

Einführung in die 2D-Spieleprogrammierung mit Pygame

Ralf Adams (TBS1, Bochum)

Version 0.7 vom 12. Dezember 2022

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: Kreise	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
3	Beispielprojekte	126
3.1	Bubbles	126
3.1.1	Requirement 1: Standards	126
3.1.2	Requirement 2: Blasen erscheinen	129
3.1.3	Requirement 3: Blasenanzahl	132
3.1.4	Requirement 4: Blasenwachstum	133
3.1.5	Requirement 5: Mauscursor	135

3.1.6	Requirement 6: Blasen zerplatzen	137
3.1.7	Requirement 7: Punktstand	138
3.1.8	Requirement 8: Spielende	141
3.1.9	Requirement 9: Zeitanpassungen	143
3.1.10	Requirement 10: Kollision anzeigen	144
3.1.11	Requirement 11: Pause	147
3.1.12	Requirement 12: Neustart	149
3.1.13	Requirement 13: Sound	152

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 wird ein Spielprojekt vollständig durchprogrammiert und damit der Einsatz der Techniken verdeutlicht.

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

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
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.

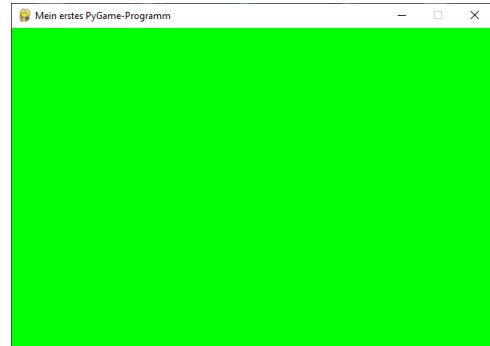


Abb. 2.1: Spielfläche

`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 werden uns nur mit Spielen beschäftigen, die unmittelbar auf dem Desktop laufen. Oder anders herum: Wir werden keinen Game-Server implementieren. Daher brauchen unsere Spiele eine *Spielfläche*/ein Fenster innerhalb dessen sich alles abspielt. Die Funktion `pygame.display.set_mode()` liefert mir eine solche 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 und picke die für mich interessanten heraus.

`event.get()`

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.

`event.type`
`pygame.QUIT`

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.

`RGB`

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.

`update()`

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.

`quit()`

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

fps

zu stellen, um die Ereignisse einzusammeln, die neuen Zustände zu berechnen und erst dann die Bildschirmausgabe zu generieren. Die Bildschirmausgabe selbst sollte 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        clock.tick(60)                            # Taktung auf 60 fps
17        for event in pygame.event.get():
18            if event.type == pygame.QUIT:
19                running = False
20            screen.fill((0, 255, 0))
21            pygame.display.flip()
22
23    pygame.quit()
24
25
26 if __name__ == '__main__':
27    main()

```

Clock

tick()

tick_busy_loop()

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 16. 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.

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. Schleifendurchläufe takten.
 2. Events abfragen.
 3. Bildschirminhalt ausgeben.
- Beim Verlassen das Pygame-System stoppen.

Es wurden folgende Pygame-Elemente eingeführt:

- `import pygame:`
<https://www.pygame.org/docs/tut/ImportInit.html>
- `os.environ['SDL_VIDEO_WINDOW_POS']:`
<https://docs.python.org/3/library/os.html#os.environ>
- `pygame.init():`
<https://www.pygame.org/docs/ref/pygame.html#pygame.init>
- `pygame.quit():`
<https://www.pygame.org/docs/ref/pygame.html#pygame.quit>
- `pygame.QUIT:`
<https://www.pygame.org/docs/ref/event.html#pygame.event.EventType.type>
- `pygame.display.flip():`
<https://www.pygame.org/docs/ref/display.html#pygame.display.flip>
- `pygame.display.get_surface():`
https://www.pygame.org/docs/ref/display.html#pygame.display.get_surface
- `pygame.display.set_caption():`
https://www.pygame.org/docs/ref/display.html#pygame.display.set_caption
- `pygame.display.set_mode():`
https://www.pygame.org/docs/ref/display.html#pygame.display.set_mode
- `pygame.display.update():`
<https://www.pygame.org/docs/ref/display.html#pygame.display.update>
- `pygame.event.get():`
<https://www.pygame.org/docs/ref/event.html#pygame.event.get>
- `pygame.event.type:`
<https://www.pygame.org/docs/ref/event.html#pygame.event.EventType.type>

- `pygame.time.Clock`:
<https://www.pygame.org/docs/ref/time.html#pygame.time.Clock>
- `pygame.Surface.fill()`:
<https://www.pygame.org/docs/ref/surface.html#pygame.Surface.fill>
- `pygame.time.Clock.tick()`:
<https://www.pygame.org/docs/ref/time.html#pygame.time.Clock.tick>
- `pygame.time.Clock.tick_busy_loop()`:
https://www.pygame.org/docs/ref/time.html#pygame.time.Clock.tick_busy_loop

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) # Ein 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        clock.tick(60)
24        for event in pygame.event.get():
25            if event.type == pygame.QUIT:
26                running = False
27            screen.fill(mycgrey)
28            pygame.draw.rect(screen, "red", myrectangle1) # Gefülltes Rechteck
29            pygame.draw.rect(screen, "red", myrectangle2, 3, 10) # Anderes Rechteck
30            pygame.draw.polygon(screen, "green", points1) # Gefülltes Polygon
31            pygame.draw.polygon(screen, "green", points2, 2) # Nicht gefülltes Polygon
32            pygame.draw.line(screen, "red", (5, 230), (240, 230), 3) # Linie
33            pygame.draw.circle(screen, "blue", (40, 150), 30) # Gefüllter Kreis
34            pygame.draw.circle(screen, "blue", (110, 150), 30, 2) # Nicht gefüllter Kreis
35            pygame.draw.circle(screen, "blue", (180, 150), 30, 5, True) # Kreisbogenschnitt
36            for i in range(255):
37                for j in range(255):
38                    screen.set_at((265+i, 10+j), (255, i, j)) # Punkte Variante 1
39                    screen.fill((i, j, 255), ((10+i, 265+j), (1, 1))) # Variante 2
40                    pygame.gfxdraw.pixel(screen, 265+i, 265+j, (i, 255, j)) # Variante 3
41
42            pygame.display.flip()
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.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 28 zeichnet `pygame.draw.rect()` ein gefülltes Rechteck. Die [Semantik](#) der Parameter sollte selbsterklärend sein. Anders der Aufruf von Zeile 29. 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 30) und ungefüllte (Zeile 31) 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 32.

circle()

Ein Kreis wird durch zwei Angaben definiert: Mittelpunkt und Radius. In Zeile 33 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 34). Interessant ist der Kreisbogenausschnitt in Zeile 35. Hier

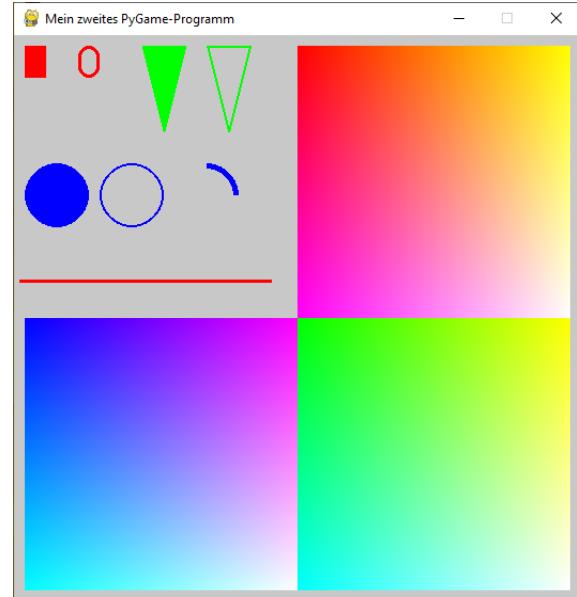


Abb. 2.4: Einige Grafikprimitive

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 38 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 39). Ein Möglichkeit einen Pixel über eine Grafikbibliothek zu setzen, ist die experimentelle `gfxdraw`-Umgebung. In Zeile 40 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: Kreise

Man kann mit Grafikprimitiven dynamische Effekte einbauen, wie beispielsweise Partikelschwärme. Ich will hier mal ein super einfaches Beispiel für einen mausgesteuerten Schwarm 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 35 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.

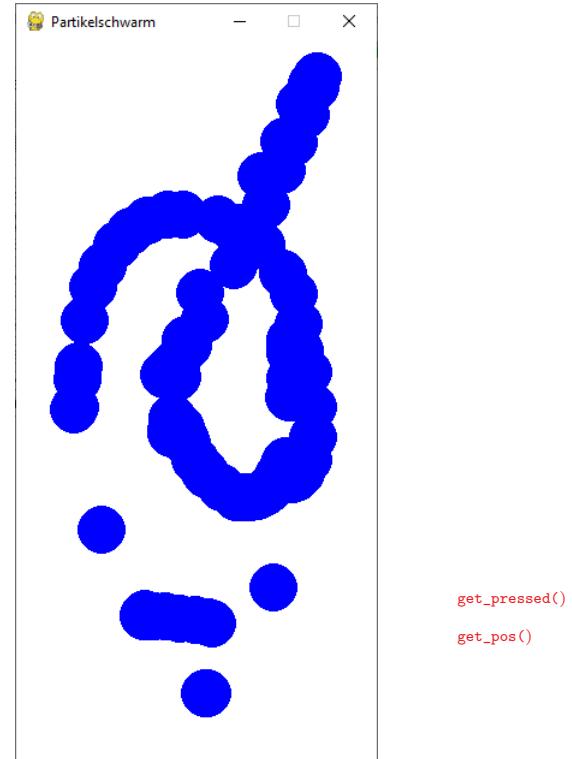


Abb. 2.5: Sicher keine Partikelfontaine

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         clock.tick(60)
31         for event in pygame.event.get():
32             if event.type == pygame.QUIT:
33                 running = False
34
35             if pygame.mouse.get_pressed()[0]: # Linke Maustaste?
36                 circles.append(Circle(pygame.mouse.get_pos()))
37
38             screen.fill("white")
39             for p in circles:
40                 p.draw()
41
42             pygame.display.flip()
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

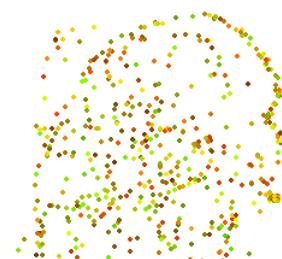


Abb. 2.6: Version 2

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()`

```

45     for p in circles:
46         p.update()
47
48     screen.fill("white")
49     for p in circles:
50         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. 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.

get_window_size()



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 52 wird nicht ein Partikel erzeugt, sondern immer gleich 5.

