?
152              self._channel.set_volume(0.9, 0.1)
153              self._channel.play(self._sounds["left"])
154          else:
155              self._channel.set_volume(0.1, 0.9)
156              self._channel.play(self._sounds["right"])
157          self._speedxy.x *= -1
158          self._respeed()

```

Etwas dynamischer wird dieser Soundeffekt in `_vertical_flip()` erzeugt. In Zeile 162 wird die relative horizontale Position ermittelt. Ist das Zentrum des Balls links, hat `rel_pos` einen Wert nahe der 0; steht der Ball weit rechts, hat er einen Wert nahe 1. Diese Werte können dann als rechte und linke Lautstärke in die Methode `set_volume()` eingesetzt werden.

Quelltext 3.32: Pong (Requirement 7) – Ball._vertical_flip()

```

160  def _vertical_flip(self) -> None:
161      if Settings.SOUNDFLAG:
162          rel_pos = self.rect.centerx / Settings.WINDOW.width # Wo bin ich?
163          self._channel.set_volume(1.0 - rel_pos, rel_pos)
164          self._channel.play(self._sounds["bounce"])
165          self._speedxy.y *= -1

```

Verbleibt noch das Ein- bzw. Ausschalten der Soundausgabe in `watch_for_events()` in Zeile 246 mit Hilfe der Funktionstaste .

Quelltext 3.33: Pong (Requirement 7) – Ball.watch_for_events()

```

232  def watch_for_events(self):
233      for event in pygame.event.get():
234          if event.type == pygame.QUIT:
235              self._running = False
236          elif event.type == pygame.KEYDOWN:
237              if event.key == pygame.K_ESCAPE:
238                  self._running = False
239              elif event.key == pygame.K_UP:
240                  if not Settings.KI["right"]:
241                      self._paddle["right"].update(action="up")
242              elif event.key == pygame.K_DOWN:
243                  if not Settings.KI["right"]:
244                      self._paddle["right"].update(action="down")
245              elif event.key == pygame.K_F2:
246                  Settings.SOUNDFLAG = not Settings.SOUNDFLAG # Toogle Soundflag

```

3.1.8 Requirement 8: Pause und Hilfebildschirm

Requirement 8 Punkte

1. Durch  werden alle Aktivitäten gestoppt und das Spiel pausiert. Wird nochmal  gedrückt, wird das Spiel fortgesetzt.
2. Durch  wird das Spiel pausiert und ein Hilfetext angezeigt. Wird nochmal  gedrückt, wird das Spiel fortgesetzt.

Für die Pause bauen wir uns – vielleicht etwas überdimensioniert – eine eigene Klasse. Das Wesentliche ist die Zeile 52. Dort wird über das Surface-Objekt der gleichen Größe wie der Bildschirm ein Grauschleier gelegt, indem man das Objekt mit grauer Farbe auffüllt. Diese Farbe hat aber im Alpha-Kanal den Wert 200, so dass der Hintergrund durchschimmert.

Quelltext 3.34: Pong (Requirement 8) – Pause

```

47 class Pause(pygame.sprite.Sprite):
48     def __init__(self, *groups: Tuple[pygame.sprite.Group]) -> None:
49         super().__init__(*groups)
50         self.rect = pygame.Rect(Settings.WINDOW.topleft, Settings.WINDOW.size)
51         self.image = pygame.surface.Surface(self.rect.size).convert_alpha()
52         self.image.fill([120, 120, 120, 200]) # transparentes Grau

```

Analog gehen wir für den Hilfsbildschirm vor. Nur wird hier noch ein Text auf dem Surface-Objekt geblättert. Der Text ist in die linke und rechte Spalte aufgeteilt, um ihn besser lesen zu können.

Quelltext 3.35: Pong (Requirement 8) – Help

```

55 class Help(pygame.sprite.Sprite):
56     def __init__(self, *groups: Tuple[pygame.sprite.Group]) -> None:
57         super().__init__(*groups)
58         self.rect = pygame.Rect(Settings.WINDOW.topleft, Settings.WINDOW.size)
59         self.image = pygame.surface.Surface(self.rect.size).convert_alpha()
60         self.image.fill([20, 20, 20, 200]) # transparentes Grau
61         font = pygame.font.Font(pygame.font.get_default_font(), 20)
62         text_l = "h\np\nESC\nF2\nnk\nn1\nnr\nnUP\nnDOWN\nnw\nns"
63         text_r = "-_toogle_modus\n-_toogle_pause_modus\n-_quit\nn\n-_toogle_sound_modus\n"
64         text_r += "-_toogle_both_paddles_KI_modus\n-_toogle_left_paddle_KI_modus\n-_toogle_right_paddle_KI_modus\nn\n"
65         text_r += "-_left_paddle_move_up\n-_left_paddle_move_down\n-_right_paddle_move_up\n-_right_paddle_move_down"
66         lines = font.render(text_l, True, "white")
67         self.image.blit(lines, (10, 10))
68         lines = font.render(text_r, True, "white")
69         self.image.blit(lines, (10 + 70, 10))

```

Im Konstruktor von Game müssen nun die beiden Flags angelegt werden, die den jeweiligen Modus repräsentieren (Zeile 232 und Zeile 233). Anschließend werden die beiden Darstellungen angelegt und einem pygame.Group.Single-Objekt zugewiesen.

Quelltext 3.36: Pong (Requirement 8) – Help

```

218 class Game:
219     def __init__(self):
220         pygame.init()
221         self._display = pygame.display.set_mode(Settings.WINDOW.size)
222         pygame.display.set_caption("My Kind of Pong")
223         self._clock = pygame.time.Clock()
224         self._background = pygame.sprite.GroupSingle(Background())
225         self._all_sprites = pygame.sprite.Group()
226         self._paddle = {}
227         self._paddle["left"] = Paddle("left", self._all_sprites)
228         self._paddle["right"] = Paddle("right", self._all_sprites)
229         self._ball = Ball(self._all_sprites)
230         self._score = Score(self._all_sprites)
231         self._running = True
232         self._pausing = False # Pause Flag
233         self._helping = False # Hilfe Flag
234         self._pause = pygame.sprite.GroupSingle(Pause())
235         self._help = pygame.sprite.GroupSingle(Help())

```

Nachdem nun alles vorbereitet ist, wird die Methode `update()` so gestaltet, dass nur dann neue Zustände berechnet werden, wenn keine der beiden Modi aktiv ist (Zeile 250).

Quelltext 3.37: Pong (Requirement 8) – Game.update()

```

249     def update(self):
250         if not (self._pausing or self._helping): # Nur bei Normalbetrieb
251             self._check_collision()
252             for i in Settings.KI.keys():
253                 if Settings.KI[i]:
254                     self._paddlecontroller(self._paddle[i])
255             self._all_sprites.update(action="move")

```

In `draw()` wird ebenfalls auf die jeweiligen Modi abgefragt und ggf. das entsprechende Sprite ausgegeben.

Quelltext 3.38: Pong (Requirement 8) – Game.draw()

```

257     def draw(self):
258         self._background.draw(self._display)
259         self._all_sprites.draw(self._display)
260         if self._pausing: # Pausedarstellung
261             self._pause.draw(self._display)
262         elif self._helping: # Hilfedarstellung
263             self._help.draw(self._display)
264         pygame.display.update()

```

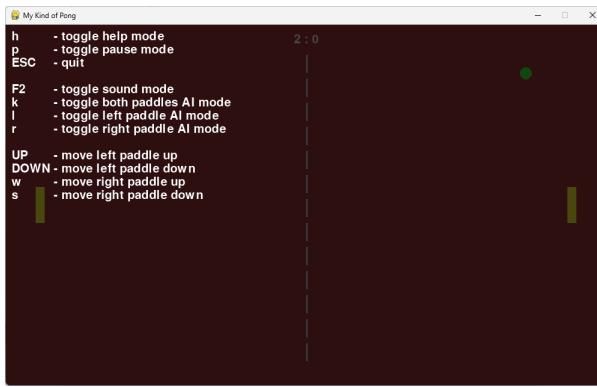


Abbildung 3.4: Pong: Hilfe-Bildschirm

3.2 Bubbles

In diesem Kapitel wird das Spiel *Bubbles* beispielhaft besprochen. Ich möchte gleich darauf hinweisen, dass die Spielidee nicht von mir stammt. Ein Schüler hat es mal als Handy-Version auf einer ITA-Messe vorgestellt. Leider kann ich mich nicht mehr an den Namen erinnern, aber auf diesem Wege ein herzliches *Dankeschön*.

Wir werden dieses Spiel systematisch Schritt für Schritt entwickeln, wobei ich davon ausgehen werde, dass die Techniken in Kapitel 2 bekannt sind. Ich werde auf Docstring-Kommentare im Quelltext verzichten, da hier im Text alles erklärt wird und die Listings sich dadurch unnötig verlängern. In der finalen Version sind sie eingetragen.

Das Spiel lässt sich beliebig erweitern: Animation des Zerplatzens, Highscoreslisten usw., aber wie so oft ist das Bessere der Feind des Guten. Ich wünsche viel Spaß beim Studium.

3.2.1 Requirement 1: Standards

Requirement 1 Standardfunktionalität

1. Fenster hat eine angemessene Größe.
2. Hintergrund ist eine passende Bitmap oder einfarbig.
3. Beendet wird mit der *ESC*-Taste oder per Mausklick auf rote „X“.
4. Alle Bitmaps werden als `pygame.sprite.DirtySprite`-Objekt erzeugt und nach dem Laden konvertiert und passend skaliert.
5. Alle Bitmaps – außer dem Hintergrund – sind transparent.
6. Alle Bitmaps – bis auf „Einzelkämpfer“ oder Hintergrund – werden in `pygame.sprite.LayeredGroup`-Objekten abgelegt.
7. Das Spiel hat eine von der *fps* unabhängige Ablaufgeschwindigkeit.

Requirement 1 auf der vorherigen Seite regelt nicht nur Konkretes, sondern auch Allgemeines und wird deshalb bei späteren Implementierungen noch einmal auftauchen. Hier erst das, was wir sofort umsetzen können.

Hier jetzt einmalig die Präambel. Ich gehe davon aus, dass Sie genügend Pythonkenntnisse besitzen, um diese jeweils zu erweitern. Die statischen Angaben zum Spiel werden hier wie gewohnt in einer separaten Klasse `Settings` abgelegt.

Es wird gefordert, dass das Fenster eine angemessene Größe hat. Mit $1220 \text{ px} \times 1002 \text{ px}$ bin ich groß genug, um die Blasen zu verteilen und klein genug, um mit der Maus noch schnell wandern zu können. Der Rest ist in den vorherigen Kapiteln schon ausführlich behandelt worden (z.B. FPS, DELTATIME oder PATH) und wird deshalb hier nicht weiter erläutert.



Abbildung 3.5: Bubbles: Hintergrundbild (aquarium.png)

Quelltext 3.39: Bubbles (Requirement 1.1) – Präambel

```

1 import os
2 from time import time
3 from typing import Dict
4
5 import pygame
6 from pygame.constants import K_ESCAPE, KEYDOWN, QUIT
7
8
9 class Settings:
10     WINDOW = pygame.rect.Rect(0, 0, 1220, 1002)
11     FPS = 60
12     DELTATIME = 1.0/FPS
13     PATH: Dict[str, str] = {}
14     PATH["file"] = os.path.dirname(os.path.abspath(__file__))
15     PATH["image"] = os.path.join(PATH["file"], "images")
16     PATH["sound"] = os.path.join(PATH["file"], "sounds")
17     CAPTION = 'Fingerübung "Bubbles"'
18
19     @staticmethod
20     def get_file(filename: str) -> str:
21         return os.path.join(Settings.PATH["file"], filename)
22
23     @staticmethod

```

```

24     def get_image(filename: str) -> str:
25         return os.path.join(Settings.PATH["image"], filename)
26
27     @staticmethod
28     def get_sound(filename: str) -> str:
29         return os.path.join(Settings.PATH["sound"], filename)

```

Die Klasse `Background` ist eine Kindklasse von `DirtySprite`, welches nur geladen und passend skaliert wird. Weil sich der Hintergrund nicht ändert, muss kein `update()` implementiert werden. Dass wir eine eigene Kindklasse programmieren, ist ein wenig wie mit Pistolen auf Spatzen schießen. Wir hätten es auch als `DirtySprite`-Objekt implementieren können. Ich habe das hier nur der Übersichtlichkeit wegen gemacht.

Interessant ist Zeile 41. Hier wird nicht ein `self.screen` verwendet, sondern der aktuelle Bildschirm wird aus Pygame heraus mit `pygame.display.get_surface()` ermittelt. Das Hintergrundbild ist in Abbildung 3.5 auf der vorherigen Seite zu sehen.

Quelltext 3.40: Bubbles (Requirement 1.2) – Background

```

31 class Background(pygame.sprite.DirtySprite):
32     def __init__(self) -> None:
33         super().__init__()
34         imagename = Settings.get_image("aquarium.png")
35         self.image: pygame.surface.Surface = pygame.image.load(imagename).convert()
36         self.image = pygame.transform.scale(self.image, Settings.WINDOW.size)
37         self.rect: pygame.rect.Rect = self.image.get_rect()
38         self.dirty = 1
39
40     def draw(self):
41         pygame.display.get_surface().blit(self.image, self.rect) # Holt sich screen

```

In der Klasse `Game` werden in `__init__()` die Pygame üblichen Methoden `init()`, `set_mode()`, `clock()` und `set_caption()` am Start aufgerufen. Auch wird das Flag der Hauptprogrammschleife erzeugt. Die Methoden `run()`, `watch_for_events()`, `update()` und `draw()` enthalten nur Basisfunktionalitäten, die hier nicht weiter erläutert werden müssen.

Quelltext 3.41: Bubbles (Requirement 1) – Methoden von `Game`