In Zeile 55 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 58). Wenn ja, wird dieser Partikel in die Liste `todelete` aufgenommen. Nach Beendigung der Update-Schleife werden die zu löschen Partikel ab Zeile 60 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         clock.tick(60)
47         for event in pygame.event.get():
48             if event.type == pygame.QUIT:
49                 running = False
50
51             if pygame.mouse.get_pressed()[0]:
52                 for i in range(5):           # 5 Partikel gleichzeitig
53                     circles.append(Circle(pygame.mouse.get_pos()))
54
55             todelete = []                 # Zwischenspeicher
56             for p in circles:
57                 p.update()
58                 if p.todelete:           # Zu löschen?
59                     todelete.append(p)
60             for p in todelete:          # Löschen
61                 circles.remove(p)
62
63             screen.fill("white")
64             for p in circles:
65                 p.draw()
66
67             pygame.display.flip()

```

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
10
11     def __init__(self, pos) -> None:
12         self.posx = pos[0] + randint(-4, 4)
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
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:
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     clock.tick(60)
51     for event in pygame.event.get():
52         if event.type == pygame.QUIT:
53             running = False
54
55         if pygame.mouse.get_pressed()[0]:
56             for i in range(5):
57                 circles.append(Circle(pygame.mouse.get_pos()))
58
59         todelete = []
60         for p in circles:
61             p.update()
62             if p.todelete:
63                 todelete.append(p)
64         for p in todelete:
65             circles.remove(p)
66
67         screen.fill("white")
68         for p in circles:
69             p.draw()
70
71     pygame.display.flip()
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:
http://www.pygame.org/docs/ref/color_list.html
- `import pygame.gfxdraw`:
<https://www.pygame.org/docs/ref/gfxdraw.html>
- `pygame.Color`:
<https://www.pygame.org/docs/ref/color.html>
- `pygame.draw.circle()`:
<https://www.pygame.org/docs/ref/draw.html#pygame.draw.circle>
- `pygame.draw.line()`:
<https://www.pygame.org/docs/ref/draw.html#pygame.draw.line>
- `pygame.draw.lines()`:
<https://www.pygame.org/docs/ref/draw.html#pygame.draw.lines>
- `pygame.draw.polygon()`:
<https://www.pygame.org/docs/ref/draw.html#pygame.draw.polygon>

- `pygame.draw.rect()`:
<https://www.pygame.org/docs/ref/draw.html#pygame.draw.rect>
- `pygame.display.get_window_size()`:
https://www.pygame.org/docs/ref/display.html#pygame.display.get_window_size
- `pygame.gfxdraw.pixel()`:
<https://www.pygame.org/docs/ref/gfxdraw.html#pygame.gfxdraw.pixel>
- `pygame.mouse.get_pos()`:
https://www.pygame.org/docs/ref/mouse.html#pygame.mouse.get_pos
- `pygame.mouse.get_pressed()`:
https://www.pygame.org/docs/ref/mouse.html#pygame.mouse.get_pressed
- `pygame.Rect`:
<https://www.pygame.org/docs/ref/rect.html>
- `pygame.Surface.set_at()`:
https://www.pygame.org/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         clock.tick(Settings.FPS)
26         for event in pygame.event.get():
27             if event.type == pygame.QUIT:
28                 running = False
29
30         screen.fill("white")
31         screen.blit(enemy_image, (10, 10))                                # Bitmap ausgeben
32         screen.blit(defender_image, (10, 80))
33         pygame.display.flip()
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 31). 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-

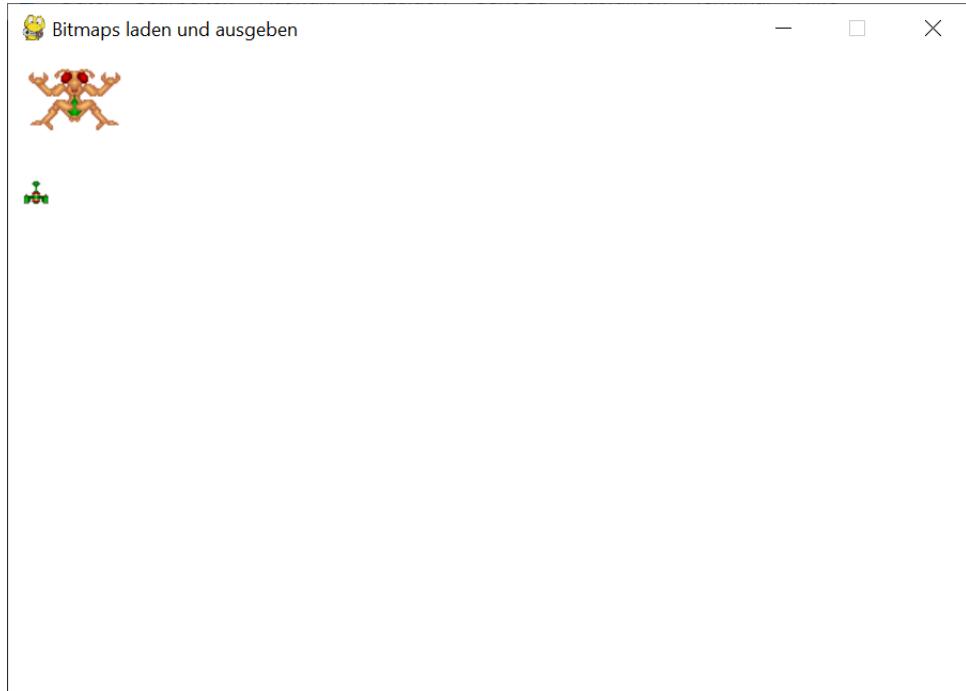


Abbildung 2.8: Bitmaps laden und ausgeben, Version 1.0

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

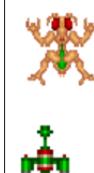


Abb. 2.10: Transparenz OK

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         clock.tick(Settings.FPS)
37         for event in pygame.event.get():
38             if event.type == pygame.QUIT:
39                 running = False
40
41         screen.fill("white")
42         alien_top = 10                                     # Abstand von oben
43         for i in range(Settings.ALIENS_NOF):             # Berechnung/Ausgabe
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)
47             screen.blit(defender_image, defender_pos)        # Benutze Position
48         pygame.display.flip()
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).

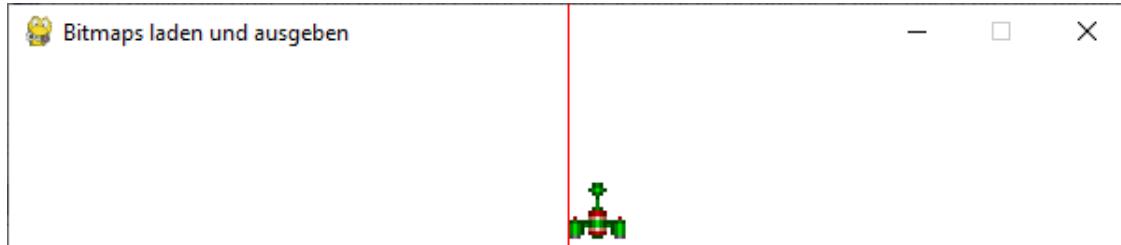


Abbildung 2.11: Bitmaps positionieren (Verteidiger)

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

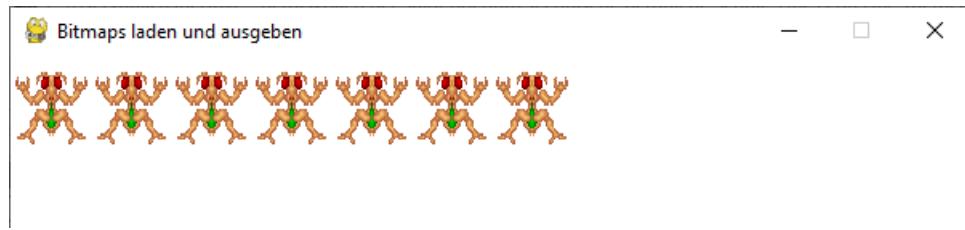


Abbildung 2.12: Bitmaps positionieren (Angreifer, Version 1)

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)`

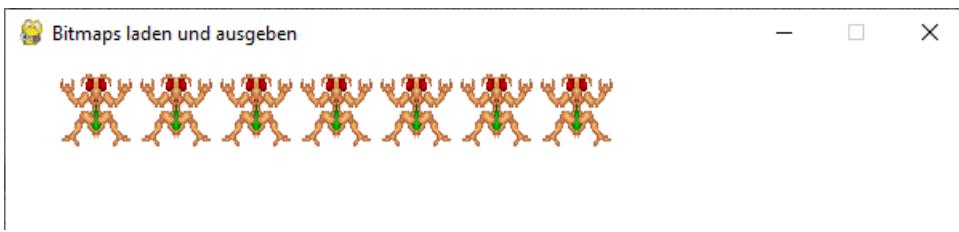


Abbildung 2.13: Bitmaps positionieren (Angreifer, Version 2)

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).

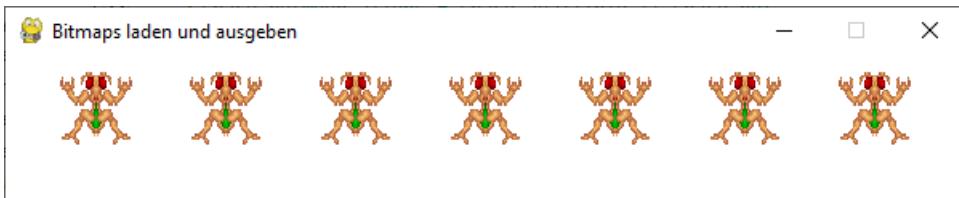


Abbildung 2.14: Bitmaps positionieren (Angreifer, Version 3)

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://www.pygame.org/docs/ref/image.html>

- `pygame.image.load()` :
<https://www.pygame.org/docs/ref/image.html#pygame.image.load>
- `pygame.Surface.blit()` :
<https://www.pygame.org/docs/ref/surface.html#pygame.Surface.blit>
- `pygame.Surface.convert()` :
<https://www.pygame.org/docs/ref/surface.html#pygame.Surface.convert>
- `pygame.Surface.convert_alpha()` :
https://www.pygame.org/docs/ref/surface.html#pygame.Surface.convert_alpha
- `pygame.Surface.set_colorkey()` :
https://www.pygame.org/docs/ref/surface.html#pygame.Surface.set_colorkey
- `pygame.transform.scale()` :
<https://www.pygame.org/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 Klasse `pygame.Rect` zur Verfügung. In Abbildung 2.15 finden Sie die meiner Ansicht nach wichtigsten Attribute der Klasse.

`Rect`

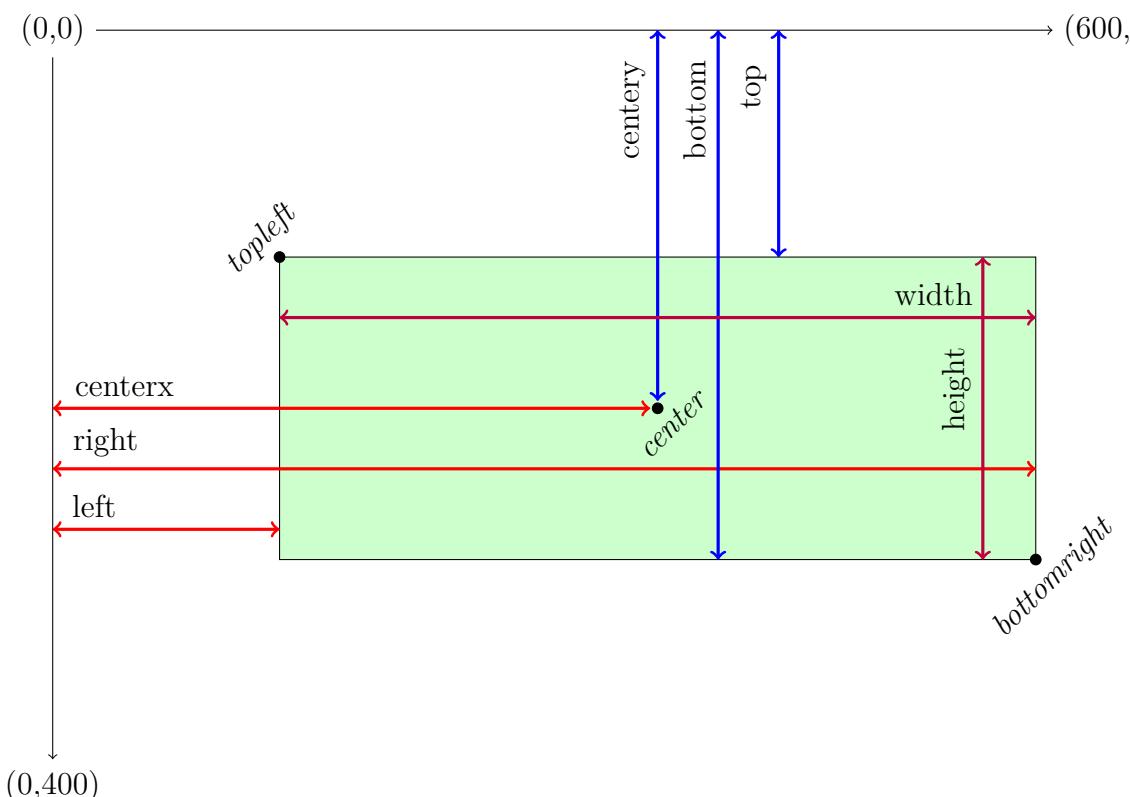


Abbildung 2.15: Elemente eines `Rect`-Objekts

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

`center`. 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         clock.tick(Settings.FPS)
28         # Events
29         for event in pygame.event.get():
30             if event.type == pygame.QUIT:
31                 running = False
32
33         # Update
34
35         # Draw
36         screen.fill("white")
37         screen.blit(defender_image, defender_rect)           # blit kann auch rect
38         pygame.display.flip()
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 ein-

fließen, 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 37).

blit()

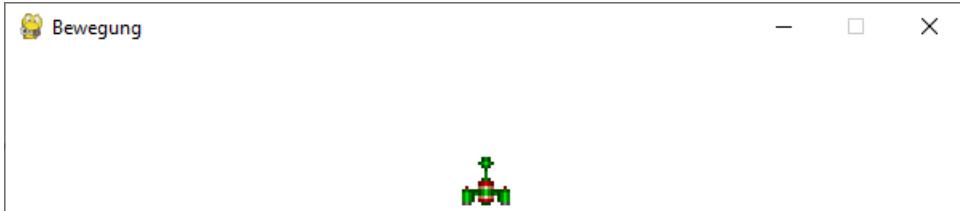


Abbildung 2.16: Bitmaps bewegen, Version 1.0

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! Richtung
- **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. 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      clock.tick(Settings.FPS)
30      # Events
31
32      # Geschwindigkeit
33      # Richtung
34
35      # Update
36
37      # Draw
38
39      # Update
40
41      # Draw
42
43      # Update
44
45      # Draw
46
47      # Update
48
49      # Draw
50
51      # Update
52
53      # Draw
54
55      # Update
56
57      # Draw
58
59      # Update
60
61      # Draw
62
63      # Update
64
65      # Draw
66
67      # Update
68
69      # Draw
70
71      # Update
72
73      # Draw
74
75      # Update
76
77      # Draw
78
79      # Update
80
81      # Draw
82
83      # Update
84
85      # Draw
86
87      # Update
88
89      # Draw
90
91      # Update
92
93      # Draw
94
95      # Update
96
97      # Draw
98
99      # Update
100
101     # Draw
102
103     # Update
104
105     # Draw
106
107     # Update
108
109     # Draw
110
111     # Update
112
113     # Draw
114
115     # Update
116
117     # Draw
118
119     # Update
120
121     # Draw
122
123     # Update
124
125     # Draw
126
127     # Update
128
129     # Draw
130
131     # Update
132
133     # Draw
134
135     # Update
136
137     # Draw
138
139     # Update
140
141     # Draw
142
143     # Update
144
145     # Draw
146
147     # Update
148
149     # Draw
150
151     # Update
152
153     # Draw
154
155     # Update
156
157     # Draw
158
159     # Update
160
161     # Draw
162
163     # Update
164
165     # Draw
166
167     # Update
168
169     # Draw
170
171     # Update
172
173     # Draw
174
175     # Update
176
177     # Draw
178
179     # Update
180
181     # Draw
182
183     # Update
184
185     # Draw
186
187     # Update
188
189     # Draw
190
191     # Update
192
193     # Draw
194
195     # Update
196
197     # Draw
198
199     # Update
200
201     # Draw
202
203     # Update
204
205     # Draw
206
207     # Update
208
209     # Draw
210
211     # Update
212
213     # Draw
214
215     # Update
216
217     # Draw
218
219     # Update
220
221     # Draw
222
223     # Update
224
225     # Draw
226
227     # Update
228
229     # Draw
230
231     # Update
232
233     # Draw
234
235     # Update
236
237     # Draw
238
239     # Update
240
241     # Draw
242
243     # Update
244
245     # Draw
246
247     # Update
248
249     # Draw
250
251     # Update
252
253     # Draw
254
255     # Update
256
257     # Draw
258
259     # Update
260
261     # Draw
262
263     # Update
264
265     # Draw
266
267     # Update
268
269     # Draw
270
271     # Update
272
273     # Draw
274
275     # Update
276
277     # Draw
278
279     # Update
280
281     # Draw
282
283     # Update
284
285     # Draw
286
287     # Update
288
289     # Draw
290
291     # Update
292
293     # Draw
294
295     # Update
296
297     # Draw
298
299     # Update
200
201     # Draw
202
203     # Update
204
205     # Draw
206
207     # Update
208
209     # Draw
2010
2011     # Update
2012
2013     # Draw
2014
2015     # Update
2016
2017     # Draw
2018
2019     # Update
2020
2021     # Draw
2022
2023     # Update
2024
2025     # Draw
2026
2027     # Update
2028
2029     # Draw
2030
2031     # Update
2032
2033     # Draw
2034
2035     # Update
2036
2037     # Draw
2038
2039     # Update
2040
2041     # Draw
2042
2043     # Update
2044
2045     # Draw
2046
2047     # Update
2048
2049     # Draw
2050
2051     # Update
2052
2053     # Draw
2054
2055     # Update
2056
2057     # Draw
2058
2059     # Update
2060
2061     # Draw
2062
2063     # Update
2064
2065     # Draw
2066
2067     # Update
2068
2069     # Draw
2070
2071     # Update
2072
2073     # Draw
2074
2075     # Update
2076
2077     # Draw
2078
2079     # Update
2080
2081     # Draw
2082
2083     # Update
2084
2085     # Draw
2086
2087     # Update
2088
2089     # Draw
2090
2091     # Update
2092
2093     # Draw
2094
2095     # Update
2096
2097     # Draw
2098
2099     # Update
2010
2011     # Draw
2012
2013     # Update
2014
2015     # Draw
2016
2017     # Update
2018
2019     # Draw
2020
2021     # Update
2022
2023     # Draw
2024
2025     # Update
2026
2027     # Draw
2028
2029     # Update
2030
2031     # Draw
2032
2033     # Update
2034
2035     # Draw
2036
2037     # Update
2038
2039     # Draw
2040
2041     # Update
2042
2043     # Draw
2044
2045     # Update
2046
2047     # Draw
2048
2049     # Update
2050
2051     # Draw
2052
2053     # Update
2054
2055     # Draw
2056
2057     # Update
2058
2059     # Draw
2060
2061     # Update
2062
2063     # Draw
2064
2065     # Update
2066
2067     # Draw
2068
2069     # Update
2070
2071     # Draw
2072
2073     # Update
2074
2075     # Draw
2076
2077     # Update
2078
2079     # Draw
2080
2081     # Update
2082
2083     # Draw
2084
2085     # Update
2086
2087     # Draw
2088
2089     # Update
2090
2091     # Draw
2092
2093     # Update
2094
2095     # Draw
2096
2097     # Update
2098
2099     # Draw
2010
2011     # Update
2012
2013     # Draw
2014
2015     # Update
2016
2017     # Draw
2018
2019     # Update
2020
2021     # Draw
2022
2023     # Update
2024
2025     # Draw
2026
2027     # Update
2028
2029     # Draw
2030
2031     # Update
2032
2033     # Draw
2034
2035     # Update
2036
2037     # Draw
2038
2039     # Update
2040
2041     # Draw
2042
2043     # Update
2044
2045     # Draw
2046
2047     # Update
2048
2049     # Draw
2050
2051     # Update
2052
2053     # Draw
2054
2055     # Update
2056
2057     # Draw
2058
2059     # Update
2060
2061     # Draw
2062
2063     # Update
2064
2065     # Draw
2066
2067     # Update
2068
2069     # Draw
2070
2071     # Update
2072
2073     # Draw
2074
2075     # Update
2076
2077     # Draw
2078
2079     # Update
2080
2081     # Draw
2082
2083     # Update
2084
2085     # Draw
2086
2087     # Update
2088
2089     # Draw
2090
2091     # Update
2092
2093     # Draw
2094
2095     # Update
2096
2097     # Draw
2098
2099     # Update
2010
2011     # Draw
2012
2013     # Update
2014
2015     # Draw
2016
2017     # Update
2018
2019     # Draw
2020
2021     # Update
2022
2023     # Draw
2024
2025     # Update
2026
2027     # Draw
2028
2029     # Update
2030
2031     # Draw
2032
2033     # Update
2034
2035     # Draw
2036
2037     # Update
2038
2039     # Draw
2040
2041     # Update
2042
2043     # Draw
2044
2045     # Update
2046
2047     # Draw
2048
2049     # Update
2050
2051     # Draw
2052
2053     # Update
2054
2055     # Draw
2056
2057     # Update
2058
2059     # Draw
2060
2061     # Update
2062
2063     # Draw
2064
2065     # Update
2066
2067     # Draw
2068
2069     # Update
2070
2071     # Draw
2072
2073     # Update
2074
2075     # Draw
2076
2077     # Update
2078
2079     # Draw
2080
2081     # Update
2082
2083     # Draw
2084
2085     # Update
2086
2087     # Draw
2088
2089     # Update
2090
2091     # Draw
2092
2093     # Update
2094
2095     # Draw
2096
2097     # Update
2098
2099     # Draw
2010
2011     # Update
2012
2013     # Draw
2014
2015     # Update
2016
2017     # Draw
2018
2019     # Update
2020
2021     # Draw
2022
2023     # Update
2024
2025     # Draw
2026
2027     # Update
2028
2029     # Draw
2030
2031     # Update
2032
2033     # Draw
2034
2035     # Update
2036
2037     # Draw
2038
2039     # Update
2040
2041     # Draw
2042
2043     # Update
2044
2045     # Draw
2046
2047     # Update
2048
2049     # Draw
2050
2051     # Update
2052
2053     # Draw
2054
2055     # Update
2056
2057     # Draw
2058
2059     # Update
2060
2061     # Draw
2062
2063     # Update
2064
2065     # Draw
2066
2067     # Update
2068
2069     # Draw
2070
2071     # Update
2072
2073     # Draw
2074
2075     # Update
2076
2077     # Draw
2078
2079     # Update
2080
2081     # Draw
2082
2083     # Update
2084
2085     # Draw
2086
2087     # Update
2088
2089     # Draw
2090
2091     # Update
2092
2093     # Draw
2094
2095     # Update
2096
2097     # Draw
2098
2099     # Update
2010
2011     # Draw
2012
2013     # Update
2014
2015     # Draw
2016
2017     # Update
2018
2019     # Draw
2020
2021     # Update
2022
2023     # Draw
2024
2025     # Update
2026
2027     # Draw
2028
2029     # Update
2030
2031     # Draw
2032
2033     # Update
2034
2035     # Draw
2036
2037     # Update
2038
2039     # Draw
2040
2041     # Update
2042
2043     # Draw
2044
2045     # Update
2046
2047     # Draw
2048
2049     # Update
2050
2051     # Draw
2052
2053     # Update
2054
2055     # Draw
2056
2057     # Update
2058
2059     # Draw
2060
2061     # Update
2062
2063     # Draw
2064
2065     # Update
2066
2067     # Draw
2068
2069     # Update
2070
2071     # Draw
2072
2073     # Update
2074
2075     # Draw
2076
2077     # Update
2078
2079     # Draw
2080
2081     # Update
2082
2083     # Draw
2084
2085     # Update
2086
2087     # Draw
2088
2089     # Update
2090
2091     # Draw
2092
2093     # Update
2094
2095     # Draw
2096
2097     # Update
2098
2099     # Draw
2010
2011     # Update
2012
2013     # Draw
2014
2015     # Update
2016
2017     # Draw
2018
2019     # Update
2020
2021     # Draw
2022
2023     # Update
2024
2025     # Draw
2026
2027     # Update
2028
2029     # Draw
2030
2031     # Update
2032
2033     # Draw
2034
2035     # Update
2036
2037     # Draw
2038
2039     # Update
2040
2041     # Draw
2042
2043     # Update
2044
2045     # Draw
2046
2047     # Update
2048
2049     # Draw
2050
2051     # Update
2052
2053     # Draw
2054
2055     # Update
2056
2057     # Draw
2058
2059     # Update
2060
2061     # Draw
2062
2063     # Update
2064
2065     # Draw
2066
2067     # Update
2068
2069     # Draw
2070
2071     # Update
2072
2073     # Draw
2074
2075     # Update
2076
2077     # Draw
2078
2079     # Update
2080
2081     # Draw
2082
2083     # Update
2084
2085     # Draw
2086
2087     # Update
2088
2089     # Draw
2090
2091     # Update
2092
2093     # Draw
2094
2095     # Update
2096
2097     # Draw
2098
2099     # Update
2010
2011     # Draw
2012
2013     # Update
2014
2015     # Draw
2016
2017     # Update
2018
2019     # Draw
2020
2021     # Update
2022
2023     # Draw
2024
2025     # Update
2026
2027     # Draw
2028
2029     # Update
2030
2031     # Draw
2032
2033     # Update
2034
2035     # Draw
2036
2037     # Update
2038
2039     # Draw
2040
2041     # Update
2042
2043     # Draw
2044
2045     # Update
2046
2047     # Draw
2048
2049     # Update
2050
2051     # Draw
2052
2053     # Update
2054
2055     # Draw
2056
2057     # Update
2058
2059     # Draw
2060
2061     # Update
2062
2063     # Draw
2064
2065     # Update
2066
2067     # Draw
2068
2069     # Update
2070
2071     # Draw
2072
2073     # Update
2074
2075     # Draw
2076
2077     # Update
2078
2079     # Draw
2080
2081     # Update
2082
2083     # Draw
2084
2085     # Update
2086
2087     # Draw
2088
2089     # Update
2090
2091     # Draw
2092
2093     # Update
2094
2095     # Draw
2096
2097     # Update
2098
2099     # Draw
2010
2011     # Update
2012
2013     # Draw
2014
2015     # Update
2016
2017     # Draw
2018
2019     # Update
2020
2021     # Draw
2022
2023     # Update
2024
2025     # Draw
2026
2027     # Update
2028
2029     # Draw
2030
2031     # Update
2032
2033     # Draw
2034
2035     # Update
2036
2037     # Draw
2038
2039     # Update
2040
2041     # Draw
2042
2043     # Update
2044
2045     # Draw
2046
2047     # Update
2048
2049     # Draw
2050
2051     # Update
2052
2053     # Draw
2054
2055     # Update
2056
2057     # Draw
2058
2059     # Update
2060
2061     # Draw
2062
2063     # Update
2064
2065     # Draw
2066
2067     # Update
2068
2069     # Draw
2070
2071     # Update
2072
2073     # Draw
2074
2075     # Update
2076
2077     # Draw
2078
2079     # Update
2080
2081     # Draw
2082
2083     # Update
2084
2085     # Draw
2086
2087     # Update
2088
2089     # Draw
2090
2091     # Update
2092
2093     # Draw
2094
2095     # Update
2096
2097     # Draw
2098
2099     # Update
2010
2011     # Draw
2012
2013     # Update
2014
2015     # Draw
2016
2017     # Update
2018
2019     # Draw
2020
2021     # Update
2022
2023     # Draw
2024
2025     # Update
2026
2027     # Draw
2028
2029     # Update
2030
2031     # Draw
2032
2033     # Update
2034
2035     # Draw
2036
2037     # Update
2038
2039     # Draw
2040
2041     # Update
2042
2043     # Draw
2044
2045     # Update
2046
2047     # Draw
2048
2049     # Update
2050
2051     # Draw
2052
2053     # Update
2054
2055     # Draw
2056
2057     # Update
2058
2059     # Draw
2060
2061     # Update
2062
2063     # Draw
2064
2065     # Update
2066
2067     # Draw
2068
2069     # Update
2070
2071     # Draw
2072
2073     # Update
2074
2075     # Draw
2076
2077     # Update
2078
2079     # Draw
2080
2081     # Update
2082
2083     # Draw
2084
2085     # Update
2086
2087     # Draw
2088
2089     # Update
2090
2091     # Draw
2092
2093     # Update
2094
2095     # Draw
2096
2097     # Update
2098
2099     # Draw
2010
2011     # Update
2012
2013     # Draw
2014
2015     # Update
2016
2017     # Draw
2018
2019     # Update
2020
2021     # Draw
2022
2023     # Update
2024
2025     # Draw
2026
2027     # Update
2028
2029     # Draw
2030
2031     # Update
2032
2033     # Draw
2034
2035     # Update
2036
2037     # Draw
2038
2039     # Update
2040
2041     # Draw
2042
2043     # Update
2044
2045     # Draw
2046
2047     # Update
2048
2049     # Draw
2050
2051     # Update
2052
2053     # Draw
2054
2055     # Update
2056
2057     # Draw
2058
2059     # Update
2060
2061     # Draw
2062
2063     # Update
2064
2065     # Draw
2066
2067     # Update
2068
2069     # Draw
2070
2071     # Update
2072
2073     # Draw
2074
2075     # Update
2076
2077     # Draw
2078
2079     # Update
2080
2081     # Draw
2082
2083     # Update
2084
2085     # Draw
2086
2087     # Update
2088
2089     # Draw
2090
2091     # Update
2092
2093     # Draw
2094
2095     # Update
2096
2097     # Draw
2098
2099     # Update
2010
2011     # Draw
2012
2013     # Update
2014
2015     # Draw
2016
2017     # Update
2018
2019     # Draw
2020
2021     # Update
2022
2023     # Draw
2024
2025     # Update
2026
2027     # Draw
2028
2029     # Update
2030
2031     # Draw
2032
2033     # Update
2034
2035     # Draw
2036
2037     # Update
2038
2039     # Draw
2040
2041     # Update
2042
2043     # Draw
2044
2045     # Update
2046
2047     # Draw
2048
2049     # Update
2050
2051     # Draw
2052
2053     # Update
2054
2055     # Draw
2056
2057     # Update
2058
2059     # Draw
2060
2061     # Update
2062
2063     # Draw
2064
2065     # Update
2066
2067     # Draw
2068
2069     # Update
2070
2071     # Draw
2072
2073     # Update
2074
2075     # Draw
2076
2077     # Update
2078
2079     # Draw
2080
2081     # Update
2082
2083     # Draw
2084
2085     # Update
2086
2087     # Draw
2088
2089     # Update
2090
2091     # Draw
2092
2093     # Update
2094
2095     # Draw
2096
2097     # Update
2098
2099     # Draw
2010
2011     # Update
2012
2013     # Draw
2014
2015     # Update
2016
2017     # Draw
2018
2019     # Update
2020
2021     # Draw
2022
2023     # Update
2024
2025     # Draw
2026
2027     # Update
2028
2029     # Draw
2030
2031     # Update
2032
2033     # Draw
2034
2035     # Update
2036
2037     # Draw
2038
2039     # Update
2040
2041     # Draw
2042
2043     # Update
2044
2045     # Draw
2046
2047     # Update
2048
2049     # Draw
2050
2051     # Update
2052
2053     # Draw
2054
2055     # Update
2056
2057     # Draw
2058
2059     # Update
2060
2061     # Draw
2062
2063     # Update
2064
2065     # Draw
2066
2067     # Update
2068
2069     # Draw
2070
2071     # Update
2072
2073     # Draw
2074
2075     # Update
2076
2077     # Draw
2078
2079     # Update
2080
2081     # Draw
2082
2083     # Update
2084
2085     # Draw
2086
2087     # Update
2088
2089     # Draw
2090
2091     # Update
2092
2093     # Draw
2094
2095     # Update
2096
2097     # Draw
2098
2099     # Update
2010
2011     # Draw
2012
2013     # Update
2014
2015     # Draw
2016
2017     # Update
2018
2019     # Draw
2020
2021     # Update
2022
2023     # Draw
2024
2025     # Update
2026
2027     # Draw
2028
2029     # Update
2030
2031     # Draw
2032
2033     # Update
2034
2035     # Draw
2036
2037     # Update
2038
2039     # Draw
2040
2041     # Update
2042
2043     # Draw
2044
2045     # Update
2046
2047     # Draw
2048
2049     # Update
2050
2051     # Draw
2052
2053     # Update
2054
2055     # Draw
2056
2057     # Update
2058
2059     # Draw
2060
2061     # Update
2062
2063     # Draw
2064
2065     # Update
2066
2067     # Draw
2068
2069     # Update
2070
2071     # Draw
2072
2073     # Update
2074
2075     # Draw
2076
2077     # Update
2078
2079     # Draw
2080
2081     # Update
2082
2083     # Draw
2084
2085     # Update
2086
2087     # Draw
2088
2089     # Update
2090
2091     # Draw
2092
2093     # Update
2094
2095     # Draw
2096
2097     # Update
2098
2099     # Draw
2010
2011     # Update
2012
2013     # Draw
2014
2015     # Update
2016
2017     # Draw
2018
2019     # Update
2020
2021     # Draw
2022
2023     # Update
2024
2025     # Draw
2026
2027     # Update
2028
2029     # Draw
2030
2031     # Update
2032
2033     # Draw
2034
2035     # Update
2036
2037     # Draw
2038
2039     # Update
2040
2041     # Draw
2042
2043     # Update
2044
2045     # Draw
2046
2047     # Update
2048
2049     # Draw
2050
2051     # Update
2052
2053     # Draw
2054
2055     # Update
2056
2057     # Draw
20
```

```

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