```

44 class Game:
45
46     def __init__(self) -> None:
47         pygame.init()
48         self._screen = pygame.display.set_mode(Settings.WINDOW.size)
49         pygame.display.set_caption(Settings.CAPTION)
50         self._clock = pygame.time.Clock()
51         self._background = Background()
52         self._running = True
53
54     def watch_for_events(self) -> None:
55         for event in pygame.event.get():
56             if event.type == QUIT:
57                 self._running = False
58             elif event.type == KEYDOWN:
59                 if event.key == K_ESCAPE:
60                     self._running = False

```

```

62     def draw(self) -> None:
63         self._background.draw()
64         pygame.display.update()
65
66     def update(self) -> None:
67         pass
68
69     def run(self) -> None:
70         time_previous = time()
71         self._running = True
72         while self._running:
73             self.watch_for_events()
74             self.update()
75             self.draw()
76             self._clock.tick(Settings.FPS)
77             time_current = time()
78             Settings.DELTATIME = time_current - time_previous
79             time_previous = time_current
80         pygame.quit()

```

Durch diese Methoden ist aber schon der generelle Ablauf des Spiels vorgegeben. Alle weiteren Eigenschaften des Spiels, sind nur noch Erweiterungen dieses Ablaufs, keine Veränderungen mehr.

Zum Schluss erfolgt der Aufruf (siehe Quelltext 3.42). Damit sind alle Unterpunkte von Requirement 1 auf Seite 152, die hier Anwendung finden, erfüllt.

Quelltext 3.42: Bubbles (Requirement 1) – Aufruf

```

83 def main():
84     os.environ["SDL_VIDEO_WINDOW_POS"] = "10,130"
85     game = Game()
86     game.run()
87
88
89 if __name__ == "__main__":
90     main()

```

3.2.2 Requirement 2: Blasen erscheinen

Requirement 2 Blasen erscheinen

1. An zufälliger Position erscheint ein Blase.
2. Zu Beginn erscheint diese jede halbe Sekunde.
3. Sie hat einen Startradius von 15 px.
4. Sie hat zu den Rändern einen Abstand von mindestens 10 px.
5. Sie hat zu allen anderen Blasen einen Mindestabstand von 10 px.

Für die Blase wird die schon transparente Grafik aus Abbildung 3.6 verwendet. Die zufällige Position muss noch eingeschränkt werden.



Das Aquarium füllt ja nicht den ganzen Bildschirm aus (siehe Abbildung 3.5 auf Seite 153), sondern steht innerhalb einer Art Fernseher. Wir müssen also eine Spielfläche (*playground*) definieren. Nur innerhalb dieser Spielfläche sollen die Blasen erscheinen.

Die Spielfläche ist ein Rechteck mit einem Abstand zum linken und oberen Bildschirmrand – `left` und `top` – und einer Breite (`width`) und Höhe (`height`). In Zeile 21 werden die entsprechenden Werte festgehalten. Der Abstand von Spielfeldrand und der Blasen untereinander wird in Zeile 20 entsprechend Requirement 2.4 mit 10 *px* definiert. Der Startradius – und damit der minimale Radius – wird wegen Requirement 2.3 in Zeile 19 mit 15 *px* festgelegt. Während des Spielens ist mir aufgefallen, dass kleinere Startradien einfach zu schlecht gesehen werden.

Quelltext 3.43: Bubbles (Requirement 2) – Ergänzungen in `Settings`

```
19 RADIUS = {"min": 15}                                     # Radius Startwert
20 DISTANCE = 50                                         # Rand-/Blasenabstand
21 PLAYGROUND = pygame.rect.Rect(90, 90, 1055, 615)      # Rechteck im Aquarium
```

Die Klasse `Timer` ist exakt die oben in Kapitel 2.10 auf Seite 83 beschriebene; dort wird alles erklärt.

Quelltext 3.44: Bubbles (Requirement 2) – `Timer`

```
36 class Timer:
37     def __init__(self, duration: int, with_start: bool = True) -> None:
38         self.duration = duration
39         if with_start:
40             self._next = pygame.time.get_ticks()
41         else:
42             self._next = pygame.time.get_ticks() + self.duration
43
44     def is_next_stop_reached(self) -> bool:
45         if pygame.time.get_ticks() > self._next:
46             self._next = pygame.time.get_ticks() + self.duration
47             return True
48         return False
```

Schauen wir uns jetzt die Klasse `Bubble` an. Der Konstruktor ist selbsterklärend, hier werden nur die üblichen Verdächtigen bearbeitet: `image`, `rect` und `radius`. Die Methode `update()` ist derzeit leer, da noch keine Veränderung verlangt wurde. Die Methode `randompos()` wird allerdings wegen Requirement 2.1 benötigt. Sie berechnet eine neues Blasenzentrum und weist dieses `rect` zu. Ggf. muss diese Methode solange wiederholt werden, bis eine freie Fläche gefunden wird (siehe Requirement 2.4 und Requirement 2.5).

Quelltext 3.45: Bubbles (Requirement 2) – `Bubble`

```
61 class Bubble(pygame.sprite.DirtySprite):
62     def __init__(self) -> None:
63         super().__init__()
64         self.radius = Settings.RADIUS["min"]
65         imagename = Settings.get_image("blase1.png")
66         self.image: pygame.surface.Surface = pygame.image.load(imagename).convert_alpha()
```

```

67         self.image = pygame.transform.scale(self.image, (Settings.RADIUS["min"],
68                                         Settings.RADIUS["min"]))
69         self.rect: pygame.Rect = self.image.get_rect()
70         self.dirty = 1
71
72     def update(self, *args: Any, **kwargs: Any) -> None:
73         pass
74
75     def randompos(self) -> None:
76         bubbledistance = Settings.DISTANCE + Settings.RADIUS["min"]
77         centerx = randint(Settings.PLAYGROUND.left + bubbledistance,
78                             Settings.PLAYGROUND.right - bubbledistance)
79         centery = randint(Settings.PLAYGROUND.top + bubbledistance,
80                             Settings.PLAYGROUND.bottom - bubbledistance)
81         self.rect.center = (centerx, centery)

```

Die Klasse `Game` muss nun entsprechend erweitert werden. In der Zeile 87 wird das `Background`-Objekt angelegt. Zeile 88 erzeugt ein `Timer`-Objekt mit einer Intervalllänge von 500 ms, wobei im ersten Intervall noch keine Blasen erzeugt werden sollen (siehe Requirement 2.2).

Der Hintergrund wird nun über den Mechanismus des DirtySprite-Konzepts geblittet. Mit Hilfe von `pygame.sprite.LayeredDirty.clear()` wird in Zeile 90 das entsprechende Image festgelegt. Die Methode `Background.draw()` kann daher gelöscht werden.

`clear()`

Quelltext 3.46: Bubbles (Requirement 2) – Konstruktor von `Game`

```

81 class Game:
82     def __init__(self) -> None:
83         pygame.init()
84         self._screen = pygame.display.set_mode(Settings.WINDOW.size)
85         pygame.display.set_caption(Settings.CAPTION)
86         self._clock = pygame.time.Clock()
87         self._background = Background() # # Timer 500ms
88         self._timer_bubble = Timer(500, False) # Alle Blasen
89         self._all_sprites = pygame.sprite.LayeredDirty() #
90         self._all_sprites.clear(self._screen, self._background.image) #
91         self._all_sprites.set_timing_threshold(1000.0/Settings.FPS)
92         self._running = True

```

In der Methode `draw()` werden lediglich die `draw()`-Methoden der Spritegruppe aufgerufen. Der Hintergrund wird über den Mechanismus des DirtySprite-Konzepts ausgegeben (siehe Abschnitt 2.14 auf Seite 115).

Auch `update()` wurde angepasst, es ruft jetzt die Methode `spawn_bubble()` auf und delegiert damit die Aufgabe, neue Blasen zu erzeugen.

Quelltext 3.47: Bubbles (Requirement 2) – `draw()` und `update()` von `Game`

```

102     def draw(self) -> None:
103         rects = self._all_sprites.draw(self._screen)
104         pygame.display.update(rects) # type: ignore
105
106     def update(self) -> None:
107         self.spawn_bubble()

```

Die Grundidee hinter `spawn_bubble()` ist, solange eine Position für eine neue Blase zu raten, bis man eine freie Fläche gefunden hat. Damit man damit nicht in einer [Endlosschleife](#) landet, wird die Anzahl der Versuche auf 100 begrenzt. Wird keine Freifläche gefunden, wird die Blase nicht der Spritegruppe hinzugefügt – sie verfällt also.

Der Radius wird dazu kurzfristig erweitert (Zeile 114) und nach der Kollisionsprüfung wieder auf seinen Ursprungswert reduziert (Zeile 116).

sprite-
collide()
collide_-
circle()

Sie sehen hier ein Beispiel dafür, dass der Methode `pygame.sprite.spritecollide()` eine Methodenreferenz mitgegeben wird – hier `pygame.sprite.collide_circle()` – und somit nicht die übliche Rechtecksprüfung vorgenommen wird.

Quelltext 3.48: Bubbles (Requirement 2) – `spawn_bubble()` von Game

```

109     def spawn_bubble(self) -> None:
110         if self._timer_bubble.is_next_stop_reached():
111             b = Bubble()
112             for _ in range(100):
113                 b.randompos()
114                 b.radius += Settings.DISTANCE           # Abstand zu Blasen
115                 collided = pygame.sprite.spritecollide(b, self._all_sprites, False,
116                                               pygame.sprite.collide_circle)
117                 b.radius -= Settings.DISTANCE           # Alter Radius!
118                 if not collided:
119                     self._all_sprites.add(b)
                     break

```

Das Ergebnis können Sie in Abbildung 3.7 sehen. Gleichmäßig sind die Bubbles auf der Spielfläche verteilt und der geforderte Abstand zum Rand und zwischen den Blasen ist dabei eingehalten.



Abbildung 3.7: Bubbles: Die Blasen haben beim Start einen Mindestabstand

3.2.3 Requirement 3: Blasenanzahl

Requirement 3 Blasenanzahl

Die maximale Anzahl der Blasen soll von der Spielfeldgröße abhängen.

Die maximale Anzahl will ich in `Game` festlegen. Ausgehend von der Fläche wird eine Obergrenze festgelegt:

Quelltext 3.49: Bubbles (Requirement 3) – Ergänzung von `Settings`

```
22 MAX_BUBBLES = PLAYGROUND.height * PLAYGROUND.width // 10000 # Erfahrungswert
```

Diese Obergrenze aus Zeile 22 wird in Zeile 114 abgefragt. Nur wenn die maximale Anzahl noch nicht erreicht wurde, wird eine neue Blase erzeugt.

Quelltext 3.50: Bubbles (Requirement 3) – Ergänzung von `Game` in `spawn_bubbles()`

```
112 def spawn_bubble(self) -> None:
113     if self._timer_bubble.is_next_stop_reached():
114         if len(self._all_sprites) <= Settings.MAX_BUBBLES: # Platz?
115             b = Bubble()
116             for _ in range(100):
117                 b.randompos()
118                 b.radius += Settings.DISTANCE
119                 collided = pygame.sprite.spritecollide(b, self._all_sprites, False,
120                                             pygame.sprite.collide_circle)
121                 b.radius -= Settings.DISTANCE
122                 if not collided:
123                     self._all_sprites.add(b)
124                     break
```

Der Rest des Programmes bleibt unverändert.

3.2.4 Requirement 4: Blasenwachstum

Requirement 4 Blasenwachstum

1. *Die verschiedenen großen Blasen werden in einem Container verwaltet.*
2. *Der maximale Radius einer Blase ist 240 px.*

Der Sinn von Requirement 4.1 ist das Einsparen von Rechenzeit. Im Spiel werden immer wieder Blasen mit einem bestimmten Radius starten und dann wachsen. Jedes mal das Bitmap auf die passende Größe zu skalieren, würde Rechenzeit verschwenden – schließlich wird die gleiche Blase ja mit den Radien mehrfach vorkommen. Aus diesem Grund ist es sinnvoll, einmal die Blase in alle möglichen Radien zu skalieren und das Ergebnis in einem Dictionary abzulegen. Der Key ist dabei der jeweilige Radius (siehe Zeile 66). scale()

Die Methode `get()` liefert mir dann zu einem Radius das passend skalierte und schon fertige Image. Vorab wird in den Zeilen 69 und 70 überprüft, ob der Radius innerhalb des Gültigkeitsbereich liegt. Falls der Radius dabei zu groß ist, wird der maximale genommen und falls er zu klein ist, der minimale.

Quelltext 3.51: Bubbles (Requirement 4.1) – BubbleContainer