```

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 Rakettentreibstoffausstoß.

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

```

Ich hoffe, dass Sie die Idee hinter dem Code erkennen. Nach Berechnung der neuen horizontalen Position, wird in Zeile 37 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 36 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.move()`.

`move()`

Quelltext 2.19: Bitmaps bewegen, Version 1.4

```

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

```

Die neue Funktion taucht in Zeile 36 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 42).

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 = 500
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*)

delatime

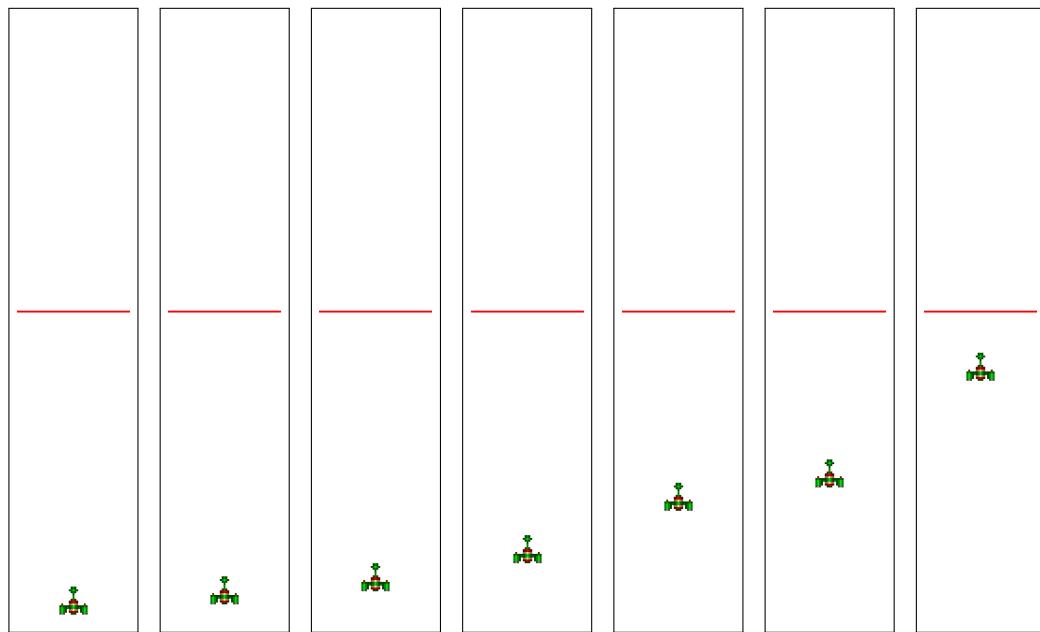


Abbildung 2.18: Nicht normalisierte Bewegung bei gleicher Geschwindigkeit aber unterschiedlichen Frameraten (von links nach rechts: 10, 30, 60, 120, 240, 300, 600)

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 = 600
10     LIMIT = 500
11     DELTATIME = 1.0/FPS
12
13
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 = 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.

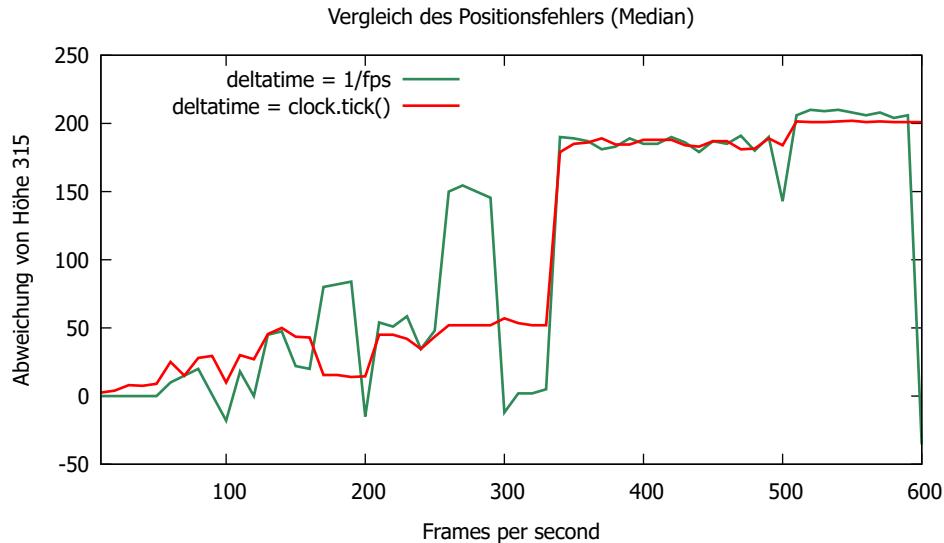


Abbildung 2.19: Vergleich des Positionsfehlers von $1/fps$ und `pygame.clock.tick()`

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

Wir müssen die Positionswerte also unabhängig vom `Rect`-Objekt in einer zusätzlichen `float`-Variablen abspeichern, damit die Nachkommastellen erhalten bleiben. In Zeile 27 wird die aktuelle Position in einem `pygame.math.Vector2`-Objekt abgespeichert. Diese Klasse ist sehr gut dafür geeignet, Positionsangaben zu speichern. Mit Hilfe von `Vector2` und seinem Verwandten `pygame.math.Vector3` können Operationen wie Addition und Multiplikation einfacher implementiert werden. Mehr dazu später.

`Vector2`

`Vector3`

Auch werden die errechneten Positionswerte nicht einfach den Integer-Attributen des `Rect`-Objektes zugewiesen, sondern mit Hilfe von `round` wird die Ganzzahl passend zu den Nachkommawerten nach den üblichen Rundungsregeln ermittelt (siehe Zeile 43).

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 = defender_image.get_rect()
25     defender_rect.centerx = Settings.WINDOW.centerx
26     defender_rect.bottom = Settings.WINDOW.bottom - 5
27     defender_pos = pygame.math.Vector2(defender_rect.left, defender_rect.top)      # float
28     defender_speed = 600
29     defender_direction_v = -1
30
31     start_time = pygame.time.get_ticks()
32     running = True
33     while running:
34         if pygame.time.get_ticks() > start_time + Settings.LIMIT:
35             defender_speed = 0
36         # Events
37         for event in pygame.event.get():
38             if event.type == pygame.QUIT:
39                 running = False
40
41         # Update
42         defender_pos[1] += defender_direction_v * defender_speed * Settings.DELTATIME
43         defender_rect.top = round(defender_pos[1])                                # Runden
44         if defender_rect.bottom >= Settings.WINDOW.bottom:
45             defender_direction_v *= -1
46         elif defender_rect.top <= 0:
47             defender_direction_v *= -1
48
49         # Draw
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         Settings.DELTATIME = clock.tick(Settings.FPS) / 1000.0
55         print(f"top={defender_rect.top}")
56
57     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 multipilizieren sich diese fehlenden Angaben nach vorne und führen wiederum zu Fehlern. Es gibt bessere Python-Funktionen zur Ermittlung von Laufzeiten.

In Zeile 33 von Quelltext 2.24 auf Seite 42 wird mit Hilfe von `time.time()` die Anzahl

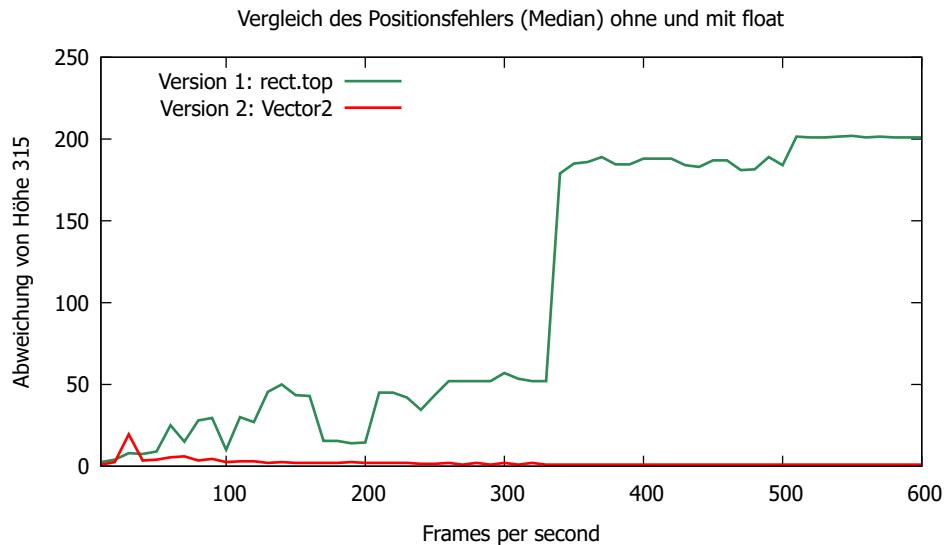


Abbildung 2.20: Vergleich des Positionsfehlers ohne und mit Vector2

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.

In Zeile 57 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 59 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.

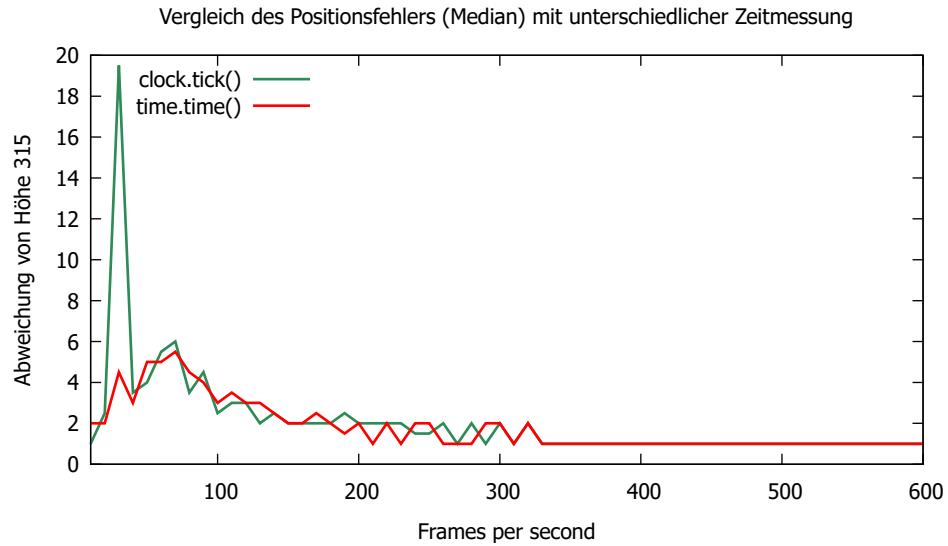


Abbildung 2.21: Vergleich der Positionsfehler mit unterschiedlichen Genauigkeiten

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 = defender_image.get_rect()
26     defender_rect.centerx = Settings.WINDOW.centerx
27     defender_rect.bottom = Settings.WINDOW.height - 5
28     defender_pos = pygame.Vector2(defender_rect.left, defender_rect.top)
29     defender_speed = 600
30     defender_direction_v = -1
31
32     start_time = pygame.time.get_ticks()
33     time_previous = time()                                     # Startzeit festhalten
34     running = True
35     while running:
36         if pygame.time.get_ticks() > start_time + Settings.LIMIT:
37             defender_speed = 0
38
39         # Events
40         for event in pygame.event.get():
41             if event.type == pygame.QUIT:
42                 running = False
43
44         # Update
45         defender_pos[1] += defender_direction_v * defender_speed * Settings.DELTATIME
46         defender_rect.top = round(defender_pos[1])
47         if defender_rect.bottom >= Settings.WINDOW.height:
48             defender_direction_v *= -1
49         elif defender_rect.top <= 0:
50             defender_direction_v *= -1

```

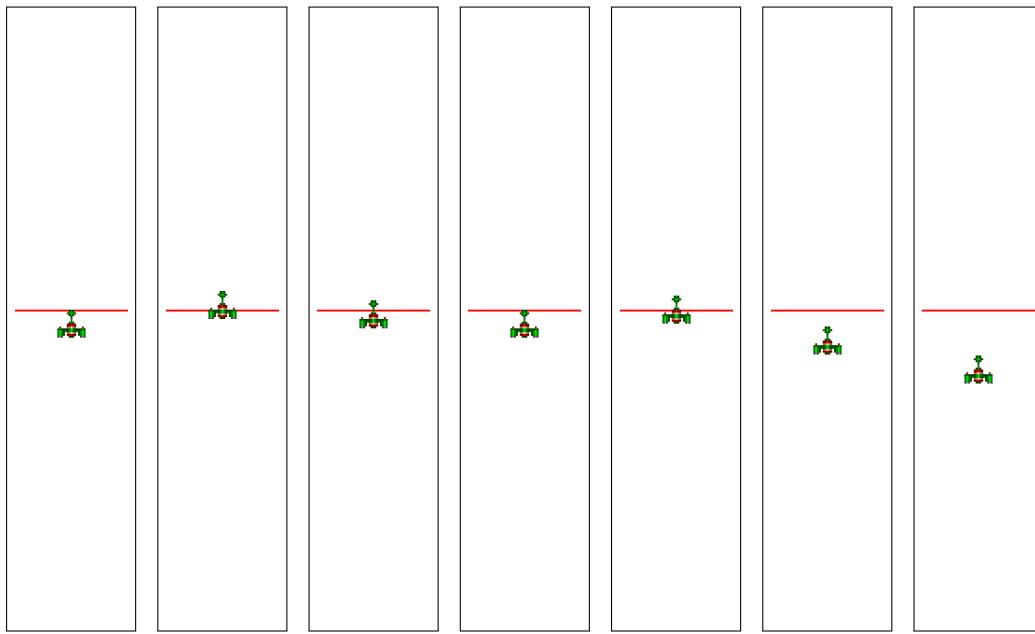


Abbildung 2.22: Normalisierte Bewegung mit $1/fps$ bei gleicher Geschwindigkeit aber unterschiedlichen Frameraten (von links nach rechts: 10, 30, 60, 120, 240, 300, 600)

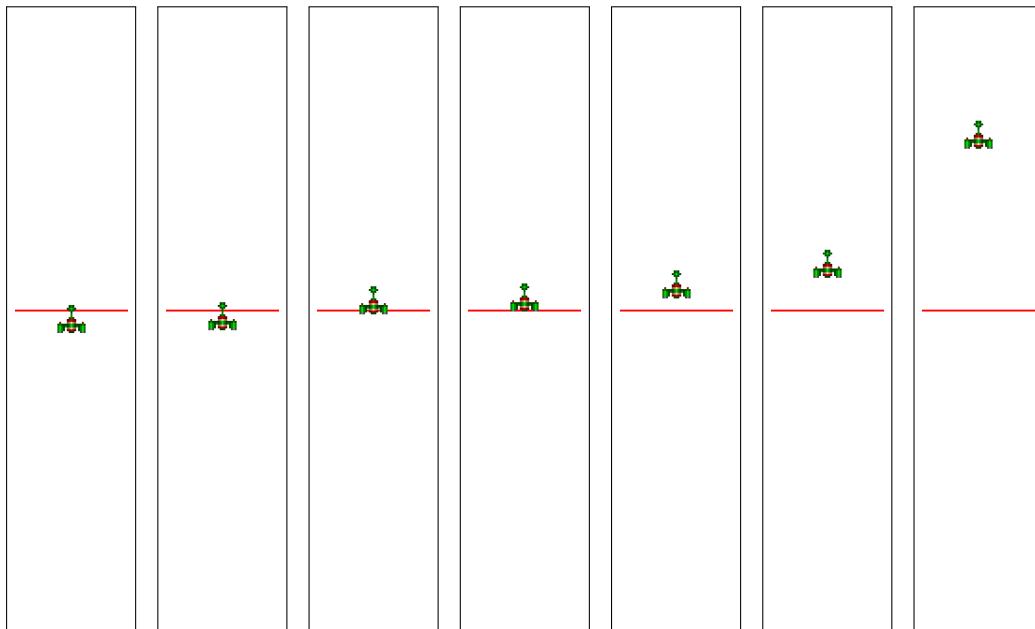


Abbildung 2.23: Normalisierte Bewegung mit `pygame.clock.tick()` bei gleicher Geschwindigkeit aber unterschiedlichen Frameraten (von links nach rechts: 10, 30, 60, 120, 240, 300, 600)