```

62 class BubbleContainer:
63     def __init__(self) -> None:
64         imagename = Settings.get_image("blase1.png")
65         image: pygame.surface.Surface = pygame.image.load(imagename).convert_alpha()
66         self._images = {i: pygame.transform.scale(image, (i * 2, i * 2)) for i in
67                         range(Settings.RADIUS["min"], Settings.RADIUS["max"] + 1)} #
68
69     def get(self, radius: int) -> pygame.surface.Surface:
70         radius = max(Settings.RADIUS["min"], radius) # Untere Grenze
71         radius = min(Settings.RADIUS["max"], radius) # Obere Grenze
72         return self._images[radius]

```

Bisher wurde nur ein Startwert und damit eine untere Grenze für den Blasenradius in `Game` definiert. Diese Definition wird nun in Zeile 19 passend zu Requirement 4.2 um die Angabe eines maximalen Radius erweitert.

Quelltext 3.52: Bubbles (Requirement 4.2) – Erweiterung von Settings

```
19     RADIUS = {"min": 15, "max": 240} # Obergrenze
```

Der `BubbleContainer` wird dem Konstruktor von `Bubble` mitgegeben, so dass diese Klasse sich daraus bedienen kann. Ein Beispiel dafür ist direkt in Zeile 79 zu finden. Das Attribut `image` wird passend zum `radius` besetzt.

Die Methode `update()` ist nun auch nicht mehr leer. Ihre wesentliche Funktion ist das Anwachsen der Blase. Dabei wird der Radius immer weiter erhöht, was zur Folge hat, dass ein immer größeres Image aus dem `BubbleContainer` geladen und angezeigt wird (Zeile 92). Der neue Radius wird in Zeile 89 bestimmt. In der gleichen Zeile wird dieser Wert mit dem maximalen Radius aus `Settings` verglichen und das Minimum der beiden ausgewählt. Diese Logik verhindert, dass der Radius zu groß wird.

Was hat es aber mit den Zeilen 91 und 94 auf sich? Der Referenzpunkt eines Image in einem Sprite ist die linke, obere Ecke. Wächst jetzt die Blase, würde sie sich nach rechts und unten vergrößern; der linke und obere Rand blieben gleich, was hässlich aussieht. Daher merken wir uns den alten Mittelpunkt, laden das neue Image, erzeugen das passende `Rect`-Objekt und verschieben es dann wieder auf den alten Mittelpunkt, so dass optisch die Blase vom Mittelpunkt aus in alle Richtungen wächst.

Quelltext 3.53: Bubbles (Requirement 4) – Ergänzung von Bubble

```

74 class Bubble(pygame.sprite.DirtySprite):
75     def __init__(self, bubble_container: BubbleContainer) -> None:
76         super().__init__()
77         self._bubble_container = bubble_container # Verweis auf Container
78         self.radius = Settings.RADIUS["min"]

```

```

79         self.image = self._bubble_container.get(self.radius)  # Zugriff auf Bubbles
80         self.rect: pygame.Rect = self.image.get_rect()
81         self.dirty = 1
82         self.fradius = float(self.radius)
83         self.speed = 100
84
85     def update(self, *args: Any, **kwargs: Any) -> None:
86         if "action" in kwargs.keys():
87             if kwargs["action"] == "grow":
88                 self.fradius += self.speed * Settings.DELTATIME
89                 self.fradius = min(self.fradius, Settings.RADIUS["max"])  # Neuer Radius
90                 self.radius = round(self.fradius)
91                 center = self.rect.center  # Alter Mittelpunkt
92                 self.image = self._bubble_container.get(self.radius)  # Neues Image
93                 self.rect = self.image.get_rect()
94                 self.rect.center = center  # Neuer MP = Alter MP
95                 self.dirty = 1
96
97     def randompos(self) -> None:
98         bubbledistance = Settings.DISTANCE + Settings.RADIUS["min"]
99         centerx = randint(Settings.PLAYGROUND.left + bubbledistance,
100                         Settings.PLAYGROUND.right - bubbledistance)
101        centery = randint(Settings.PLAYGROUND.top + bubbledistance,
102                         Settings.PLAYGROUND.bottom - bubbledistance)
103        self.rect.center = (centerx, centery)

```

Die Methode `update()` in `Game` muss nur noch um den Aufruf aller `update()` in den Blasen erweitert werden. Dies geht sehr bequem über den Mechanismus der Spritegruppe. Wie bei `draw()` kann auch für die gesamte Gruppe mit einem Schlag `update()` aufgerufen werden (siehe Zeile 131).

Quelltext 3.54: Bubbles (Requirement 4) – Ergänzung von `update()` in `Game`

```

130     def update(self) -> None:
131         self._all_sprites.update(action="grow")  # Bubbles aktualisieren
132         self.spawn_bubble()

```

Der BubbleContainer wird angelegt

Quelltext 3.55: Bubbles (Requirement 4) – Ergänzung im Konstruktor von `Game`

```

109         self._clock = pygame.time.Clock()
110         self._bubble_container = BubbleContainer()
111         self._background = Background()

```

und in der Methode `spawn_bubble()` wird der Aufruf des Konstruktors von `Bubble` um den `BubbleContainer` ergänzt.

Quelltext 3.56: Bubbles (Requirement 4) – Ergänzung von `spawn_bubble()` in `Game`

```

134     def spawn_bubble(self) -> None:
135         if self._timer_bubble.is_next_stop_reached():
136             if len(self._all_sprites) <= Settings.MAX_BUBBLES:
137                 b = Bubble(self._bubble_container)  # Verweis auf Bubbles
138                 for _ in range(100):

```

Die Blasen wachsen nun um ihren Mittelpunkt herum nach außen. Das Ergebnis könnte dann wie in Abbildung 3.8 auf der nächsten Seite aussehen.



Abbildung 3.8: Bubbles: Die Blasen sind gewachsen/verwachsen

3.2.5 Requirement 5: Mauscursor

Requirement 5 Mauscursor

Befindet sich die Maus innerhalb einer Blase, soll sich das Aussehen ändern.

Durch diese Anforderung soll der Spieler optisch unterstützt werden. Er kann schneller erkennen, ob er die Blase schon erreicht hat. Pygame selbst kennt keine Methode/Funktion um zu testen, ob ein Punkt innerhalb eines Kreises liegt. Die Abbildung 3.9 liefert mir aber einen einfachen Ansatz, das Problem zu lösen.

Der Wert d ist der Abstand in Pixel zwischen dem Mittelpunkt des Kreises (x_1, y_1) und dem Punkt (x_2, y_2) . Ist $d \leq r$, so liegt der Punkt innerhalb des Kreises bzw. berührt ihn. Erweitern wir also `Bubble` um eine entsprechende Methode.

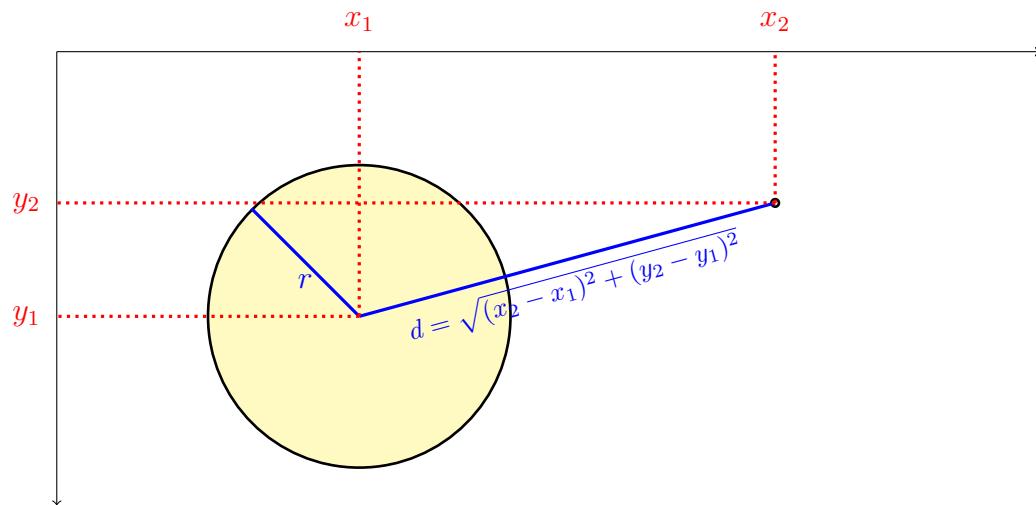


Abbildung 3.9: Kollisionserkennung: Punkt innerhalb des Kreises ([Satz des Pythagoras](#))?

Quelltext 3.57: Bubbles (Requirement 5) – `collidepoint()` in Game

```

149     def collidepoint(self, point: Tuple[int, int], sprite: pygame.sprite.Sprite) -> bool:
150         if hasattr(sprite, "radius"):
151             deltax = point[0] - sprite.rect.centerx # type: ignore
152             deltay = point[1] - sprite.rect.centery # type: ignore
153             return sqrt(deltax * deltax + deltay * deltay) <= sprite.radius # type: ignore
154         return False

```

Mit Hilfe dieser Methode ist die Lösung nun kein Problem mehr. Die Variable `is_over` ist ein Flag, welches sich merken soll, ob die Mauskoordinaten innerhalb der Blase liegen oder nicht. Der Normalfall ist, dass die Maus nicht innerhalb einer Blase liegt, und daher wird die Variable mit `False` initialisiert.

Danach wird mit `pygame.mouse.get_pos()` die aktuelle Mausposition ermittelt. Diese Mausposition wird in Zeile 160 in die Methode `Bubble.collidepoint()` gestopft. Falls eine Blase gefunden wurde, die mit der Maus kollidiert, wird das Flag auf `True` gesetzt und die Schleife mit `break` beendet, was uns ein wenig Rechenzeit einspart, da so nicht mehr alle anderen Blasen untersucht werden. Abhängig vom Flag wird dann der Mauscursor gesetzt.

Quelltext 3.58: Bubbles (Requirement 5) – `set_mousecursor()` in Game

```

160     if self.collidepoint(pos, b): # Innerhalb?
161         is_over = True
162         break
163     if is_over:
164         pygame.mouse.set_cursor(pygame.SYSTEM_CURSOR_HAND)
165     else:
166         pygame.mouse.set_cursor(pygame.SYSTEM_CURSOR_CROSSHAIR)
167
168     def run(self) -> None:
169         time_previous = time()
170         self._running = True

```

Die Methode `update()` in `Game` muss noch um den Aufruf der Überprüfung erweitert werden.

Quelltext 3.59: Bubbles (Requirement 5) – `update()` in Game

```

131     def update(self) -> None:
132         self._all_sprites.update(action="grow")
133         self.spawn_bubble()
134         self.set_mousecursor() # Mauscursor

```

Testen Sie das Programm mal aus. Positionieren Sie die Maus in eine linke untere Ecke außerhalb einer Blase und warten Sie, bis durch das Wachsen die Blase die Maus berührt.

3.2.6 Requirement 6: Blasen zerplatzen

Requirement 6 Blasen zerplatzen

Bei einem Linksklick innerhalb einer Blase, soll die Blase zerplatzen.

MOUSE-
BUTTON-
DOWN
get_pos()

Für die Umsetzung dieser Anforderung ist schon mit der Implementierung der Methode `Bubble.collidepoint()` fast alles erledigt. Wir müssen diese Methode nur geschickt einsetzen – es sind in der Tat nur wenige Restarbeiten nötig. In `watch_for_events()` wird zunächst der linke Mausklick abgefangen (Zeile 126) und die aktuelle Mausposition an die – neu erstellte Methode – `sting()` übergeben (Zeile 128).

Hinweis: Implementieren Sie grundsätzlich so wenig Logik wie möglich in `watch_for_events()`. Diese Methode ist ein Verteiler; die Verarbeitung sollte immer in Methoden ausgelagert werden.

Quelltext 3.60: Bubbles (Requirement 6) – `watch_for_event()` in Game

```

119  def watch_for_events(self) -> None:
120      for event in pygame.event.get():
121          if event.type == QUIT:
122              self._running = False
123          elif event.type == KEYDOWN:
124              if event.key == K_ESCAPE:
125                  self._running = False
126          elif event.type == MOUSEBUTTONDOWN: # Mausklick?
127              if event.button == 1: # left
128                  self.sting(pygame.mouse.get_pos()) # Aufruf

```

Die Methode `sting()` ist nun denkbar simpel. Es werden alle `Bubble`-Objekte durchwandert und dahingehend abgefragt, ob die Mausposition innerhalb des Radius liegt (Zeile 173). Wenn *Ja*, dann wird das entsprechende Objekt aus der `Spritegroup` mit `kill()` entfernt.

kill()

Quelltext 3.61: Bubbles (Requirement 6) – `sting()` in Game

```

171  def sting(self, mousepos: Tuple[int, int]) -> None:
172      for bubble in self._all_sprites:
173          if self.collidepoint(mousepos, bubble): # Innerhalb?
174              bubble.kill()

```

3.2.7 Requirement 7: Punktestand

Requirement 7 Punktestand

1. Das Spiel startet mit 0 Punkten.
2. Zerplatzt eine Blase, wird der Punktestand proportional zum Radius erhöht.
3. Der Punktestand wird im unteren Teil angezeigt.

Das Anstechen der Blasen soll natürlich mit Punkten belohnt werden. Dazu müssen die Punkte ermittelt und ausgegeben werden. Die einfachste Art den Punktestand festzuhalten ist eine statische Variable in `Settings` oder eine globale Variable. Ich bevorzuge Variante 1 (Quelltext 3.62 auf der nächsten Seite).

Quelltext 3.62: Bubbles (Requirement 7.1) – Erweiterung von Game

24 POINTS = 0 # Globaler Punktestand

Da das Anstechen nun nicht mehr nur für ein Verschwinden sorgt, sondern auch für die Aktualisierung des Punktestands, habe ich dazu einen neuen Methoden in `Bubble` angelegt. In Zeile 110 wird einfach der Radius der Blase auf den Punktestand addiert.

Quelltext 3.63: Bubbles (Requirement 7.2) – `stung()` in `Bubble`108 def stung(self):
109 self.kill()
110 Settings.POINTS += self.radius # Increment points

Der Aufruf von `stung()` erfolgt durch ein angepasstes `update()`.

Quelltext 3.64: Bubbles (Requirement 7.2) – `update()` in `Bubble`88 def update(self, *args: Any, **kwargs: Any) -> None:
89 if "action" in kwargs.keys():
90 if kwargs["action"] == "grow":
91 self.fradius += self.speed * Settings.DELTATIME
92 self.fradius = min(self.fradius, Settings.RADIUS["max"])
93 self.radius = round(self.fradius)
94 center = self.rect.center
95 self.image = self._bubble_container.get(self.radius)
96 self.rect = self.image.get_rect()
97 self.rect.center = center
98 self.dirty = 1
99 elif kwargs["action"] == "sting":
100 self.stung()

Die Methoden `sting()` und `update()` in `Game` müssen dazu passend verändert werden (Zeile 199 und Zeile 160).

Quelltext 3.65: Bubbles (Requirement 7.2) – `sting()` in `Game`196 def sting(self, mousepos: Tuple[int, int]) -> None:
197 for bubble in self._all_sprites:
198 if self.collidepoint(mousepos, bubble):
199 bubble.update(action="sting") # Nach Bubble verschobenQuelltext 3.66: Bubbles (Requirement 7.2) – `update()` in `Game`159 def update(self) -> None:
160 self._all_sprites.update(action="grow") #
161 self.spawn_bubble()
162 self.set_mousecursor()

Verbleibt Requirement 7.3. Ähnlich wie für die Spielfläche möchte ich die Maße für den unteren Teil als Ausgabebox in `Settings` festlegen.

Rect

Quelltext 3.67: Bubbles (Requirement 7.3) – Erweiterung von `Settings`

25 BOX = pygame.Rect(90, 770, 1055, 130) # Ausgabebox

Für die Punktausgabe selbst bastle ich mir wieder eine kleine Klasse, die das Problem kapselt: `Points`. Im Konstruktor wird ein `Font`-Objekt erzeugt, welches mir in `update()` den Punktestand rendert. Die Position der Textausgabe wird aus den Angaben in `Settings` ermittelt. Den Rest erledigt die `Sprite`-Klasse für mich.

Quelltext 3.68: Bubbles (Requirement 7.3) – Points

```

113 class Points(pygame.sprite.DirtySprite):
114     def __init__(self) -> None:
115         super().__init__()
116         self._font = pygame.font.Font(pygame.font.get_default_font(), 18)
117         self.oldpoints = -1
118         self.dirty = 1
119
120     def update(self, *args: Any, **kwargs: Any) -> None:
121         if self.oldpoints != Settings.POINTS:
122             self.image = self._font.render(f"Points: {Settings.POINTS}", True, "red")
123             self.rect = self.image.get_rect()
124             self.rect.left = Settings.BOX.left
125             self.rect.top = Settings.BOX.top
126             self.dirty = 1

```

Verbleiben einige Erweiterungen in `Game`. Im Konstruktor wird das `Points`-Objekt in das `LayeredDirty`-Objekt gesteckt.

Quelltext 3.69: Bubbles (Requirement 7.3) – Erweiterung des Konstruktors von Game

```

140     self._all_sprites.set_timing_threshold(1000.0 / Settings.FPS)
141     self._all_sprites.add(Points()) # Points
142     self._running = True

```



Abbildung 3.10: Bubbles: Ausgabe Punktestand

In Abbildung 3.10 können Sie die Punkteausgabe in der unteren Hälfte sehen. Diese Fläche könnte man später auch noch für eine Liste der besten zehn Punktestände oder andere Ausgaben verwenden.

3.2.8 Requirement 8: Spielende

Requirement 8 Spielende

1. Berühren sich zwei Blasen, ist das Spiel verloren.
2. Berührt eine Blase den Rand, ist das Spiel verloren.

Hinweis: Damit das Spiel spielbar wird, habe ich die Wachstumsgeschwindigkeit einer Bubble auf 10 gesetzt.

Quelltext 3.70: Bubbles (Requirement 8) – `Bubble.speed`

86

```
self.speed = 10
```

Die Grundstruktur unseres Spiels ermöglicht es, diese Anforderung recht leicht durch eine Erweiterung von `update()` in `Game` zu realisieren.

Quelltext 3.71: Bubbles (Requirement 8) – Erweiterung von `update()` in `Game`

```
162     def update(self) -> None:
163         if self.check_bubblecollision(): # Spielende?
164             self._running = False
165         else:
166             self._all_sprites.update(action="grow")
167             self.spawn_bubble()
168             self.set_mousecursor()
```

In der neuen Methode `check_bubblecollision()` wird überprüft, ob sich Blasen berühren oder eine Blase an den Rand stößt. Diese Methode wird einfach als Entscheider (Zeile 163) dafür genommen, ob das Spiel zu beenden ist. Falls *Ja*, wird das Flag der Hauptprogrammschleife gesetzt; falls *Nein*, wird wie gewohnt die restliche Spiellogik abgearbeitet. Die beiden verschachtelten `for`-Schleifen ab Zeile 208 durchwandern die Gruppe der Blasen zweimal und vermeiden dabei zwei Dinge:

`sprites()`

- Eine Blase darf sich nicht mit sich selbst vergleichen: Daher beginnt der Index der inneren Schleife immer um eins versetzt zum aktuellen Index der äußeren Schleife, und der äußere Schleifenindex endet vor dem letzten Element der Blasengruppe.
- Wenn Blase 1 schon mit Blase 2 verglichen wurde, sollte Blase 2 nicht nochmal mit Blase 1 verglichen werden: Auch dies wird durch den versetzen Index erreicht.

In Zeile 213 wird Requirement 8.1 überprüft. Dabei wird die auf der Kreisform basierende Kollisionsprüfung mit `collide_circle()` verwendet. In der Zeile 215 und Zeile 217 wird Requirement 8.2 umgesetzt. Dabei wird ausgenutzt, dass die Spielfläche ein Rechteck ist und das Sprite ebenfalls ein Rechteck besitzt. Die Methode `pygame.rect.Rect.contains()` überprüft dabei, ob ein Rechteck innerhalb eines anderen liegt. Ist dies nicht der Fall – also verlässt die Blase die Spielfläche –, liegt eine Kollision vor.

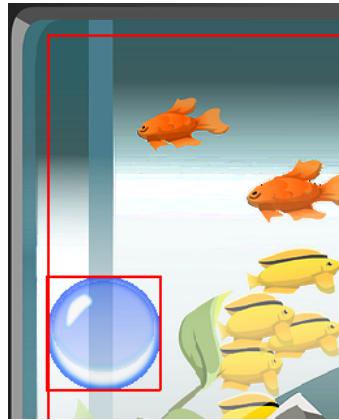


Abbildung 3.11: Bubbles – Kollision mit dem Rand

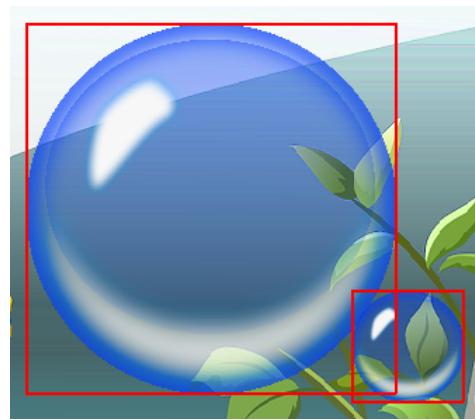


Abbildung 3.12: Bubbles – Kollision der Blasen

Quelltext 3.72: Bubbles (Requirement 8) – `check_bubblecollision()` in `Game`

```

207     def check_bubblecollision(self) -> bool:
208         for index1 in range(0, len(self._all_sprites) - 1): # Bubbles prüfen
209             for index2 in range(index1 + 1, len(self._all_sprites)):
210                 bubble1 = self._all_sprites.sprites()[index1]
211                 bubble2 = self._all_sprites.sprites()[index2]
212                 if type(bubble1).__name__ == "Bubble" and type(bubble2).__name__ == "Bubble":
213                     if pygame.sprite.collide_circle(bubble1, bubble2): # Blasen kollidieren
214                         return True
215                     if not Settings.PLAYGROUND.contains(bubble1): # Blase1 berührt Rand
216                         return True
217                     if not Settings.PLAYGROUND.contains(bubble2): # Blase2 berührt Rand
218                         return True

```

In Abbildung 3.11 wird die Kollision der Blase mit dem Rand dargestellt. Um das besser erkennen zu können, habe ich Hilfslinien ausgegeben. Sie können gut sehen, dass das Rechteck der Blase nicht mehr im Rechteck der Spielfläche liegt. Abbildung 3.12 zeigt die Kollision zweier Blasen. Auch hier sind Hilfslinien eingezeichnet. Die Hilfslinien werden Ihnen eingezeichnet, wenn Sie die drei Kommentarzeichen in `Game.draw()` entfernen.

Quelltext 3.73: Bubbles (Requirement 8) – Hilfslinien in `Game`

```

155     def draw(self) -> None:
156         rects = self._all_sprites.draw(self._screen)
157         # pygame.draw.rect(self._screen, "red", Settings.playground, 2)
158         # for b in self._all_bubbles:
159         #     pygame.draw.rect(self._screen, "red", b.rect, 2) # type: ignore
160         pygame.display.update(rects) # type: ignore

```

3.2.9 Requirement 9: Zeitanpassungen

Requirement 9 Zeitanpassungen

Die Blasen sollen im Lauf der Zeit schneller wachsen.

Weil im Laufe der Zeit die Blasen schneller wachsen sollen, will ich ihnen die Wachstumsgeschwindigkeit als Übergabeparameter im Konstruktor mitgeben. In Zeile 86 wird dieser Parameter in ein Attribut geparkt.

Quelltext 3.74: Bubbles (Requirement 9) – `Bubble`

```
77 class Bubble(pygame.sprite.DirtySprite):
78     def __init__(self, bubble_container: BubbleContainer, speed: int) -> None:
79         super().__init__()
80         self._bubble_container = bubble_container
81         self.radius = Settings.RADIUS["min"]
82         self.image = self._bubble_container.get(self.radius)
83         self.rect: pygame.Rect = self.image.get_rect()
84         self.dirty = 1
85         self.fradius = float(self.radius)
86         self.speed = speed # Wachstumsgeschwindigkeit
```

Das sind schon alle Anpassungen in `Bubble`, der Rest passiert in `Game`. In Zeile 138 wird ein Timer erstellt, der mir alle 1000 ms ein Signal geben wird. Darunter wird die anfängliche Wachstumsgeschwindigkeit der Blasen auf 10 px/s gestellt.

Timer

Quelltext 3.75: Bubbles (Requirement 9) – Konstruktor von `Game`

```
136     self._background = Background()
137     self._timer_bubble = Timer(500, False)
138     self._timer_bubble_speed = Timer(1000, False) #
139     self._bubble_speed = 10
140     self._all_sprites = pygame.sprite.LayeredDirty()
```

In `spawn_bubble()` wird der Timer abgefragt und ggf. die Blasenwachstumsgeschwindigkeit¹ erhöht (Zeile 173). Die maximale Wachstumsgeschwindigkeit wird dabei auf 100 px/s begrenzt; schneller scheint mir nicht spielbar. Bei jedem Timer-Signal wird dabei die Geschwindigkeit um 5 px/s erhöht. Dies geschieht in dieser Methode, da dann die neue Geschwindigkeit für die zu erstellenden Blasen zur Verfügung steht.

Quelltext 3.76: Bubbles (Requirement 9) – `Game.spawn_bubble()`

```
172     def spawn_bubble(self) -> None:
173         if self._timer_bubble_speed.is_next_stop_reached(): #
174             if self._bubble_speed < 100:
175                 self._bubble_speed += 5
176         if self._timer_bubble.is_next_stop_reached():
177             if len(self._all_sprites) <= Settings.MAX_BUBBLES:
178                 b = Bubble(self._bubble_container, self._bubble_speed)
179                 for _ in range(100):
180                     b.randompos()
181                     b.radius += Settings.DISTANCE
182                     collided = pygame.sprite.spritecollide(b, self._all_sprites, False,
183                                                 pygame.sprite.collide_circle)
184                     b.radius -= Settings.DISTANCE
185                     if not collided:
186                         self._all_sprites.add(b)
187                         break
```

¹ Deutsch ist schon eine coole Sprache ;-)

Wenn Sie jetzt das Spiel ausprobieren (`bubbles09.py`), werden Sie einen leichten Start und eine moderate Steigerung der Spielschwierigkeit bemerken.

3.2.10 Requirement 10: Kollision anzeigen

Requirement 10 Kollision anzeigen

Wenn Blasen mit dem Rand oder miteinander kollidieren, sollen sie die Farbe wechseln und für 2 s sichtbar bleiben, bevor die Anwendung sich beendet.

Bisher beendet sich das Spiel so schnell, dass ich nicht überprüfen kann, ob ich eigentlich zu Recht verloren habe, oder ob das Programm spinnt. Ich möchte durch diese Anforderung die beiden kollidierenden Blasen oder die Blase, die den Rand berührt, andersfarbig sehen können. Ich habe dazu die Blase rot eingefärbt (siehe Abbildung 3.13).



Dazu braucht es einen zweiten `BubbleContainer` mit den skalierten roten Blasen. Um auf diese leichter zugreifen zu können, sind diese in `Game` als statisches Dictionary angelegt.

In Zeile 138 ist dazu ein Dictionary angelegt worden. Unter einem Schlüssel kann ich dort nun beliebige `BubbleContainer`-Objekte ablegen.

Quelltext 3.77: Bubbles (Requirement 10) – Erweiterung von Game