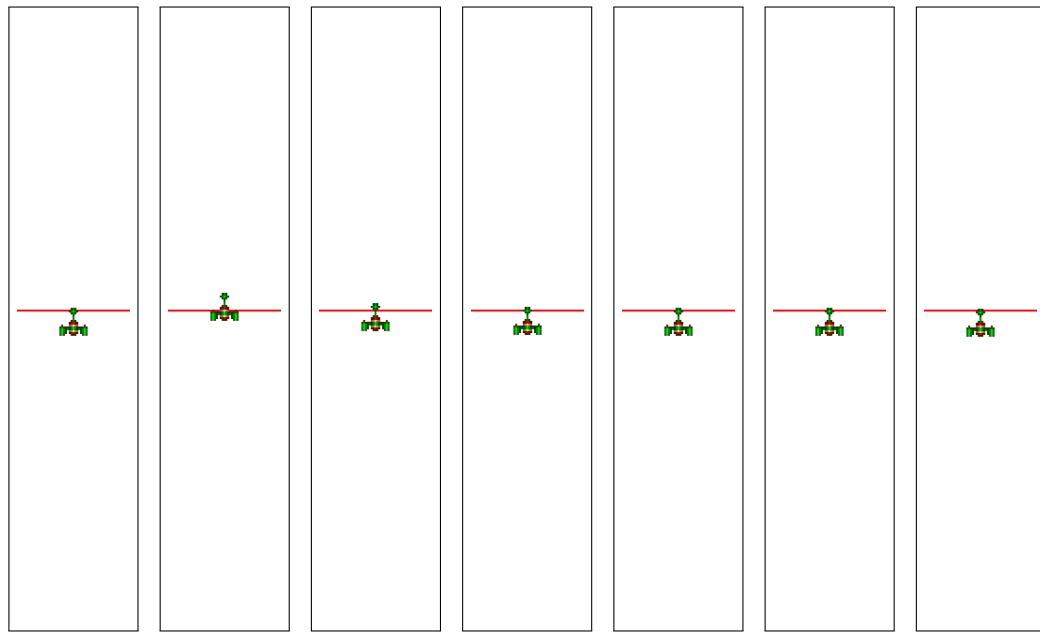


Abbildung 2.24: Normalisierte Bewegung mit `pygame.clock.tick()` (float) bei gleicher Geschwindigkeit aber unterschiedlichen Frameraten (von links nach rechts: 10, 30, 60, 120, 240, 300, 600)

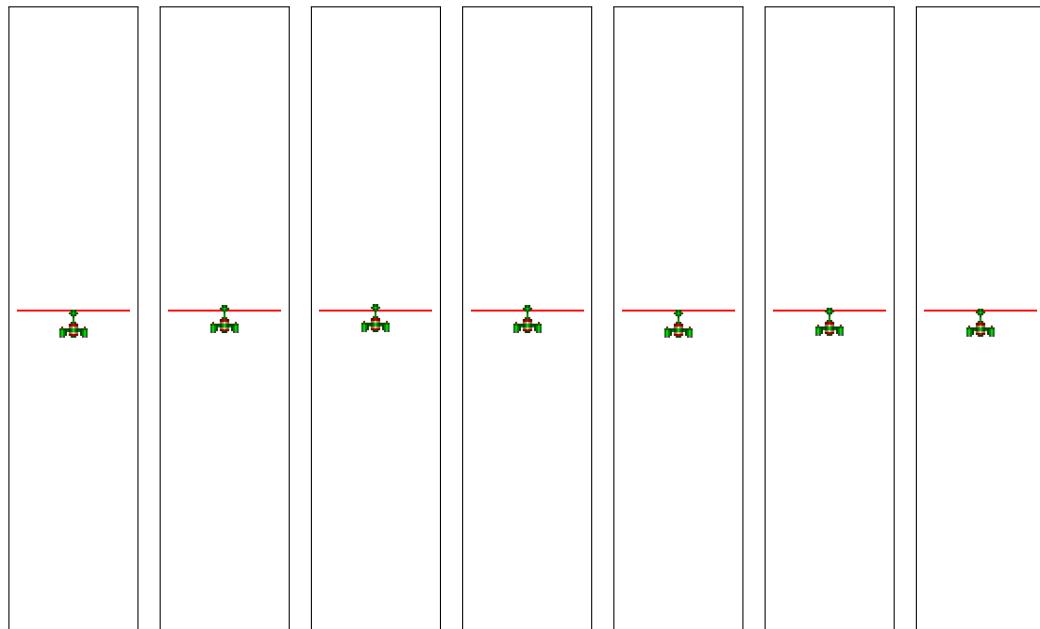


Abbildung 2.25: Normalisierte Bewegung mit `time.time()` bei gleicher Geschwindigkeit aber unterschiedlichen Frameraten (von links nach rechts: 10, 30, 60, 120, 240, 300, 600), Version 3

```
50
51     # Draw
52     screen.fill("white")
53     pygame.draw.line(screen, "red", (0, 315), (Settings.WINDOW.width, 315), 2)
54     screen.blit(defender_image, defender_rect)
55     pygame.display.flip()
56     clock.tick(Settings.FPS)
57     time_current = time()                      # Aktuelle Zeit festhalten
58     Settings.DELTATIME = time_current - time_previous  # Zeitverbrauch
59     time_previous = time_current                # Neue Startzeit
60
61     pygame.quit()
```

Was war neu?

Die Position eines Objektes wird in einem `Rect`-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.

Das Rechteck des Bitmaps speichert seine Werte als Ganzzahlen ab. Dadurch gehen die Nachkommastellen verloren und führen zu Positionsfehlern. Parallel zum Rechteck sollte daher ein `Vector2`-Objekt zur Speicherung der Positionskoordinaten verwendet werden. Dieses speichert die Werte als Fließkommazahlen ab.

Es wurden folgende Pygame-Elemente eingeführt:

- `pygame.Rect`:
<https://www.pygame.org/docs/ref/rect.html>
- `pygame.Rect.move()`:
<https://www.pygame.org/docs/ref/rect.html#pygame.Rect.move>
- `pygame.Surface.get_rect()`:
https://www.pygame.org/docs/ref/surface.html#pygame.Surface.get_rect
- `pygame.time.get_ticks()`:
https://www.pygame.org/docs/ref/surface.html#pygame.time.get_ticks
- `pygame.math.Vector2`:
<https://www.pygame.org/docs/ref/math.html#pygame.math.Vector2>
- `pygame.math.Vector3`:
<https://www.pygame.org/docs/ref/math.html#pygame.math.Vector3>

2.5 Sprite-Klasse

Im letzten Beispiel viel 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`.

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 = self.image.get_rect()
20          self.rect.centerx = Settings.WINDOW.centerx
21          self.rect.bottom = Settings.WINDOW.bottom - 5
22          self.position = pygame.math.Vector2(self.rect.left, self.rect.top)
23          self.speed = pygame.math.Vector2(300, 0)
24
25      def update(self) -> None:                   # Zustandsberechnung
26          newpos = self.position + (self.speed * Settings.DELTATIME)
27          if newpos.x + self.rect.width >= Settings.WINDOW.right:
28              self.change_direction()
29          elif newpos.x <= Settings.WINDOW.left:
30              self.change_direction()
31          else:
32              self.position = newpos
33              self.rect.left = round(newpos.x)
34
35      def draw(self, screen: pygame.surface.Surface) -> None: # Malen
36          screen.blit(self.image, self.rect)
37
38      def change_direction(self) -> None:           # OO style
39          self.speed.x *= -1

```

Die Zeilen des Konstruktors (Zeile 15ff.) entsprechen denen der vorherigen Version, sind aber um die `Vector2`-Objekte zur Vermeidung von Rundungsfehlern ergänzt. Lediglich der Präfix `defender_` wird durch `self.` ersetzt, wodurch die Variablen zu Attributen der Klasse werden. Sie sollten keine Schwierigkeiten haben, diese zu verstehen.

Hier sehen Sie erstmalig die Verwendung von `Vector`-Objekte ausformuliert. Die Position und die beiden Richtungen (horizontal und vertikal) werden in solchen Vektoren abgespeichert.

Jede Kindklasse von `Sprite` muss zwei Attribute haben: `rect` und `image`. Auf diese

`Sprite`

`Vector2`

`self.rect`
`self.image`

beiden Attribute greifen nämlich die schon vorformulierten Lösungen zur Kollisionserkennung, Bildschirmausgabe etc. zu. Wir werden später noch den Nutzen sehen.

In Zeile 25ff. werden die Kollisionserkennungen und die Zustandsänderungen formuliert. Hier fällt besonders die Berechnung der neuen Position auf. Da wir hier erstmalig mit `Vector2`-Objekten arbeiten, schauen wir uns das mal genauer an:

```
(self.speed * Settings.DELTATIME)
```

`Settings.DELTATIME` ist ein skalarer Fließkommawert, beispielsweise 0.01, hingegen ist `self.speed` ein `Vector2`-Objekt, beispielsweise (10, 20). Was macht also das Sternchen? Es multipliziert beide Werte von `speed` mit `DELTATIME`. Unser Beispiel sähe dann so:

$$\begin{pmatrix} 10 \\ 20 \end{pmatrix} * 0.01 = \begin{pmatrix} 0.1 \\ 0.2 \end{pmatrix}$$

Ich habe also als Ergebnis der Klammer wieder ein `Vector2`-Objekt. Nun möchte ich dieses Objekt mit der Position addieren:

$$\begin{pmatrix} 0.1 \\ 0.2 \end{pmatrix} + \begin{pmatrix} 100 \\ 200 \end{pmatrix} = \begin{pmatrix} 100.1 \\ 200.2 \end{pmatrix}$$

Genau das passiert mit

```
newpos = self.position + (self.speed * Settings.DELTATIME)
```

Neu ist der Aufruf der Methode `change_direction()`. Diese Methode (Zeile 38) 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()` (Zeile 35) wird die Bildschirmausgabe gekapselt.

Quelltext 2.26: Sprites (2), Version 1.0

```

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 = Defender()                                # Objekt anlegen
50
51     time_previous = time()
52     running = True
53     while running:
54         # Events
55         for event in pygame.event.get():
56             if event.type == pygame.QUIT:
57                 running = False

```

```

58
59     # Update
60     defender.update()                      # Aufruf
61
62     # Draw
63     screen.fill("white")
64     defender.draw(screen)                  # Aufruf
65     pygame.display.flip()
66
67     clock.tick(Settings.FPS)
68     time_current = time()
69     Settings.DELTATIME = time_current - time_previous
70     time_previous = time_current
71     pygame.quit()

```

Die Verwendung der Klasse `Defender` ist nun denkbar einfach geworden. In der Zeile 49 wird ein Objekt der Klasse erzeugt. In Zeile 60 wird `update()` aufgerufen und in Zeile 64 `draw()`.

Ein Vorteil der neuen Architektur ist die besser 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, wird erstmal das `update()` wieder einfach. Wir berechnen lediglich die neue Position.

Quelltext 2.27: Sprites (1), Version 1.1

```

25 def update(self) -> None:
26     self.position = self.position + (self.speed * Settings.DELTATIME)
27     self.rect.left = round(self.position.x)

```

Damit die Ränder mal sichtbar werden und ich die Kollision besser erkennbar mache, 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

```

36 class Border(pygame.sprite.Sprite):
37
38     def __init__(self, leftright: str) -> None:
39         super().__init__()
40         self.image: pygame.surface.Surface =
41             pygame.image.load("images/brick01.png").convert_alpha()
42         self.image = pygame.transform.scale(self.image, (35, Settings.WINDOW.width))
43         self.rect: pygame.rect.Rect = self.image.get_rect()
44         if leftright == 'right':
45             self.rect.left = Settings.WINDOW.width - self.rect.width
46
47     def draw(self, screen: pygame.surface.Surface) -> None:
48         screen.blit(self.image, self.rect)

```

Das `update()` ist bei einer starren Wand funktionslos und bleibt daher leer. Nun erzeuge ich die beiden Ränder:

Quelltext 2.29: Sprites (3), Version 1.1

```
58 border_left = Border('left')
59 border_right = Border('right')
```

Bisher war alles easy.

Quelltext 2.30: Sprites (4), Version 1.1

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

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.

`collide_rect()`

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

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

```

71     defender.draw(screen)
72     all_border.draw(screen)           # Mit einem Rutsch
73     pygame.display.flip()
74
75     clock.tick(Settings.FPS)
76     time_current = time()
77     Settings.DELTATIME = time_current - time_previous
78     time_previous = time_current
79     pygame.quit()

```

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

Group

spritecollide()

Die erste Stelle ist Zeile 65 und dort wird eine andere Version der Kollisionsprüfung verwendet: `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 Raumschiffe, die von einem Felsen getroffen wurden, löschen.

Die zweite Stelle ist Zeile 72. Hier wird nicht mehr für jedes Objekt einzeln `draw()` aufgerufen, 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 66f.).

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

```

45  class Game(object):
46
47      def __init__(self) -> None:
48          os.environ['SDL_VIDEO_WINDOW_POS'] = "10, 50"
49          pygame.init()
50          self.screen = pygame.display.set_mode(Settings.WINDOW.size)
51          pygame.display.set_caption("Sprite")
52          self.clock = pygame.time.Clock()
53          self.defender = pygame.sprite.GroupSingle(Defender())
54          self.all_border = pygame.sprite.Group()
55          self.all_border.add(Border('left'))
56          self.all_border.add(Border('right'))
57          self.running = False
58
59      def run(self) -> None:
60          time_previous = time()
61          self.running = True
62          while self.running:
63              self.watch_for_events()
64              self.update()
65              self.draw()
66              self.clock.tick(Settings.FPS)
67              time_current = time()
68              Settings.DELTATIME = time_current - time_previous
69              time_previous = time_current
70          pygame.quit()
71
72      def watch_for_events(self) -> None:
73          for event in pygame.event.get():
74              if event.type == pygame.QUIT:
75                  self.running = False
76
77      def update(self) -> None:
78          if pygame.sprite.spritecollide(self.defender.sprite, self.all_border, False):
79              self.defender.sprite.change_direction() # Gefällt mir nicht!
80          self.defender.update()
81
82      def draw(self) -> None:
83          self.screen.fill((255, 255, 255))
84          self.defender.draw(self.screen)
85          self.all_border.draw(self.screen)
86          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 79 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 31 durch `update()` aufgerufen und nicht mehr von außen.

Quelltext 2.33: `Defender.update()`

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

Der Aufruf erfolgt dann in Quelltext 2.34 in Zeile 83 indirekt durch die Verwendung des Übergabeparameters.