```
137 class Game:
138     BUBBLE_CONTAINER: Dict[str, BubbleContainer] = {} # Mehrere
```

Der Konstruktor von `BubbleContainer` bekommt nun einen Dateinamen mitgegeben, so dass hier verschiedene Grafiken zu Grunde gelegt werden können.

Quelltext 3.78: Bubbles (Requirement 10) – Änderung Konstruktor von BubbleContainer

```
65 class BubbleContainer:
66     def __init__(self, filename: str) -> None: # Jetzt mit Dateinamen
67         imagename = Settings.get_image(filename)
68         image: pygame.surface.Surface = pygame.image.load(imagename).convert_alpha()
69         self._images = {i: pygame.transform.scale(image, (i * 2, i * 2)) for i in
70             range(Settings.RADIUS["min"], Settings.RADIUS["max"] + 1)}
```

Der Konstruktor von `Game` füllt nun das statische Dictionary `BUBBLE_CONTAINER` auf (Zeile 145 und Zeile 146).

Quelltext 3.79: Bubbles (Requirement 10) – Änderung vom Konstruktor von Game

```
144     self._clock = pygame.time.Clock()
145     Game.BUBBLE_CONTAINER["blue"] = BubbleContainer("blase1.png") # blau
146     Game.BUBBLE_CONTAINER["red"] = BubbleContainer("blase2.png") # rot
147     self._background = Background()
```

In Bubble sind nun mehrere Änderungen nötig. Durch das neue Attribut `mode` (Zeile 80) wird die Farbe der Blase bestimmt. Jedesmal, wenn nun das Image aus dem `BubbleContainer` geladen wird, wird über dieses Attribut gesteuert, welcher der beiden `BubbleContainer` als Datenquelle verwendet werden soll. Beispielhaft sei hier Zeile 95 in `update()` erwähnt.

Quelltext 3.80: Bubbles (Requirement 10) – Konstruktor von `Bubble` und `update()`

```

77 class Bubble(pygame.sprite.DirtySprite):
78     def __init__(self, speed: int) -> None:
79         super().__init__()
80         self.mode = "blue" # Farbmodus
81         self.radius = Settings.RADIUS["min"]
82         self.image = Game.BUBBLE_CONTAINER[self.mode].get(self.radius)
83         self.rect: pygame.rect.Rect = self.image.get_rect()
84         self.dirty = 1
85         self.fradius = float(self.radius)
86         self.speed = speed
87
88     def update(self, *args: Any, **kwargs: Any) -> None:
89         if "action" in kwargs.keys():
90             if kwargs["action"] == "grow":
91                 self.fradius += self.speed * Settings.DELTATIME
92                 self.fradius = min(self.fradius, Settings.RADIUS["max"])
93                 self.radius = round(self.fradius)
94                 center = self.rect.center
95                 self.image = Game.BUBBLE_CONTAINER[self.mode].get(self.radius) #
96                 self.rect = self.image.get_rect()
97                 self.rect.center = center
98                 self.dirty = 1
99             elif kwargs["action"] == "sting":
100                 self.stung()
101             elif "mode" in kwargs.keys():
102                 self.set_mode(kwargs["mode"])

```

Ändert sich der Modus, muss die andere Farbe nachgeladen werden. Dies erfüllt die Methode `set_mode` in `Bubble`.

Quelltext 3.81: Bubbles (Requirement 10) – `set_mode()` in `Bubble`

```

104     def set_mode(self, mode: str) -> None:
105         if mode != self.mode:
106             self.dirty = 1
107             self.mode = mode
108             self.image = Game.BUBBLE_CONTAINER[self.mode].get(self.radius)

```

Jetzt muss nur noch im Falle einer Kollision – also eines Spielendes – der Modus geändert werden. In Abbildung 3.14 auf der nächsten Seite können Sie sehen, wie die beiden kollidierenden Blasen rot erscheinen. Wie das geschieht, können Sie beispielhaft in Zeile 230 sehen.

Quelltext 3.82: Bubbles (Requirement 10) – `check_bubblecollision()` in `Game`

```

223     def check_bubblecollision(self) -> bool:
224         for index1 in range(0, len(self._all_sprites) - 1):
225             for index2 in range(index1 + 1, len(self._all_sprites)):

```

```

226         bubble1 = self._all_sprites.sprites()[index1]
227         bubble2 = self._all_sprites.sprites()[index2]
228         if type(bubble1).__name__ == "Bubble" and type(bubble2).__name__ == "Bubble":
229             if pygame.sprite.collide_circle(bubble1, bubble2):
230                 bubble1.update(mode="red") # rot
231                 bubble2.update(mode="red")
232             return True
233             if not Settings.PLAYGROUND.contains(bubble1):
234                 bubble1.update(mode="red")
235             return True
236             if not Settings.PLAYGROUND.contains(bubble2):
237                 bubble2.update(mode="red")
238             return True
239         return False

```

Damit mir Zeit bleibt, die Kollision zu sehen, will ich am Ende 2 s warten. Die Methode `pygame.time.wait()` hält die Anwendung entsprechend lang an (Zeile 252).

Quelltext 3.83: Bubbles (Requirement 10) – Wartezeit in `run()`

```

241     def run(self) -> None:
242         time_previous = time()
243         self._running = True
244         while self._running:
245             self.watch_for_events()
246             self.update()
247             self.draw()
248             self._clock.tick(Settings.FPS)
249             time_current = time()
250             Settings.DELTATIME = time_current - time_previous
251             time_previous = time_current
252             pygame.time.wait(2000) # Kurz warten
253             pygame.quit()

```

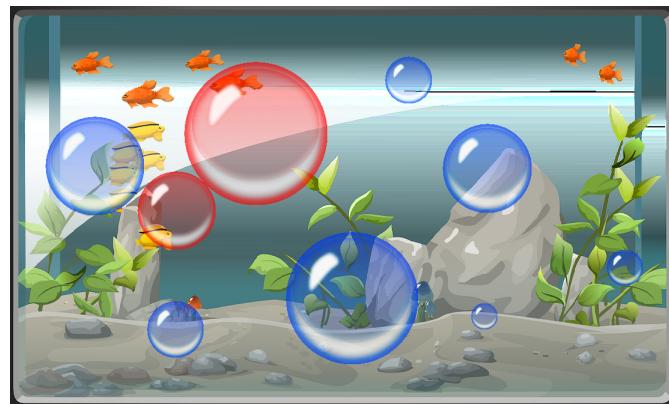


Abbildung 3.14: Bubbles: Kollision anzeigen

3.2.11 Requirement 11: Pause

Requirement 11 Pause

Mit der rechten Maustaste oder der Taste „P“ springt das Spiel in den Pausenmodus oder beendet diesen. Der aktuelle Spielstand friert ein und wird „eingegraut“.

Die Idee hinter dieser Anforderung ist, dass eine notwendige Unterbrechung nicht zwangsläufig bedeutet, dass man verliert. In Abbildung 3.15 können Sie sehen, wie der Pausenbildschirm aussehen sollte. Zunächst werden die Pygame-Konstanten KEYUP, MOUSEBUTTONDOWNUP und K_p importiert (Zeile 8).

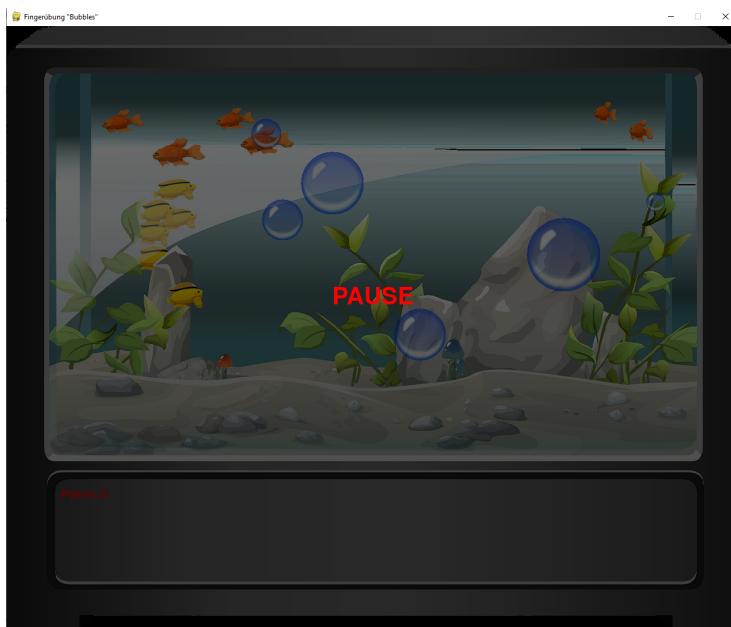


Abbildung 3.15: Bubbles: Pausenbildschirm

Quelltext 3.84: Bubbles (Requirement 11) – Zusätzliche Konstanten

```
8 from pygame.constants import K_ESCAPE, KEYDOWN, KEYUP, MOUSEBUTTONDOWN, MOUSEBUTTONUP, QUIT,  
    K_p #
```

Im Konstruktor von `Game` wird das Flag `pausing` definiert. Dieser steuert später, ob sich das Spiel im Pausenmodus befindet oder nicht.

Quelltext 3.85: Bubbles (Requirement 11) – Konstruktor in Game

In `watch_for_events()` wird nun abgefragt, ob die P-Taste (Zeile 175) oder die rechte Maustaste (Zeile 178) gedrückt wurde. In beiden Fällen wird die neue Methode `setpause()` aufgerufen.

Quelltext 3.86: Bubbles (Requirement 11) – `watch_for_events()` in Game

```

167  def watch_for_events(self) -> None:
168      for event in pygame.event.get():
169          if event.type == QUIT:
170              self._running = False
171          elif event.type == KEYDOWN:
172              if event.key == K_ESCAPE:
173                  self._running = False
174              elif event.type == KEYUP:
175                  if event.key == K_p:  #
176                      self.setpause()
177                  elif event.type == MOUSEBUTTONUP:
178                      if event.button == 3:  # right
179                          self.setpause()
180                  elif event.type == MOUSEBUTTONDOWN:
181                      if event.button == 1:  # left
182                          self.sting(pygame.mouse.get_pos())

```

Für die Darstellung der Pause, habe ich die – vielleicht etwas überflüssige – Klasse `Pause` implementiert.

Quelltext 3.87: Bubbles (Requirement 11) – Pause

```

76  class Pause(pygame.sprite.DirtySprite):
77      def __init__(self) -> None:
78          super().__init__()
79          imagename = Settings.get_image("pause.png")
80          self.image: pygame.surface.Surface = pygame.image.load(imagename).convert_alpha()
81          self.rect = self.image.get_rect()
82          self.dirty = 1

```

Im Konstruktor von `Game` wird ein Objekt der Klasse `Pause` angelegt, damit es in `draw()` verwendet werden kann.

Quelltext 3.88: Bubbles (Requirement 11) – Konstruktor in Game

```

165      self._pause = Pause()

```

Ich muss aber noch die Methode `setpause()` erklären. Diese fügt das `Pause`-Objekt in die Liste der Sprites ein oder holt sie wieder raus, abhängig davon, ob ich im Pausenmodus bin oder nicht. Anschließend wird der Boolesche-Wert des Flags negiert ([Toggling](#)).

Quelltext 3.89: Bubbles (Requirement 11) – `setpause()` in Game

```

200  def setpause(self):
201      if not self._pausing:
202          self._all_sprites.add(self._pause)
203      else:
204          self._pause.kill()
205      self._pausing = not self._pausing

```

Mehr ist nicht nötig, da der Rest von den üblichen `update()`- und `draw()`-Mechanismen erledigt wird.

3.2.12 Requirement 12: Neustart

Requirement 12 Neustart

Am Ende des Spiels soll erfragt werden, ob der Spieler das Spiel neu starten möchte oder nicht.

Die Grundidee der Implementierung ist dabei, dass mit Hilfe von zwei Flags der Status des Spiels festgelegt wird. Wie bei der Pause brauchen wir ein Flag, welches steuert, ob der halbtransparente Vordergrund über das Spiel gelegt wird (`_restarting`). Dies ist immer dann der Fall, wenn die Kollisionsprüfung der Blasen eine Kollision feststellt. Flag

Das andere Flag – `_do_start` – markiert, ob der Spieler einen Neustart möchte. An den entscheidenden Stellen in `update()` und `draw()` werden dann diese Flags abgefragt.

Zunächst werden die Tastaturkonstanten für die Antwort importiert (Zeile 8).

Quelltext 3.90: Bubbles (Requirement 12) – Erweiterung des Imports