Quelltext 2.34: `Game.update()`

```
81     def update(self) -> None:
82         if pygame.sprite.spritecollide(self.defender.sprite, self.all_border, False):
83             self.defender.update(action="direction")  # Besser
84             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://www.pygame.org/docs/ref/rect.html#pygame.Rect.move>
- `pygame.Rect.move_ip()`:
https://www.pygame.org/docs/ref/rect.html#pygame.Rect.move_ip
- `pygame.sprite.Group`:
<https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.Group>
- `pygame.sprite.GroupSingle`:
<https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.GroupSingle>
- `pygame.sprite.GroupSingle.sprite`:
<https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.GroupSingle>

- `pygame.sprite.Sprite`:
<https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.Sprite>
- `pygame.sprite.collide_rect()`:
https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.collide_rect
- `pygame.sprite.spritecollide()`:
<https://www.pygame.org/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 Raumschiff stehen bleiben oder sich wieder in Bewegung setzen können. Auch kann das Spiel jetzt durch die Escape-Taste verlassen werden ([Boss-Taste](#)).

Zunächst bereiten wir die Verteidiger-Klasse vor bzw. wandeln sie ein wenig ab (Quelltext [2.35](#)). Das Sprite wird nun nicht mehr unten sondern mittig platziert (Zeile [21](#)). Das Raumschiff soll sich nun auch vertikal bewegen können. Dazu braucht es entweder zwei entsprechende Variablen oder aber ein `Vector2`-Objekt. Ich nehme ein `Vector2`-Objekt (Zeile [23](#)), 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()` werden nun beide Koordinaten berücksichtigt und aktualisiert. Bewegen und Stehenbleiben wird einfach dadurch erreicht, dass ich die Geschwindigkeit in `start()` auf 100 bzw. in `stop()` auf 0 setze.

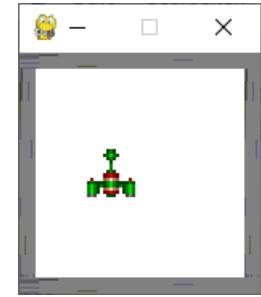


Abb. 2.26: Ränder

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

```

14  class Defender(pygame.sprite.Sprite):
15
16      def __init__(self) -> None:
17          super().__init__()
18          self.image = pygame.image.load("images/defender01.png").convert_alpha()
19          self.image = pygame.transform.scale(self.image, (30, 30))
20          self.rect = self.image.get_rect()
21          self.rect.center = Settings.WINDOW.center # 
22          self.position: pygame.math.Vector2 = pygame.math.Vector2(self.rect.left,
23              self.rect.top)
23          self.direction: pygame.math.Vector2 = pygame.math.Vector2(1, 0) # 2 Dimensionen
24          self.change_direction("right")
25          self.change_direction("start")
26
27      def update(self, *args: Any, **kwargs: Any) -> None:
28          if "action" in kwargs.keys():
29              if kwargs["action"] == "move":
30                  self.position = self.position + self.speed * Settings.DELTATIME *
31                      self.direction
32                  self.rect.left, self.rect.top = round(self.position.x),
33                      round(self.position.y)
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 == "right":
41              self.direction.x, self.direction.y = (1, 0)
42          elif direction == "left":
43              self.direction.x, self.direction.y = (-1, 0)
44          elif direction == "up":
45              self.direction.x, self.direction.y = (0, -1)

```

```

44     elif direction == "down":
45         self.direction.x, self.direction.y = (0, 1)
46     elif direction == "stop":
47         self.speed = 0
48     elif direction == "start":
49         self.speed = 100

```

Die Klasse **Border** wird trivialerweise so erweitert, dass alle vier Seiten der Spielfläche nun durch eine Steinwand begrenzt werden (Quelltext 2.36). 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 2.36: Bewegung durch Tastatur steuern (2), **Border**

```

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

```

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

Quelltext 2.37: Bewegung durch Tastatur steuern (3), **Game**-Konstruktor

```

75 class Game(object):
76
77     def __init__(self) -> None:
78         os.environ['SDL_VIDEO_WINDOW_POS'] = "10, 50"
79         pygame.init()
80         self.screen = pygame.display.set_mode(Settings.WINDOW.size)
81         pygame.display.set_caption("Sprite")
82         self.clock = pygame.time.Clock()
83         self.defender = pygame.sprite.GroupSingle(Defender())
84         self.all_border = pygame.sprite.Group()
85         self.all_border.add(Border('left'))
86         self.all_border.add(Border('right'))
87         self.all_border.add(Border('top'))
88         self.all_border.add(Border('down'))
89         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 108) 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 2.38: Bewegung durch Tastatur steuern (4), `Game.watch_for_events()`

```

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

```

`K_ESCAPE`

Fangen wir mit der Boss-Taste an. In Zeile 109 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 111ff. 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 119 gestoppt.

`event.mod`
`KMOD_LSHIFT`

Um den Einsatz der Shift-Taste (Umschalttaste) mal zu demonstrieren, habe ich hier das `r` doppelt belegt (Zeile 122). 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://www.pygame.org/docs/ref/key.html>
- `pygame.KEYDOWN`, `pygame.KEYUP`:
<https://www.pygame.org/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>		Leerzeichen (space)
<code>K_EXCLAIM</code>	<code>!</code>	Ausrufezeichen
<code>K_QUOTEDBL</code>	<code>"</code>	Gänsefußchen
<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_COMMA</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
<code>K_6</code>	<code>6</code>	6
<code>K_7</code>	<code>7</code>	7
<code>K_8</code>	<code>8</code>	8

Tabelle 2.5: Liste von vordefinierten Tastaturkonstanten (Fortsetzung)

Konstante	Bedeutung	Beschreibung
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
K_x	x	x
K_y	y	y
K_z	z	z

Tabelle 2.5: Liste von vordefinierten Tastaturkonstanten (Fortsetzung)

Konstante	Bedeutung	Beschreibung
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
K_F11		F11
K_F12		F12
K_F13		F13

Tabelle 2.5: Liste von vordefinierten Tastaturkonstanten (Fortsetzung)

Konstante	Bedeutung	Beschreibung
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 Alterntivtaste
K_LALT		Linke Alterntivtaste
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		Bildschirmdruck/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 Alterntivtaste
KMOD_RALT	Rechte Alterntivtaste
KMOD_ALT	Linke oder rechte Alterntivtaste oder beide
KMOD_LMETA	Linke Metataste

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

Konstante	Beschreibung
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 ihn 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 2.39: 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 2.40: 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 = 'Hello World!') -> 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 50). Darunter soll durch `info` ausgegeben werden, mit welcher Schriftgröße und -farbe der Gruß erzeugt wurde (Zeile 50).

Quelltext 2.41: Text mit Fonts ausgeben (3), Hauptprogramm

```

43 def main():
44     os.environ['SDL_VIDEO_WINDOW_POS'] = "500, 150"
45     pygame.init()
46     clock = pygame.time.Clock()
47     screen = pygame.display.set_mode(Settings.WINDOW.size)
48     pygame.display.set_caption("Textausgabe mit Fonts")
49
50     hello = TextSprite(24, [255, 255, 255], (Settings.WINDOW.center)) # Gruß
51     info = TextSprite(12, [255, 0, 0], (Settings.WINDOW.centerx, Settings.WINDOW.bottom-20))
52         # Fontinfo
53     all_sprites = pygame.sprite.Group()
54     all_sprites.add(hello, info)
55
55     running = True
56     while running:
57         clock.tick(Settings.FPS)
58         for event in pygame.event.get():
59             if event.type == QUIT:
60                 running = False
61             elif event.type == KEYDOWN:
62                 if event.key == K_ESCAPE:
63                     running = False
64                 elif event.key == K_KP_PLUS or event.key == K_PLUS:      # Größer
65                     hello.fontsize_update(+1)
66                 elif event.key == K_KP_MINUS or event.key == K_MINUS:  # Kleiner
67                     hello.fontsize_update(-1)
68                 elif event.key == K_r:
69                     if event.mod & KMOD_SHIFT:
70                         hello.fontcolor_update((-1, 0, 0))           # Weniger Rot
71                     else:
72                         hello.fontcolor_update((+1, 0, 0))           # Mehr Rot
73                 elif event.key == K_g:
74                     if event.mod & KMOD_SHIFT:
75                         hello.fontcolor_update((0, -1, 0))           # Weniger Grün
76                     else:
77                         hello.fontcolor_update((0, +1, 0))           # Mehr Grün
78                 elif event.key == K_b:
79                     if event.mod & KMOD_SHIFT:
80                         hello.fontcolor_update((0, 0, -1))           # Weniger Blau
81                     else:
82                         hello.fontcolor_update((0, 0, +1))           # Mehr Blau
83
84             info.text = f"size={hello.fontsize}, r={hello.fontcolor[0]}, g={hello.fontcolor[1]},"
85             b={hello.fontcolor[2]}"
86             all_sprites.update()
87             screen.fill((200, 200, 200))
88             all_sprites.draw(screen)
89             pygame.display.flip()
90
90     pygame.quit()

```

Dieser Gruß kann durch die Plus- und Minus-Tasten in seiner Größe verändert werden (Zeile 64ff.). Die Tasten **r**, **g** und **b** werden dazu verwendet, den jeweiligen Farbkanal zu manipulieren. Der Großbuchstabe erhöht den Wert (z.B. in Zeile 70), der Kleinbuchstabe reduziert ihn (z.B. in Zeile 72).

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

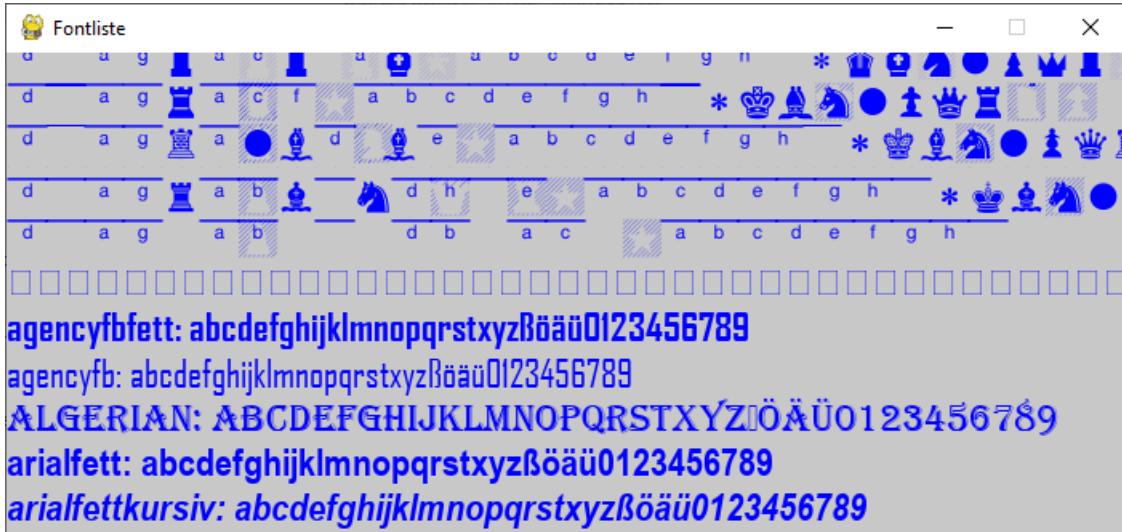


Abbildung 2.28: Fontliste

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.

Quelltext 2.42: 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 2.43: 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 70) 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 2.44: Fontliste (3), Hauptprogramm (1)

```

61 def main():
62
63     os.environ['SDL_VIDEO_WINDOW_POS'] = "650, 40"
64
65     pygame.init()
66     clock = pygame.time.Clock()
67     screen = pygame.display.set_mode(Settings.WINDOW.size)
68     pygame.display.set_caption("Fontliste")
69
70     fonts = pygame.font.get_fonts()                      # Ermittle installierte Fonts
71
72     list_of_fontsprites = pygame.sprite.Group()
73     height = 0
74     width = 0
75     for name in fonts:
76         try:
77             t = TextSprite(name, 24, [0, 0, 255])
78             t.rect.top = height
79             height += t.rect.height
80             width = t.rect.width if t.rect.width > width else width
81             list_of_fontsprites.add(t)
82         except OSError as err:
83             print(f"OS error {err}")
84         except pygame.error as perr:
85             print(f"Pygame error: {perr} with font {name}")
86
87     bigimage = pygame.sprite.GroupSingle(BigImage())
88     bigimage.sprite.create_image(width, height)
89     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 89).

Quelltext 2.45: Fontliste, Hauptprogramm (2)

```

91     running = True
92     while running:
93         clock.tick(60)
94         for event in pygame.event.get():
95             if event.type == QUIT:
96                 running = False
97             elif event.type == KEYDOWN:
98                 if event.key == K_ESCAPE:
99                     running = False
100                if event.key == K_UP:
101                    bigimage.update(-Settings.WINDOW.height//3)
102                if event.key == K_DOWN:
103                    bigimage.update(Settings.WINDOW.height//3)
104
105                bigimage.draw(screen)
106                pygame.display.flip()
107
108    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://www.pygame.org/docs/ref/font.html>
- `pygame.font.get_default_font()`:
https://www.pygame.org/docs/ref/font.html#pygame.font.get_default_font
- `pygame.font.get_fonts()`:
https://www.pygame.org/docs/ref/font.html#pygame.font.get_fonts
- `pygame.font.match_font()`:
https://www.pygame.org/docs/ref/font.html#pygame.font.match_font
- `pygame.font.Font.render()`:
`pygame.Surface.subsurface()`:
<https://www.pygame.org/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](#)).

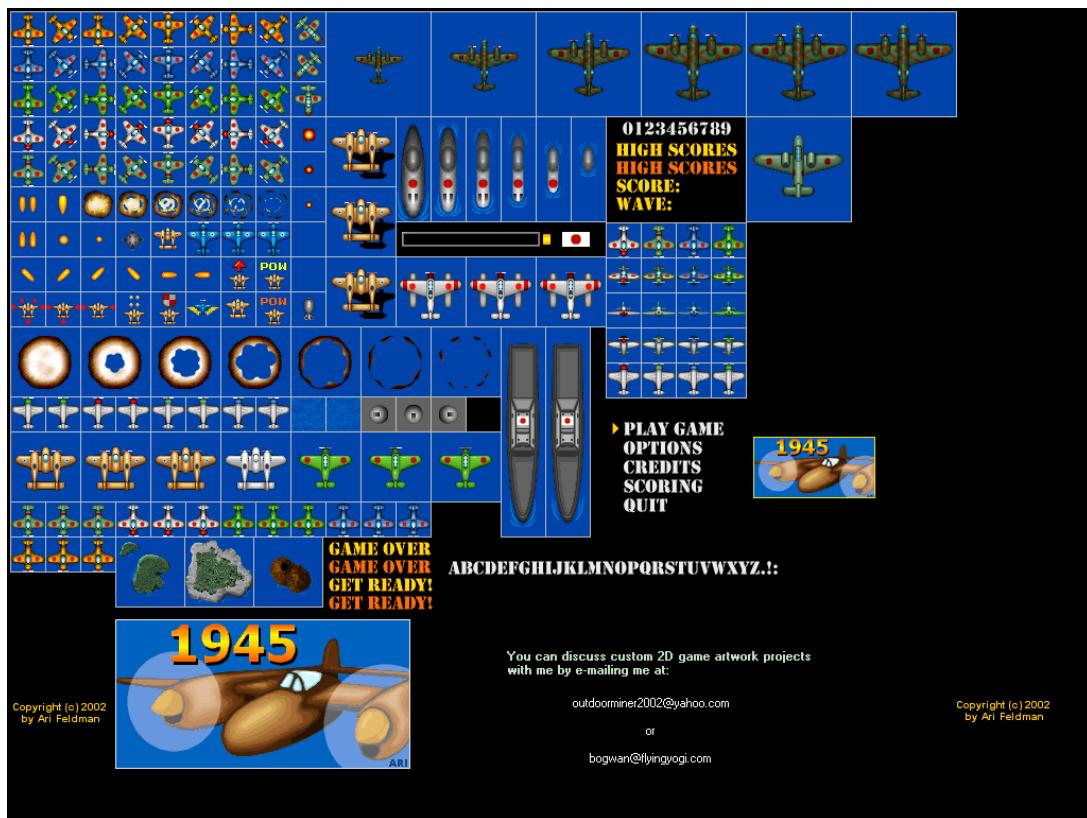


Abbildung 2.29: Beispiel für eine Spritelib

Der erste Teil von Quelltext 2.46 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 2.46: 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
15    @staticmethod
16    def filepath(name: str) -> str:
17        return os.path.join(Settings.PATH['file'], name)
18
19    @staticmethod
20    def imagepath(name: str) -> str:
21        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*.