```
8 from pygame.constants import K_ESCAPE, KEYDOWN, KEYUP, MOUSEBUTTONDOWN, MOUSEBUTTONUP, QUIT,
  K_j, K_n, K_p #
```

Die Aufgabe, eine Rückfrage in den Vordergrund zu schieben, ist eigentlich schon mit der Klasse `Pause` gelöst; ich kann daher die Klasse verallgemeinern, indem ich sie in `Message` umbenenne und den Dateinamen dem Konstruktor als String-Parameter übergebe (Zeile 77).

Quelltext 3.91: Bubbles (Requirement 12) – von Pause zu Message

```
76 class Message(pygame.sprite.DirtySprite):
77     def __init__(self, filename: str) -> None: #
78         super().__init__()
79         imagename = Settings.get_image(filename)
80         self.image: pygame.surface.Surface = pygame.image.load(imagename).convert_alpha()
81         self.rect = self.image.get_rect()
82         self.dirty = 1
```

In `Game` werden im Konstruktor ab Zeile 161 die Anpassungen für den Neustart programmiert. Im wesentlichen werden für die Pause und den Neustart die beiden `Message`-Objekte erzeugt und alle Attribute, die bei einem Start/Neustart zurückgesetzt werden müssen, werden in der neuen Methode `restart()` bearbeitet.

Quelltext 3.92: Bubbles (Requirement 12) – Umbau Konstruktor von Game

```
148     def __init__(self) -> None:
149         pygame.init()
```

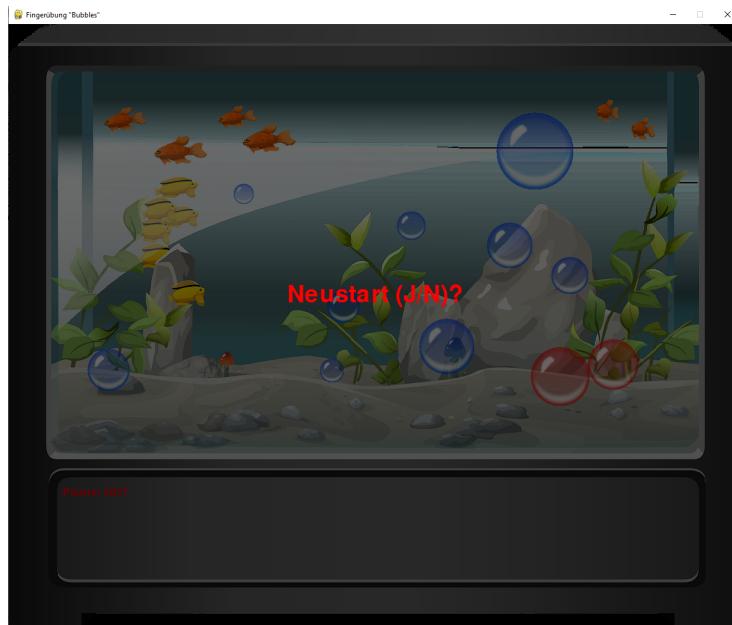


Abbildung 3.16: Bubbles: Neustartbildschirm

```

150     self._screen = pygame.display.set_mode(Settings.WINDOW.size)
151     pygame.display.set_caption(Settings.CAPTION)
152     self._clock = pygame.time.Clock()
153     Game.BUBBLE_CONTAINER["blue"] = BubbleContainer("blase1.png")
154     Game.BUBBLE_CONTAINER["red"] = BubbleContainer("blase2.png")
155     self._background = Background()
156     self._all_sprites = pygame.sprite.LayeredDirty()
157     self._all_sprites.clear(self._screen, self._background.image)
158     self._all_sprites.set_timing_threshold(1000.0 / Settings.FPS)
159     self._running = True
160     self._pausing = False
161     self._pause = Message("pause.png")  #
162     self._restart = Message("neustart.png")
163     self.restart()

```

Der Punktestand wird zurückgesetzt, die Spritegroup von den Blasen geleert, die Timer neu aufgesetzt, die Blasenwachstumsgeschwindigkeit auf Anfang gesetzt und die beiden oben beschriebenen Flags auf `False` gesetzt.

Quelltext 3.93: Bubbles (Requirement 12) – `restart()` in Game

```

206     def restart(self):
207         Settings.POINTS = 0
208         self._all_sprites.empty()
209         self._all_sprites.add(Points())
210         self._bubble_speed = 10
211         self._timer_bubble = Timer(500, False)
212         self._timer_bubble_speed = Timer(10000, False)
213         self._do_start = False
214         self._restarting = False

```

Aufgerufen wird die Methode in `update()`, wenn das entsprechende Flag `do_start` ge-

setzt ist. Auch wird in `update()` der Neustart-Bildschirm in die Spritegroup eingefügt und das Flag `_restarting` auf `True` gesetzt, wenn eine Kollision erkannt wurde.

Quelltext 3.94: Bubbles (Requirement 12) – `update()` in Game

```

193     def update(self) -> None:
194         if self._do_start: # Neustart?
195             self.restart()
196         if not self._pausing and self._running:
197             if self.check_bubblecollision():
198                 if not self._restarting:
199                     self._all_sprites.add(self._restart)
200                     self._restarting = True
201                 else:
202                     self._all_sprites.update(action="grow")
203                     self.spawn_bubble()
204                     self.set_mousecursor()

```

Die Antwort auf den Neustart-Bildschirm wird in `watch_for_events()` abgefragt und in entsprechende Flaginhalte umgesetzt. Antwortet der Spieler mit `J` (Zeile 175), muss das Spiel ja neu gestartet werden. Deshalb wird `do_start` auf `True` gesetzt. Gibt er `N` ein, soll das Spiel beendet werden, weshalb das Flag der Hauptprogrammschleife mit `False` bestückt wird (Zeile 177).

Quelltext 3.95: Bubbles (Requirement 12) – Erweiterung von `watch_for_events()`

```

172     elif event.type == KEYUP:
173         if event.key == K_p:
174             self.setpause()
175         elif event.key == K_j: #
176             self._do_start = True
177         elif event.key == K_n: #
178             self._running = False

```

Da wir nun am Spielende eine halbtransparente Vordergrundausgabe haben, brauchen wir keine zweisekündige Pause, um die kollidierenden Blasen anzusehen (siehe Abbildung 3.16 auf der vorherigen Seite). Daher kann Zeile 292 auskommentiert werden.

Quelltext 3.96: Bubbles (Requirement 12) – `run()` in Game

```

292     # pygame.time.wait(2000) # Neustart?

```

3.2.13 Requirement 13: Sound

Requirement 13 Sound

1. Das Erscheinen der Blasen wird mit einem Sound unterlegt.
 2. Das Zerstechen wird mit einem Sound unterlegt.
 3. Das Berühren wird mit einem Sound unterlegt.
-

Zum Schluss noch eine kleine Sound-Untermalung. Ähnlich wie bei den Blasen-Sprites, möchte ich keine Performance durch permanentes Laden der Sound-Dateien verlieren. Daher werden die Sounds in einem statischen Dictionary abgelegt (Zeile 147).

Quelltext 3.97: Bubbles (Requirement 13) – SOUND_CONTAINER

```
145 class Game:
146     BUBBLE_CONTAINER: Dict[str, BubbleContainer] = {}
147     SOUND_CONTAINER: Dict[str, pygame.mixer.Sound] = {} #
```

Im Konstruktor von Game wird das Dictionary mit Objekten der Sound-Klasse aufgefüllt. Die Klasse für Soundeffekte ist `pygame.mixer.Sound` (siehe Zeile 156ff.).

Sound

Quelltext 3.98: Bubbles (Requirement 13) – SOUND_CONTAINER auffüllen

```

155     Game.BUBBLE_CONTAINER["red"] = BubbleContainer("blase2.png")
156     Game.SOUND_CONTAINER["bubble"] =
157         pygame.mixer.Sound(Settings.get_sound("plopp1.mp3")) #
158     Game.SOUND_CONTAINER["burst"] = pygame.mixer.Sound(Settings.get_sound("burst.mp3"))
159     Game.SOUND_CONTAINER["clash"] = pygame.mixer.Sound(Settings.get_sound("glas.wav"))
        self._background = Background()

```

Nun müssen die Sounds nur noch an der geeigneten Stelle mit `pygame.mixer.Sound.play()` abgespielt werden. Zuerst der Sound, wenn eine neue Blase erscheint: in `spawn_bubble()` in Zeile 243.

`play()`

Quelltext 3.99: Bubbles (Requirement 13.1) – `spawn_bubble()`

```

241             if not collided:
242                 self._all_sprites.add(b)
243                 Game.SOUND_CONTAINER["bubble"].play() #
244                 break

```

Dann, wenn in `sting()` eine Blase zerplatzt (Zeile 268):

Quelltext 3.100: Bubbles (Requirement 13.2) – `sting()`

```

265     def sting(self, mousepos: Tuple[int, int]) -> None:
266         for bubble in self._all_sprites:
267             if self.collidepoint(mousepos, bubble):
268                 Game.SOUND_CONTAINER["burst"].play() #
269                 bubble.update(action="sting")

```

Und zum Schluss bei der Kollision mit anderen Blasen oder dem Rand in `update()`. Dabei muss noch berücksichtigt werden, ob das Spiel gerade den Abfrage für den Neustart anzeigt. Wenn *Ja*, darf der Sound nicht noch einmal abgespielt werden; ansonsten würde permanent der Berühren-Sound abgespielt.

Quelltext 3.101: Bubbles (Requirement 13.3) – `update()`