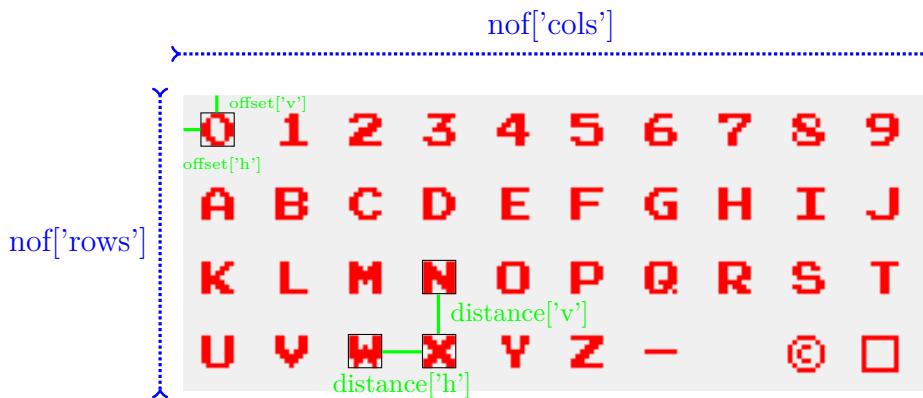


Abbildung 2.30: Bedeutung der Angaben in Spritelib

Quelltext 2.47: Textbitmaps (2), Spritelib

```

24 class Spritelib(pygame.sprite.Sprite):
25
26     def __init__(self, filename: str) -> None:
27         super().__init__()
28         self.image = pygame.image.load(Settings.imagepath(filename)).convert()
29         self.rect = self.image.get_rect()
30         self.nof = {'rows': 4, 'cols': 10}
31         self.letter = {'width': 18, 'height': 18}
32         self.offset = {'h': 6, 'v': 6}
33         self.distance = {'h': 14, 'v': 14}
34
35     def draw(self, screen: pygame.surface.Surface) -> None:
36         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 2.48: Textbitmaps (3): Konstruktor von `Letters`

```

39 class Letters(object):
40
41     def __init__(self, spritelib: Spritelib, colornumber: int) -> None:
42         super().__init__()
43         self.spritelib = spritelib
44         self.letters: dict[str, pygame.surface.Surface] = {}
45         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 51 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 54 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 55 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 56 wird die obere Koordinate und in Zeile 58 die linke Koordinate der Position berechnet. Wenn Sie Zeile 54 verstanden haben, sollten diese beiden Berechnungen keine Schwierigkeiten mehr bereiten. Höhe und Breite in Zeile 58 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 2.49: Textbitmaps (4): `create_letter_bitmap()` von Letters

```

47     def create_letter_bitmap(self, colordnumber: int):
48         lettername = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
49                     'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
50                     'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
51                     'u', 'v', 'w', 'x', 'y', 'z', ' ', 'copy', 'square') #
52         index = 0
53         startpos = (self.spritelib.offset['h'], self.spritelib.offset['v'] + colordnumber *
54                     self.spritelib.nof['rows'] *
55                     * (self.spritelib.letter['height'] + self.spritelib.distance['v'])) #
56         for row in range(self.spritelib.nof['rows']):                      # Zeilen
57             for col in range(self.spritelib.nof['cols']):                      # Spalten
58                 left = startpos[0] + col * (self.spritelib.letter['width'] +
59                     self.spritelib.distance['h']) #
60                 top = startpos[1] + row * (self.spritelib.letter['height'] +
61                     self.spritelib.distance['v']) #
62                 width, height = self.spritelib.letter.values()      # Größe
63                 r = pygame.rect.Rect(left, top, width, height)
64                 self.letters[lettername[index]] = self.spritelib.image.subsurface(r) #
65                 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 2.50: Textbitmaps (5): `get_letter()` und `get_text()` von Letters

```

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

```

```

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

```

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 99: 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 101: 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 2.51: Textbitmaps (6): `TextBitmaps`

```

80 class TextBitmaps(object):
81
82     def __init__(self) -> None:
83         pygame.init()
84         self.screen = pygame.display.set_mode(Settings.WINDOW.size)
85         pygame.display.set_caption('Textausgabe mit Bitmaps')
86         self.clock = pygame.time.Clock()
87         self.filename = "chars.png"
88         self.running = False
89         self.input = ""
90
91     def watch_for_events(self) -> None:
92         for event in pygame.event.get():
93             if event.type == QUIT:
94                 self.running = False
95             elif event.type == KEYDOWN:
96                 if event.key == K_ESCAPE:
97                     self.running = False
98                 elif event.key == pygame.K_BACKSPACE:
99                     self.input = self.input[:-1]           # Letztes Zeichen abschneiden
100                else:
101                    self.input += event.unicode        # Tastaturwert als unicode-Zeichen
102
103    def run(self) -> None:
104        spritelib = Spritelib(self.filename)
105        letters = Letters(spritelib, 2)
106        self.running = True
107        while self.running:
108            self.clock.tick(60)
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
115

```
    pygame.quit()
```

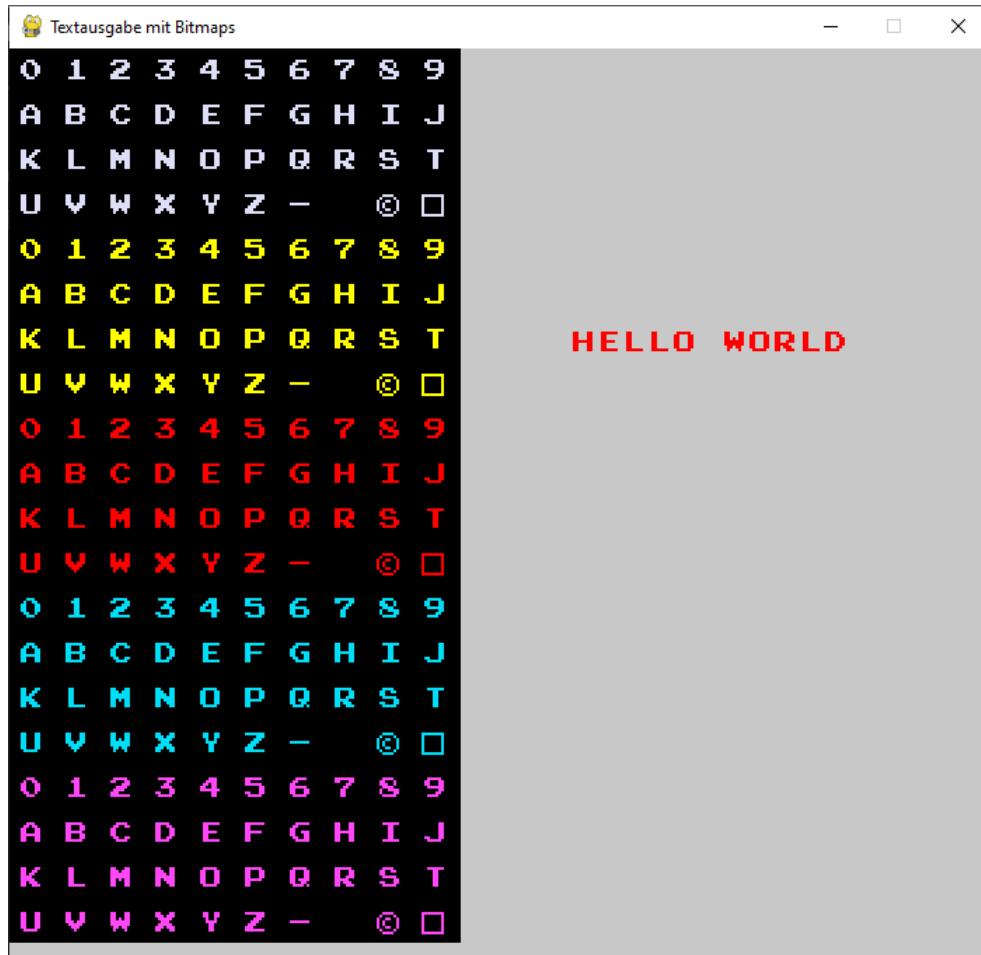


Abbildung 2.31: Textausgabe mit Bitmaps

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://www.pygame.org/docs/ref/event.html>
- `pygame.Surface.subsurface()`:
<https://www.pygame.org/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

```

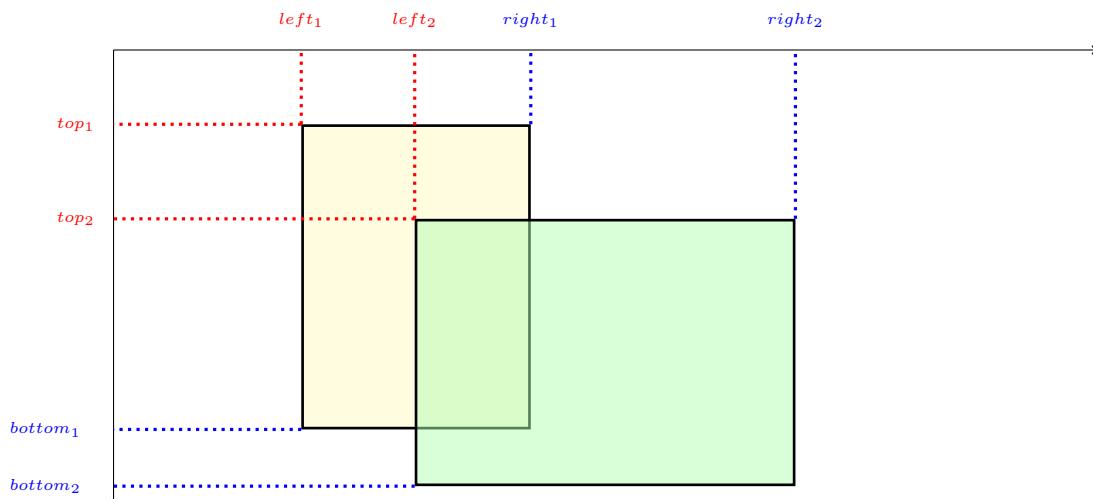


Abbildung 2.32: Kollisionserkennung mit Rechtecken

- **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: $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} < r_1 + r_2$.

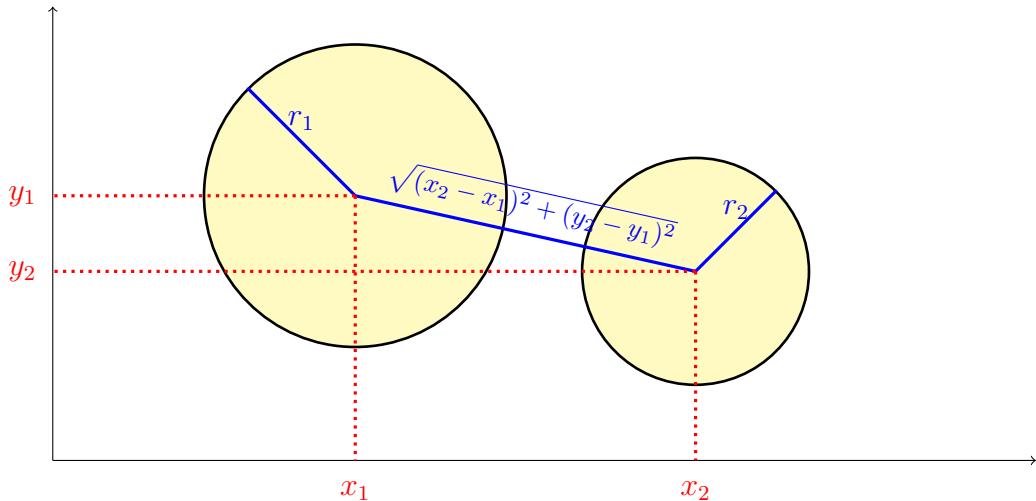


Abbildung 2.33: Kollisionserkennung mit Kreisen

- **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.

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.

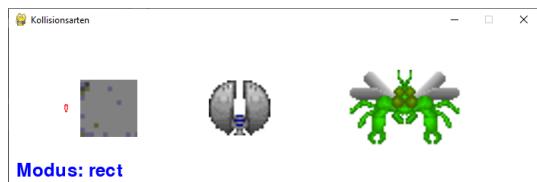
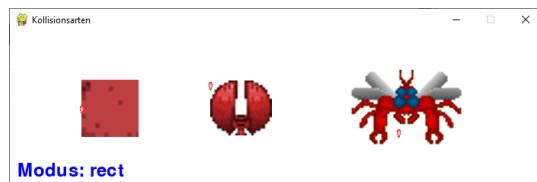
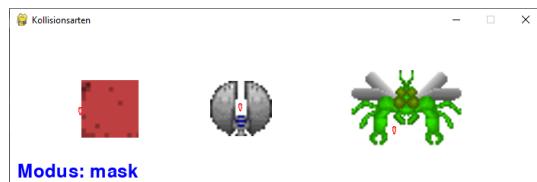


Abbildung 2.34: Vier Sprites

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 2.52: 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 2.53: 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 2.54: 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 2.55: 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 2.56: 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.clock.tick(Settings.FPS)
90             self.watch_for_events()
91             self.update()
92             self.draw()
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 2.57: 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 2.58: 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 2.59: 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 2.60: 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 2.61: Kollisionsarten (10): Der Aufruf von Game

```

155     def main():
156         game = Game()
157         game.run()
158
159
160     if __name__ == '__main__':
161         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://www.pygame.org/docs/ref/mask.html#pygame.mask.from_surface
- `pygame.sprite.collide_circle()`:
https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.collide_circle
- `pygame.sprite.collide_mask()`:
https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.collide_mask
- `pygame.sprite.collide_rect()`:
https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.collide_rect
- `pygame.sprite.spritecollide()`:
<https://www.pygame.org/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 2.62ff. 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 2.62: 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 2.63: 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.Rect = self.image.get_rect()
32         self.rect.topleft = (10, 10)
33         self.direction = 1
34         self.position = pygame.math.Vector2(self.rect.left, self.rect.top)
35         self.speed = pygame.math.Vector2(150, 0)
36
37     def update(self, *args: Any, **kwargs: Any) -> None:
38         newpos = self.position + (self.speed * Settings.DELTATIME * self.direction)
39         self.rect.left = round(newpos.x)
40         if self.rect.left < 10 or self.rect.right >= Settings.WINDOW.right - 10:
41             self.direction *= -1
42         self.position += (self.speed * Settings.DELTATIME * self.direction)
43         self.rect.left = round(self.position.x)

```

kill()

Auch `Bullet` ist in weiten Teilen eine Wiederholung. Interessant dürfte Zeile 61 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 2.64: Zeitsteuerung (3), Version 1.0: `Bullet`

```

46 class Bullet(pygame.sprite.Sprite):
47
48     def __init__(self, picturefile: str, startpos: Tuple[int, int]) -> None:
49         super().__init__()
50         self.image = pygame.image.load(Settings.imagepath(picturefile)).convert_alpha()
51         self.rect: pygame.rect.Rect = self.image.get_rect()
52         self.rect.center = startpos
53         self.direction = 1
54         self.position = pygame.math.Vector2(self.rect.left, self.rect.top)
55         self.speed = pygame.math.Vector2(0, 100)
56
57     def update(self, *args: Any, **kwargs: Any) -> None:
58         self.position += (self.speed * Settings.DELTATIME * self.direction)
59         self.rect.top = round(self.position.y)
60         if self.rect.top > Settings.WINDOW.bottom - 30:
61             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 84ff. 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.

ck() deltatime

Quelltext 2.65: Zeitsteuerung (4), Version 1.0: Konstruktor und `run()` von `Game`

```

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

```

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

Quelltext 2.66: Zeitsteuerung (5), Version 1.0: `watch_for_events()` und `draw()` von `Game`

```

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

```

Die Methode `update()` ist nur bzgl. Zeile 105 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 Enemys. Das horizontale Zentrum von Feuerball und Enemy soll gleich sein. Das vertikale Zentrum ist etwas nach unten verschoben; sieht besser aus.