```

199     if self._do_start:
200         self.restart()
201     if not self._pausing and self._running:
202         if self.check_bubblecollision():
203             if not self._restarting:
204                 Game.SOUND_CONTAINER["clash"].play() #
205                 self._all_sprites.add(self._restart)
206                 self._restarting = True
207             else:
208                 self._all_sprites.update(action="grow")
209                 self.spawn_bubble()
210                 self.set_mousecursor()

```

Und Schluss :-)

Abbildungsverzeichnis

2.1	Spielfläche	5
2.2	Ressourcenverbrauch ohne Taktung	7
2.3	Ressourcenverbrauch mit Taktung	8
2.4	Einige Grafikprimitive	12
2.5	Sicher keine Partikelfontaine	13
2.6	Version 2	14
2.7	Partikelfontaine, Version 5: fast fertig	16
2.8	Bitmaps laden und ausgeben, Version 1.0	22
2.9	Größen OK	22
2.10	Transparenz OK	23
2.11	Bitmaps positionieren (Verteidiger)	25
2.12	Bitmaps positionieren (Angreifer, Version 1)	26
2.13	Bitmaps positionieren (Angreifer, Version 2)	27
2.14	Bitmaps positionieren (Angreifer, Version 3)	27
2.15	Elemente eines <code>Rect</code> -Objekts	29
2.16	Bitmaps bewegen, Version 1.0	31
2.17	Der Verteidiger bewegt sich und prallt ab	33
2.18	Nicht normalisierte Bewegung	36
2.19	Vergleich des Positionsfehlers von $1/fps$ und <code>pygame.clock.tick()</code>	39
2.20	Vergleich des Positionsfehlers von <code>Rect</code> und <code>FRect</code>	41
2.21	Vergleich der Positionsfehler mit unterschiedlichen Genauigkeiten	42
2.22	Normalisierte Bewegung mit $1/fps$	43
2.23	Normalisierte Bewegung mit <code>pygame.clock.tick()</code>	43
2.24	Normalisierte Bewegung mit <code>pygame.clock.tick()</code> (float)	44
2.25	Normalisierte Bewegung mit <code>time.time()</code>	44
2.26	Ränder	54
2.27	Textausgabe mit Fonts	62
2.28	Fontliste	65
2.29	Beispiel für eine Spritelib	69
2.30	Bedeutung der Angaben in Spritelib	71
2.31	Textausgabe mit Bitmaps	74
2.32	Kollisionserkennung mit Rechtecken	75
2.33	Kollisionserkennung mit Kreisen	76
2.34	Vier Sprites	77
2.35	Rechtecksprüfung (Montage)	77
2.36	Kreisprüfung (Montage)	77

2.37	Maskenprüfung (Montage)	77
2.38	Feuerball ohne Zeitsteuerung	83
2.39	Feuerball mit Zeitsteuerung	87
2.40	Animation einer Katze: Einzelsprites	90
2.41	Animation einer Explosion: Einzelsprites	96
2.42	Mausaktionen	98
2.43	Sound: Stereoeffekt	108
2.44	Dirty Sprite – Demo-Spiel	115
2.45	Performancevergleich mit Testkonfiguration 2	122
2.46	Testkonfiguration 2 mit privatem PC	123
2.47	Testkonfiguration 2 mit Surface Pro	123
2.48	Testkonfiguration 2 mit privatem PC und reduzierter fps	123
2.49	Bestätigung der Unterschiede	123
2.50	Selbst erstellte Events	128
3.1	Pong: der Hintergrund	136
3.2	Pong: mit Schläger, Ball und Punktstand	144
3.3	Pong: Schlägerfarbe markiert KI-Modus (links KI, rechts manuell)	147
3.4	Pong: Hilfe-Bildschirm	152
3.5	Bubbles: Hintergrundbild (aquarium.png)	153
3.6	Blase	155
3.7	Bubbles: Die Blasen haben beim Start einen Mindestabstand	158
3.8	Bubbles: Die Blasen sind gewachsen/verwachsen	162
3.9	Kollisionserkennung: Punkt innerhalb des Kreises?	162
3.10	Bubbles: Ausgabe Punktteststand	166
3.11	Bubbles – Kollision mit dem Rand	168
3.12	Bubbles – Kollision der Blasen	168
3.13	Blase 2	170
3.14	Bubbles: Kollision anzeigen	172
3.15	Bubbles: Pausenbildschirm	173
3.16	Bubbles: Neustartbildschirm	176

Glossar

äquidistant Der Abstand von Elementen ist immer der gleiche. Bei gleich großen Elementen bedeutet dies, dass der Platz zwischen diesen immer gleich ist. Bei nicht gleichgroßen Elementen muss es einen Bezugspunkt geben. Sollen die Mittelpunkte der Elemente immer die gleiche Distanz haben, oder sollen der rechte Rand des einen immer den gleichen Abstand zum linken Rand des nächsten haben? Auch wird zwischen horizontaler und vertikaler Äquidistanz unterschieden. [23](#)

Alpha-Kanal Für jedes Pixel eines Bildes werden Farbinformationen meist im [RGB](#)-Format abgespeichert: R-Kanal, G-Kanal und B-Kanal. Durch eine zusätzliche Information kann man noch angeben, wie durchscheinend das Pixel sein soll. Diese zusätzlich Informationen nennt man den Alpha-Kanal. [11](#)

Array Eine Datenstruktur, welche Werte unter einem einzigartigen Index (meist eine positive ganze Zahl) ablegt. Im engeren Sinne enthalten Array immer nur Elemente des gleichen Datentyps. Bei Sprachen wie PHP oder Python gilt das nicht. [73](#)

Bitmap Der Begriff Bitmap hat hier zwei Bedeutungsebenen: Allgemein meint er Farb- und Transparenzinformationen eines Bildes in einer Datei. Typische Beispiele sind Dateien im Format [Joint Photographic Experts Group \(jpeg\)](#), [Portable Network Graphics \(png\)](#) oder [Windows Bitmap Format \(bmp\)](#). Im Speziellen ist damit das Bitmap-Dateiformat zur Bildspeicherung (Windows Bitmap, BMP) gemeint. [6](#)

Boss-Taste Bei Betätigung der Boss-Taste wird das Spiel ohne Rückfragen so schnell wie möglich beendet. Der Boss kommt herein, der Lehrer steht hinter einem, ... [54](#)

Dictionary Eine Datenstruktur, welche Werte unter einem einzigartigen Schlüssel ablegt. Andere Namen sind: Zuordnungstabelle, assoziatives Array, Hashtable. [71](#)

Doublebuffer Dies ist ein zweiter Speicherbereich, der genauso groß ist wie der Bildschirmspeicher. Wird jetzt etwas auf die Spielfläche gezeichnet, passiert dies zunächst auf diesem zweiten Speicher. Erst wenn alle Spielelemente ihr neues Aussehen gemalt haben, wird mit einem Schlag der alte Bildschirmspeicher mit dem zweiten ausgetauscht. Bei bestimmten Hardware- oder Grafikkonfigurationen kann es passieren, dass der Bildschirmspeicher neu gemalt wird, obwohl das Spiel noch nicht alle neuen Zustände abgebildet hat. Dadurch können hässliche Artefakte entstehen. Durch das Doublebuffering wird dieser Effekt vermieden. [6](#)

DTP-Punkt Maßeinheit für Schriftgrößen. [188](#)

Endlosschleife In der Informatik ist eine Endlosschleife eine Folge von Anweisungen, die sich immer wiederholt und für die es kein definiertes Abbruchkriterium gibt. In den meisten Fällen sind Endlosschleifen nicht gewollt und damit ein Fehler in der Anwendung. Sie entstehen oft durch fehlerhafte Schleifenbedingungen. Endlosschleifen werden manchmal aber auch gezielt eingesetzt: `while True:` [158](#)

Event Ein Event dient in der Softwaretechnik zur Steuerung des Programmflusses. Das Programm wird nicht linear durchlaufen, sondern es werden spezielle Ereignisbehandlungsrouterien (engl. listener, observer, event handler) immer dann ausgeführt, wenn ein bestimmtes Ereignis auftritt. Ereignisorientierte Programmierung gehört zu den parallelen Programmiertechniken, hat also deren Vor- und Nachteile (Quelle: Wikipedia). [127](#)

fade Kommt vom englischen *to fade* für *verblassen*. In der Musik und bei Grafiken unterscheidet man einen *fadein* und einen *fadeout*. Bei einem *fadein* erscheint das Bild langsam bzw. wird die Lautstärke bei 0 beginnend auf die Ziellautstärke erhöht. Ein *fadeout* tut das Gegenteil. [105](#)

Flag Eine meist boolsche Variable, die eine Operation/Schleife ein- und ausschaltet. [7](#)

Fließkommazahl Bei einer Fließkommazahl werden Zahlen als Summen von Zweierpotenzen dargestellt, wobei der Exponent auch negativ sein kann. Beispiel: $6,75 = 2^2 + 2^1 - 1 + 2^{-2}$. Da der Speicherplatz beschränkt ist oder es für bestimmte Zahlen keine endliche Darstellung gibt, bricht die Summenbildung zu irgendeinem Zeitpunkt ab. Die dabei nicht mehr berücksichtigten Summanden führen zu den Rundungsfehlern. [38](#)

Font In digitaler Form vorhandene Information über einen Zeichensatz. Er ist meist in einem dieser drei Formate verfügbar: als Bitmap, als Vektorgrafik oder als Beschreibung. [62](#)

frames per second Maximale Anzahl der Bilder pro Sekunde. [8](#), [188](#)

Framework In der Informatik ist damit eine Arbeitsumgebung gemeint. Dies können einzelne Klassen, Funktionsbibliotheken oder ganze **Integrated Development Environment (IDE)** sein. [46](#)

Funktion Eine Funktion ist in der Programmierung ein Anweisungsblock mit einem Namen. Sie können Parametersätze haben und Ergebnisse zurückliefern. In der Regel gilt dabei das Prinzip, dass alle Werte innerhalb der Funktion lokal sind. [5](#)

Ganzzahl Bei einer Ganzkommazahl werden Zahlen als Summen von Zweierpotenzen dargestellt, wobei der Exponent Null oder positiv ist. Beispiel: $17 = 2^4 + 2^0$. Der Wertebereich ist durch den Speicherplatz, den man einer ganzen Zahl zur Verfügung stellt, definiert. Stehen n Bits zur Verfügung, können ohne Vorzeichen Zahlen im Bereich $[0, 2^n - 1]$ und mit Vorzeichen im Bereich $[2^{n-1}, 2^{n-1} - 1]$ dargestellt werden. [39](#)

Generative Pre-trained Transformer ChatGPT ist ein Prototyp eines Chatbots, also eines textbasierten Dialogsystems als Benutzerschnittstelle, der auf maschinellem Lernen beruht. Den Chatbot entwickelte das US-amerikanische Unternehmen OpenAI, das ihn im November 2022 veröffentlichte. (Quelle: Wikipedia). [188](#)

Grad (°) Maßeinheit für einen Winkel. Der Vollkreis hat dabei 360°. [102](#)

Hauptprogrammschleife Jedes nichtriviale Programm muss entscheiden, ob es noch weiterlaufen soll, oder ob die Verarbeitung beendet werden kann. Falls die Verarbeitung noch nicht beendet werden kann oder soll, muss mit der Benutzerinteraktion oder anderen Programmfunctionen fortgefahren werden und zwar solange, bis das Programm beendet werden kann oder soll. Dies wird in der Regel durch eine Hauptprogrammschleife gesteuert. Beispiele: Das Betriebssystem läuft, solange bis es heruntergefahren wird. Die Windows-Anwendung läuft, bis ALT+F4 betätigt wurde. [6](#)

Integrated Development Environment Integrierte Entwicklungsumgebung. Diese heißen *integriert*, da sie nicht nur einen Compiler und Linker enthalten, sondern auch einen Editor, Debugger, Profiler etc. [183](#), [188](#)

Joint Photographic Experts Group Verlustbehaftete komprimierte Bildinformationen. [182](#), [188](#)

Klasse Eine Klasse beschreibt die Attribute und die Methoden (Funktionen) einer inhaltlich abgeschlossenen Programmierereinheit. In der Praxis gibt es viele Varianten von Klassen, aber im Prinzip wird definiert, welche Informationen eine Klasse ausmacht (z.B. Marke, Farbe und Baujahr eines Autos) und was man mit einem Objekt der Klasse alles tun kann (z.B. beschleunigen, kaufen und tanken beim einem Auto). Die Informationen werden *Attribute* genannt und die Möglichkeiten *Methoden* oder *member funtions*. [5](#)

Kollisionserkennung Überprüfung, ob zwei Bitmaps sich in irgendeiner einer Art und Weise *berühren*. In Pygame nutzen wir drei Arten der Kollisionserkennung: Schneiden sich die umgebenden Rechtecke der Bitmaps, schneiden sich die Innenkreise der Bitmaps und haben nicht-transparente Pixel der Bitmaps die selbe Koordinate. [27](#)

Konstante Eine Konstante ist ein Wert, der zur Laufzeit eines Programmes nicht mehr geändert werden kann. In vielen Programmiersprachen können Variablen durch Schlüsselwörter wie `const` als Konstanten – also Unveränderlichen – deklariert werden. Direkte beispielsweise Zahlen- oder Stringangaben im Quelltext sind ebenfalls Konstanten. [5](#)

Linienzug Eine Folge miteinander verbundener Linien. Wird meist durch eine Folge von Punkten definiert. Bei einem geschlossenen Linienzug spricht man von einem **Polygon**. [12](#), [185](#)

Maske Ein Maske (engl. mask) ist ein Bitmap, welches die wichtigen von den unwichtigen Pixel eines Sprites unterscheidbar macht. Bei Sprites mit Transparenzen kann die Maske einfach dadurch ermittelt werden, dass alle transparenten Pixel unwichtig sind. Um Speicherplatz und Rechenzeit zu sparen, werden die Masken oft nicht in den üblichen Bitmap-Formaten abgelegt, sondern Bit für Bit. Ein Byte kann also die Maskeninformation für 8 Pixel kodieren. [76](#)

Message Queue Warteschlange des Betriebssystem zur Verwaltung von Ereignissen, die vom System erzeugt oder empfangen wurden. Laufende Anwendungen können diese Nachrichten für sich deklarieren und aus der Warteschlange entnehmen. [6](#)

Millisekunden Der 1/1000 Teil einer Sekunde. [87](#), [188](#)

mp3 Abkürzung von *ISO MPEG Audio Layer 3*. Eine im wesentlichen vom deutschen Elektrotechnikingenieur und Mathematiker Karlheinz Brandenburg entwickeltes Kodierungs- und Kompressionsverfahren von Sound und Musik. [105](#)

Namensraum Innerhalb eines Namensraums müssen alle Namen für Klassen, Funktionen und Konstanten eindeutig sein. In der Regel werden Namensräume in Python anhand der Module und Pakete definiert. [5](#)

Objektorientiert Die Analyse, das Design oder die Implementierung entspricht den allgemeinen Vorgaben der Objektorientierung. [188](#)

ogg Kodierung von Sound-Dateien. Kommt vom englischen *to ogg*. Ziel war eine lizenfreie, einfache und effiziente Kodierung von Sound. [105](#)

Pixel Die kleinste bei gegebener Auflösung ansteuerbare Bildschirmfläche. [6](#), [188](#)

Polygon Ein geschlossener [Linienzug](#). Wird meist durch eine Folge von Punkten definiert, wobei der letzte Punkt mit dem ersten verbunden wird. [12](#), [184](#)

Portable Network Graphics Verlustfrei komprimierte Bildinformationen. [182](#), [188](#)

Pygame Pygame ist ein Verbund von Modulen, der die Entwicklung von Computerspielen in Python unterstützt. [4](#)

Pylance Pylance ist die Standard Python-Erweiterung von Visual Code zur Unterstützung der Python-Programmierung. Seine wentsentlichen Features sind die Typüberwachung und Autovervollständigung. [101](#)

Python Python ist eine höhere Interpretersprache mit prozeduralen und objektorientierten Paradigmen. Sie wurde 1991 von Guido van Rossum entwickelt und erfreut sich derzeit größter Beliebtheit. [4](#)

Radiant (rad) Maßeinheit für einen Winkel. Der Vollkreis hat dabei 2π rad. [102](#)

Red Green Blue Additive Farbkodierung. [188](#)

Rendern Das Erzeugen eines Bildes – meist in Bitmap-Format – aus einer Bildbeschreibungsangabe. [62](#)

Satz des Pythagoras In einem rechtwinkligen Dreieck ist die Summe der Kathetenquadrate gleich dem Hypotenusenquadrat: $c^2 = \sqrt{a^2 + b^2}$. Der Satz ist nach dem Mathematiker *Pythagoras von Samos* (um 570 v.Chr. bis um 510 v.Chr.) benannt. [162](#)

Semantik Bedeutung einer Angabe. Wird meist in Abgrenzung zu [Syntax](#) einer Angabe verwendet. [12](#), [186](#)

Signatur Die Signatur einer Funktion/Methode beschreibt die formalen Eigenschaften, die von einer Funktion/Methode von außen sichtbar ist. Dazu gehören die Sichtbarkeit, der Rückgabetyp, der Name und die Übergabeparameter. [52](#)

Simple Direct Media Layer Eine plattformunabhängige API für die Programmierung von Grafiken, Sounds und Eingabegräte. [5](#), [188](#)

Single Responsibility Principle Jede Klasse / jede Funktion sollte nur eine Verantwortlichkeit haben. Die Klasse / die Funktion sollte sich auf diese Aufgabe konzentrieren. Kapseln Sie eine Lösung in eine Klasse oder eine Methode. [50](#), [188](#)

Singleton Ein Design-Pattern, welches sicherstellt, dass es immer nur ein Objekt einer Klasse gibt. Dieses wird dann meist (halb-)öffentlich zur Verfügung gestellt. Das Singleton ist wegen seiner konzeptionellen Nähe zu globalen Variablen umstritten. [111](#)

Slicing Eine Technik, mit deren Hilfe man Teilmengen aus Strings oder Arrays bequem ausschneiden oder extrahieren kann. [73](#)

Solid-State-Drive Festspeicherplattentechnologie, welche nicht auf magnetische Prinzipien, sondern auf Halbleitertechnik basiert. [188](#)

Sprite Ein Grafikobjekt, welches auf einem Hintergrund platziert wird und meist auch Eigenschaften hat, die über die reine Anzeige hinausgehen. So können Sprites sich oft bewegen oder werden animiert oder lösen bei Kontakt eine Reaktion aus. Üblicherweise meint man damit immer 2D-Objekte. Andere Namen sind *moveable object (MOB)* oder *blitter object (BOB)*. [21](#), [186](#)

Spritelib Meist eine Grafikdatei im Bitmap-Format, welches viele einzelne [Sprites](#) enthält. [69](#)

Stereofonie Verfahren um mit mehr als einer Schallquelle einen mehrdimensionalen Schalleindruck zu erzeugen. [188](#)

Syntax Form oder Grammatik einer Angabe. Wird meist in Abgrenzung zur [Semantik](#) einer Angabe verwendet. [186](#)

Toggling In der Informatik bedeutet dies, dass der Wert einer Booleschen Variable von `True` nach `False` bzw. von `False` nach `True` wechselt: *to toggle = umschalten*. [174](#)

Trade-off Jeder Vorteil wird durch einen Nachteil erkauft. Algorithmisch muss dann anhand der Datenlage abgewägt werden, ob in der Gesamtbetrachtung der Nutzen die Kosten überwiegt. Beispiel: Durch die Verwendung von Indizes werden Zugriffe auf Datenbankinhalte dramatisch beschleunigt (Nutzen). Um diese Beschleunigung zu erreichen, müssen Dateien angelegt werden, und das Anlegen, Ändern

und Löschen von Daten wird langsamer, da diese Dateien dann mit gepflegt werden müssen (Kosten). [121](#)

True Type Font Die Schriftinformation wird nicht im Bitmap-Format, sondern in einer Art Vektorgrafikformat abgespeichert. Dadurch lassen sich *beliebige* Schriftgrößen generieren. [188](#)

Umgebungsvariable Dies sind Variablen, die nicht vom Programm, sondern von der Programmumgebung verwaltet werden. Die Programmumgebung kann das Betriebssystem sein, aber auch eine Server. Über Umgebungsvariablen kann die Umgebung mit meinem Programm Informationen austauschen. In unserem Beispiel wird der Fensterverwaltung bzw. dem Betriebssystem mitgeteilt, an welcher Koordinate die linke obere Ecke des Fensters auf dem Bildschirm erscheinen soll. [5](#)

Unicode Ein Verfahren zur Kodierung von Zeichen und Symbolen. Gängige Umsetzungen sind UTF-8, UTF-16 und UTF-32. [73](#)

Universal Serial Bus Bitserielles Datenübertragungsprotokoll. [188](#)

Windows Bitmap Format Bildinformationen im Windows Bitmap-Format. [182](#), [188](#)

Akronyme

bmp Windows Bitmap Format. [181](#), [187](#), *Glossar: Windows Bitmap Format*

ChatGPT Generative Pre-trained Transformer. [135](#), [187](#), *Glossar: Generative Pre-trained Transformer*

fps frames per second. [8](#), [187](#), *Glossar: frames per second*

IDE Integrated Development Environment. [182](#), [187](#), *Glossar: Integrated Development Environment*

jpeg Joint Photographic Experts Group. [181](#), [187](#), *Glossar: Joint Photographic Experts Group*

ms Millisekunden. [87](#), [187](#), *Glossar: Millisekunden*

OO Objektorientiert. [62](#), [187](#), *Glossar: Objektorientiert*

png Portable Network Graphics. [181](#), [187](#), *Glossar: Portable Network Graphics*

pt DTP-Punkt. [62](#), [187](#), *Glossar: DTP-Punkt*

px Pixel. [6](#), [187](#), *Glossar: Pixel*

RGB Red Green Blue. [7](#), [181](#), [187](#), *Glossar: Red Green Blue*

SDL Simple Direct Media Layer. [5](#), [187](#), *Glossar: Simple Direct Media Layer*

SRP Single Responsibility Principle. [50](#), [187](#), *Glossar: Single Responsibility Principle*

SSD Solid-State-Drive. [6](#), [187](#), *Glossar: Solid-State-Drive*

Stereo Stereofonie. [108](#), [187](#), *Glossar: Stereofonie*

ttf True Type Font. [66](#), [187](#), *Glossar: True Type Font*

USB Universal Serial Bus. [6](#), [187](#), *Glossar: Universal Serial Bus*

Index

- äquidistant, 23
- __main__, 51
- Alpha-Kanal, 11, 23, 150
- Animation, 90
- assoziatives Array, 182
- Ball, 140
- Bitmap, 21
 - ausgeben, 21
 - bewegen, 29
 - laden, 21
- Blasen erscheinen, 155
- Blasen zerplatzen, 163
- Blasenanzahl, 159
- Blasenwachstum, 159
- Bubbles, 152
- Dictionary, 54, 71
- Dirty Sprite, 115
- Doublebuffer, 6
- deltatime, 33, 35, 41, 84
- Ereignis, 56
- Event, 127
 - type, 128
- Farbe
 - Alpha-Kanal, 11
 - Information, 11
 - Namen, 12
- Farbnamen, 19, 116, 125
- Flag, 6, 175
- Font, 62
 - farbe, 63
 - größe, 63
- Frame, 115
- Framerate, 123
- fade, 105
- float, 39
- Geschwindigkeit, 31
- Grafikprimitive, 11
 - Kreis, 11, 12
 - Linie, 11, 12
 - Polygon, 12
 - Punkt, 11, 13
 - Rechteck, 12
- gravity, 15
- Hashtable, 182
- Hauptprogrammschleife, 6
- Hintergrundmusik, 105
- int, 39
- Kanal, 109
- Kollision, 75
- Kollision anzeigen, 170
- Kollisionserkennung
 - Kreis, 75
 - Pixel, 76
 - Rechteck, 75
- Kreis, 11, 12
- Linie, 11, 12
- Maske, 76
- Maus, 98
- Mauscursor, 162
- Mausrad, 99
- main loop, 6
- mp3, 105
- Neustart, 175

ogg, 105
Pause, 173
Polygon, 12
Pong, 135
Punkt, 11, 13
Punkte, 142, 145, 146, 148, 150
Punktestand, 164
Pythagoras, Satz von, 162

Rechteck, 12, 156
Rendern, 62
Richtung, 31
Richtungswechsel, 32

SDL_VIDEO_WINDOW_POS, 5, 9
Schläger, 137
Schwerkraft, 15
Shift-Taste, 56
Sound, 177
Soundausgaben, 104
Soundeffekte, 105
Spielende, 167
Sprite, 46
 self.image, 46
 self.rect, 46
Standardfunktionalität, 135, 152
screen, 6
self.mask, 78
self.radius, 78, 156
self.rect, 78, 156

Tastatur, 54
 -konstanten, 57
 -schalter, 60
Timer, 88, 156, 169
Transparenz, 11, 23
time
 time(), 40

Zeitanpassungen, 168
Zeitsteuerung, 83
Zuordnungstabelle, 182

Index für den Namensraum pygame

Color, [11](#), [19](#)
Event, [56](#)
KEYDOWN, [56](#), [57](#)
KEYUP, [56](#), [57](#), [173](#)
KEY, [57](#)
KMOD_LSHIFT, [56](#)
K_DOWN, [56](#)
K_ESCAPE, [56](#)
K_LEFT, [56](#)
K_RIGHT, [56](#)
K_SPACE, [56](#)
K_UP, [56](#)
MOUSEBUTTONDOWN, [99](#), [102](#), [164](#)
MOUSEBUTTONUP, [99](#), [102](#), [173](#)
NUMEVENTS, [129](#), [134](#)
QUIT, [7](#), [9](#)
Rect, [165](#)
 collidepoint(), [100](#), [102](#)
 height, [156](#)
 left, [156](#)
 move(), [52](#)
 move_ip(), [52](#)
 top, [156](#)
 width, [156](#)
SYSTEM_CURSOR_CROSSHAIR, [163](#)
SYSTEM_CURSOR_HAND, [163](#)
Surface, [6](#), [136](#)
 blit(), [21](#), [28](#), [31](#)
 convert(), [22](#), [28](#)
 convert_alpha(), [23](#), [28](#)
 fill(), [7](#), [10](#)
 get_rect(), [30](#), [45](#), [78](#)
 set_at(), [13](#), [20](#)
 set_colorkey(), [23](#), [28](#), [78](#)
 subsurface(), [66](#), [68](#), [72](#), [74](#)
USEREVENT, [129](#), [134](#)
clock
 tick(), [38](#), [40](#)
display
 flip(), [6](#), [9](#), [120](#)
 get_surface(), [6](#), [9](#), [154](#)
 get_window_size(), [16](#), [20](#)
 set_caption(), [6](#), [9](#), [154](#)
 set_mode(), [6](#), [9](#), [154](#)
 update(), [7](#), [9](#), [120](#), [125](#)
draw
 circle(), [12](#), [19](#)
 line(), [12](#), [19](#)
 lines(), [12](#), [19](#)
 polygon(), [12](#), [20](#)
 rect(), [12](#), [20](#)
event
 get(), [134](#)
event
 Event(), [143](#)
 Event, [129](#), [134](#)
 unicode, [73](#), [74](#)
 button, [99](#), [102](#)
 get(), [7](#), [9](#)
 key, [56](#)
 mod, [56](#)
 post(), [129](#), [134](#)
 pos, [99](#)
 type, [7](#), [9](#)
font
 Font, [62](#), [68](#)
 render(), [63](#), [68](#)
 get_default_font(), [62](#), [68](#)
 get_fonts(), [66](#), [68](#)
 match_font(), [66](#), [68](#)

gfxdraw
 pixel(), 13, 20
image, 27
 load(), 21, 28
init(), 6, 9, 34, 105, 154
key, 56
mask
 from_surface(), 78, 82
math
 Vector2D, 54
 Vector2, 39, 45
 Vector3, 45
mixer
 Channel, 109, 113
 play(), 110, 113
 set_volume(), 111, 113
 Sound, 105, 109, 114, 178
 get_volume(), 105, 114
 play(), 106, 109, 114, 179
 set_volume(), 111, 114
 find_channel(), 109, 114
 init(), 6, 104, 114
music
 fadeout(), 106, 114
 get_volume(), 105, 114
 load(), 105, 114
 pause(), 107, 114
 play(), 105, 106, 114
 set_volume(), 105, 107, 111, 114
 unpause(), 107, 114
mouse
 get_pos(), 13, 20, 100, 102, 163, 164
 get_pressed(), 13, 20
 set_visible(), 100, 102
quit(), 7, 9
rect
 FRect, 29, 39, 45
 bottomright, 29
 bottom, 29
 centerx, 29
 centery, 29
 center, 29
 height, 29
 left, 29

move(), 45, 46
move_ip(), 47
right, 29
size, 29
topleft, 29
top, 29
width, 29
Rect, 12, 20, 29, 45
 bottomright, 29
 bottom, 29
 centerx, 29
 centery, 29
 center, 29
 contains(), 167
 height, 29
 left, 29
 move(), 33, 45
 right, 29
 size, 29
 topleft, 29
 top, 29
 width, 29
sprite
 DirtySprite, 118, 125
 dirty, 115, 118, 125
 GroupSingle, 50, 52, 136
 sprite, 50, 52
 Group, 49, 52, 119, 139
 sprites(), 167
 LayeredDirty, 119, 125, 166
 clear(), 120, 125, 157
 draw(), 120, 125
 set_timing_threshold(), 121, 126
 set_timing_threshold(), 126
 Sprite, 46, 53
 kill(), 84, 89, 96, 164
 update(), 52
 collide_circle(), 81, 82, 158
 collide_mask(), 81, 82
 collide_rect(), 48, 53, 81, 82, 145
 spritecollide(), 49, 53, 81, 82, 158
 DirtySprite, 152
 LayeredGroup, 152
time

Clock, [8](#), [10](#), [154](#)
 tick(), [8](#), [10](#), [84](#)
 tick_busy_loop(), [8](#), [10](#)
 get_ticks(), [34](#), [45](#), [87](#), [89](#)
 set_timer(), [133](#), [134](#)
 wait(), [172](#)
transform
 rotate(), [102](#), [102](#)
 scale(), [22](#), [28](#), [159](#)