Quelltext 2.67: Zeitsteuerung (6), Version 1.0: `update()` und `new_bullet()` von `Game`

```

104     def update(self) -> None:
105         self.new_bullet()                                     # Feuerballabwurf
106         self.all_bullets.update()
107         self.enemy.update()
108
109     def new_bullet(self) -> None:
110         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 84 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 75 und Zeile 76).

Quelltext 2.68: Zeitsteuerung (7), Version 1.1: Konstruktor von `Game`

```

74     self.all_bullets = pygame.sprite.Group()
75     self.time_counter = 0                                # Zähler
76     self.time_range = 30                                # Obergrenze
77     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 2.69: Zeitsteuerung (8), Version 1.1: `new_bullet()` von `Game`

```

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

```

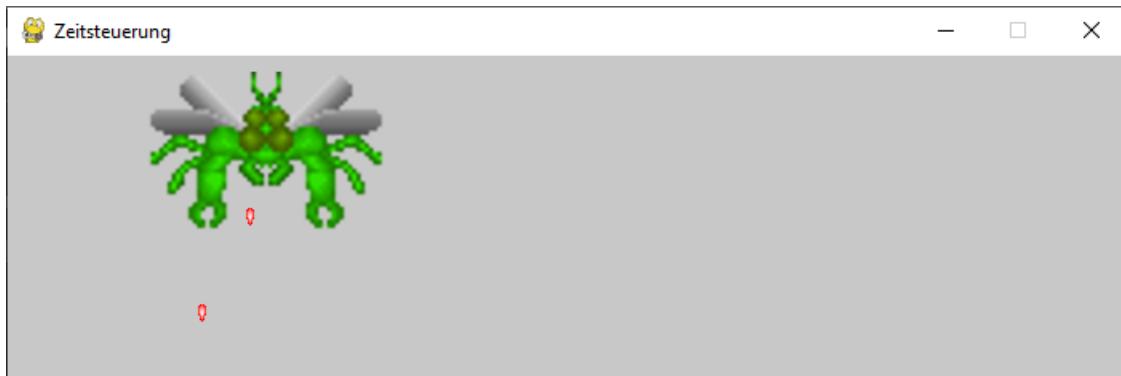


Abbildung 2.39: Feuerball mit Zeitsteuerung

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 75 die aktuelle Anzahl der *ms* seit Programmstart gemessen und in Zeile 76 wird festgehalten, wie viele *ms* ein Zeitintervall dauert; wir wollen alle halbe Sekunde einen Feuerball abwerfen, also 500.

Quelltext 2.70: Zeitsteuerung (9), Version 1.2: Konstruktor von `Game`

```

74     self.all_bullets = pygame.sprite.Group()
75     self.time_stamp = pygame.time.get_ticks()           # Zeitpunkt festhalten
76     self.time_duration = 500                            # Intervalldauer
77     self.running = False

```

Danach wird in `new_bullet()` abgeprüft, ob das Intervallende erreicht wurde. In Zeile 112 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 112.

Quelltext 2.71: Zeitsteuerung (10), Version 1.2: `new_bullet()` von Game

```

111  def new_bullet(self) -> None:
112      if pygame.time.get_ticks() >= self.time_stamp + self.time_duration: #
113          self.all_bullets.add(Bullet("shoot.png", self.enemy.sprite.rect.move(0,
114                                         20).center))
115          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 2.72: 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 91); die beiden Variablen von eben werden nicht mehr gebraucht.

Quelltext 2.73: Zeitsteuerung (12), Version 1.3: `Timer`-Objekt erzeugen

```

90      self.all_bullets = pygame.sprite.Group()
91      self.bullet_timer = Timer(500)                      # Timer ohne Verzögerung
92      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 127 abgefragt, ob das Intervallende erreicht wurde und fertig!

Quelltext 2.74: Zeitsteuerung (13), Version 1.3: Timer-Objekt verwenden

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

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://www.pygame.org/docs/ref/time.html#pygame.time.get_ticks
- `pygame.sprite.Sprite.kill()`:
<https://www.pygame.org/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

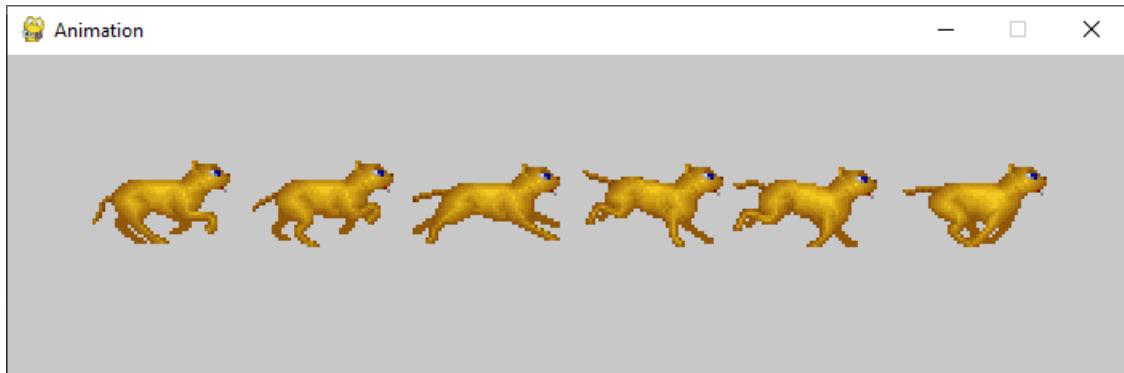


Abbildung 2.40: Animation einer Katze: Einzelsprites

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 2.75 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 2.75: 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_KP_MINUS, K_KP_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 2.76: 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 2.77: 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 2.78: 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_KP_PLUS:
111                 self.cat.sprite.update(animation_delta=-10)
112             elif event.key == K_KP_MINUS:
113                 self.cat.sprite.update(animation_delta=10)

```

Der restliche Quelltext (Quelltext 2.79) 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 2.79: 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:
122             {self.cat.sprite.animation_time.duration}", True, "white")
123         text_rect = text_image.get_rect()
124         text_rect.centerx = Settings.WINDOW.centerx
125         text_rect.bottom = Settings.WINDOW.bottom - 50
126         self.screen.blit(text_image, text_rect)
         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 eine einfache Animationsklasse (siehe Quelltext 2.80 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 2.80: 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 2.81: 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     self.image: pygame.surface.Surface = self.animation.next()
85     self.rect: pygame.rect.Rect = self.image.get_rect()
86     self.rect.center = Settings.WINDOW.center
87
88     def update(self, *args: Any, **kwargs: Any) -> None:
89         if "animation_delta" in kwargs.keys():
90             self.change_animation_time(kwargs["animation_delta"])
91         self.image = self.animation.next()
92         # implement game logic here
93
94     def change_animation_time(self, delta: int) -> None:
95         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 bin¹.

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 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.

¹ Was für eine Grammatik! Aber ich kann mich rausreden: Im westfälischen Dialekt gibt es ähnlich wie im Englischen eine Verlaufsform :-)

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.

Quelltext 2.82: 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)
90          self.rect.centery = random.randint(self.rect.height,
91                                              Settings.WINDOW.height-self.rect.height)
92          self.anim = Animation([f"explosion0{i}.png" for i in range(1, 5)], False, 50)  #
93          self.timer_lifetime = Timer(random.randint(100, 2000), False)      # Lebenszeit
94          self.bumm = False
95
96      def update(self, *args: Any, **kwargs: Any) -> None:
97          if self.timer_lifetime.is_next_stop_reached():
98              self.bumm = True
99          if self.bumm:
100              self.image = self.anim.next()
101              c = self.rect.center
102              self.rect = self.image.get_rect()
103              self.rect.center = c
104          if self.anim.is-ended():
105              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 2.83: 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.

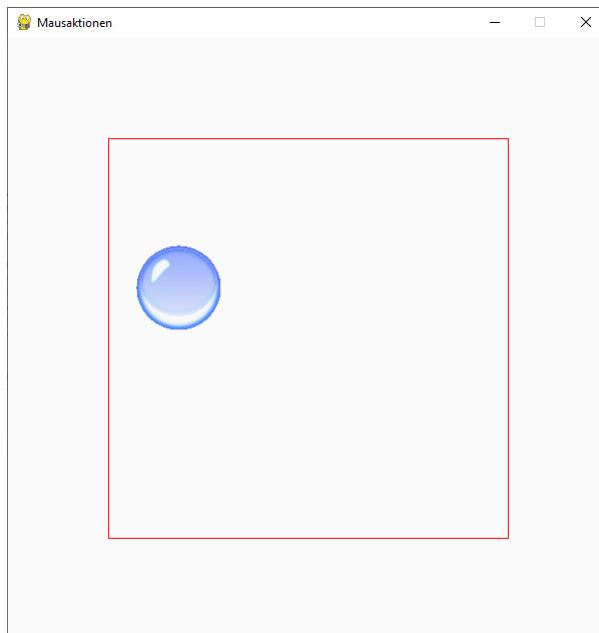


Abbildung 2.42: Mausaktionen

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 70 wird das `Ball`-Objekt erzeugt.

Quelltext 2.84: Mausaktionen: Statics und Konstruktor von Game

```

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

```

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

Quelltext 2.85: Mausaktionen: Game.run()

```

73     def run(self) -> None:
74         time_previous = time()
75         while self._running:
76             self.watch_for_events()
77             self.update()
78             self.draw()
79             self._clock.tick(Game.DELTATIME)
80             time_current = time()
81             Game.deltatime = time_current - time_previous
82             time_previous = time_current
83             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 92 wird der `event.type` abgefragt und anschließend wird ermittelt, welche Maustaste denn gedrückt wurde.

MOUSE-
BUTTON-
DOWN
MOUSE-
BUTTONUP

event.button

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 93), 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 97). Das Mausrad wird ebenfalls wie ein Mausbutton verarbeitet. Je nach Drehrichtung wird dabei ein anderer Zahlenkode zurückgeliefert (siehe Zeile 99 und Zeile 101). Wird das Mausrad gedrückt – also geklickt – soll ja das Spiel beendet werden. In Zeile 95 wird dies abgefragt und umgesetzt.

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

event.pos

Quelltext 2.86: Mausaktionen: Game.watch_for_events()

```

85  def watch_for_events(self) -> None:
86      for event in pygame.event.get():
87          if event.type == QUIT:
88              self._running = False
89          elif event.type == KEYDOWN:
90              if event.key == K_ESCAPE:
91                  self._running = False
92          elif event.type == MOUSEBUTTONDOWN:
93              if event.button == 1:                                # Maustaste gedrückt
94                  self._ball.update(rotate=90)                      # left
95              elif event.button == 2:                             # middle
96                  self._running = False
97              elif event.button == 3:                                # right
98                  self._ball.update(rotate=-90)
99              elif event.button == 4:                                # scroll up
100                 self._ball.update(scale=2)
101             elif event.button == 5:                                # scroll down
102                 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 108 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 2.87: Mausaktionen: Game.update() und Game.draw()

```

104  def update(self):
105      self._ball.update(center=pygame.mouse.get_pos())
106      pygame.mouse.set_visible(
107          not Game.INNER_RECT.collidepoint(pygame.mouse.get_pos()))           # Unsichtbar?
108      self._ball.update(go=True)
109
110
111  def draw(self) -> None:
112      self._screen.fill((250, 250, 250))
113      pygame.draw.rect(self._screen, "red", Game.INNER_RECT, 1)
114      self._ball.draw(self._screen)
115      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 20, Zeile 31, Zeile 34 oder Zeile 37. 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 2.88: Mausaktionen: Ball

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

```

35         self.resize(kwargs["scale"])
36
37     if "center" in kwargs.keys():
38         self.set_center(kwargs["center"])
39
40     def draw(self, screen: pygame.surface.Surface) -> None:
41         screen.blit(self.image, self.rect)
42
43     def rotate(self, angle: float) -> None:
44         self.image_orig = pygame.transform.rotate(self.image_orig, angle)
45
46     def resize(self, delta: int) -> None:
47         if self.rect.width < Game.INNER_RECT.width:
48             if self._scale + delta > 0:
49                 self._scale += delta
50
51     def set_center(self, center: Tuple[int, int]) -> None:
52         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.mouse.get_pos()`:
https://www.pygame.org/docs/ref/mouse.html#pygame.mouse.get_pos
- `pygame.mouse.set_visible()`:
https://www.pygame.org/docs/ref/mouse.html#pygame.mouse.set_visible
- `pygame.MOUSEBUTTONDOWN`, `pygame.MOUSEBUTTONUP`:
<https://www.pygame.org/docs/ref/event.html>
- Liste der Mausbuttons: Tabelle 2.7
- `pygame.Rect.collidepoint()`:
<https://www.pygame.org/docs/ref/rect.html#pygame.Rect.collidepoint>
- `pygame.transform.rotate()`:
<https://www.pygame.org/docs/ref/transform.html#pygame.transform.rotate>

Tabelle 2.7: Liste der Mausbuttons

Konstante	Beschreibung
0	nicht definiert
1	linke Maustaste

Tabelle 2.7: Liste der Mausbuttons (Fortsetzung)

Konstante	Beschreibung
2	mittlere Maustaste/Mausrad
3	rechte Maustaste
4	Mausrad zu sich drehen (up)
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 2.89: 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 K_ESCAPE, KEYDOWN, KEYUP, QUIT, K_f, K_j, K_p
7
8
9 class Settings:
10     WINDOW: pygame.rect.Rect = pygame.rect.Rect(0, 0, 600, 800)    # Rect
11     FPS = 60
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     PATH["sound"] = os.path.join(PATH["file"], "sounds")
16     START_DISTANCE = 20
17
18     @staticmethod
19     def get_file(filename: str) -> str:
20         return os.path.join(Settings.PATH["file"], filename)
21
22     @staticmethod
23     def get_image(filename: str) -> str:
24         return os.path.join(Settings.PATH["image"], filename)
25
26     @staticmethod
27     def get_sound(filename: str) -> str:
28         return os.path.join(Settings.PATH["sound"], filename)

```

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 33 implizit durch `pygame.init()`. In Zeile 40 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.

`get_volume()`

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 45 mit `pygame.mixer.music.set_volume()` festgelegt wurde, in der Zeile 46. 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.

`Hintergrund-musik`

`set_volume()`
`play()`

`fade`

`Soundeffekte`

Für Soundeffekte wird jeweils ein eigenes `Sound`-Objekt angelegt (Zeile 47f.). 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 2.90: Sound: Konstruktor und `sounds()` von Game

```

31 class Game:
32     def __init__(self) -> None:
33         pygame.init()                                     # Auch mixer
34         self.screen = pygame.display.set_mode(Settings.WINDOW.size)
35         pygame.display.set_caption('Fingerübung "Sound"')
36         self.clock = pygame.time.Clock()
37         self.font_bigsize = pygame.font.Font(pygame.font.get_default_font(), 40)
38         self.running = True
39         self.pause = False
40         self.volume: float = 0.1                         # Lautstärke
41         self.sounds()
42
43     def sounds(self) -> None:
44         pygame.mixer.music.load(Settings.get_sound("Lucifer.mid"))
45         pygame.mixer.music.set_volume(self.volume)        # Lautstärke
46         pygame.mixer.music.play(-1, 0.0)                  # Endlos abspielen
47         self.bubble: pygame.mixer.Sound =
48             pygame.mixer.Sound(Settings.get_sound("plopp1.mp3")) #
49         self.clash: pygame.mixer.Sound = pygame.mixer.Sound(Settings.get_sound("glas.wav"))

```

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

Quelltext 2.91: Sound: `watch_for_events()` von Game

```

50     def watch_for_events(self) -> None:
51         for event in pygame.event.get():
52             if event.type == QUIT:
53                 self.running = False
54             elif event.type == KEYDOWN:
55                 if event.key == K_ESCAPE:
56                     self.running = False
57             elif event.type == KEYUP:
58                 if event.key == K_f:
59                     self.music_start_stop(fadeout=5000)
60                 elif event.key == K_j:
61                     self.music_start_stop(loop=-1)
62                 elif event.key == K_p:
63                     self.pause_alter()
64             elif event.type == pygame.MOUSEBUTTONDOWN:
65                 if event.button == 1:                                # left
66                     self.sound_play(bubble=True)
67                 elif event.button == 3:                                # right
68                     self.sound_play(clash=True)
69                 elif event.button == 4:                                # up
70                     self.volume_alter(0.05)
71                 elif event.button == 5:                                # down
72                     self.volume_alter(-0.05)

```

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 2.92: Sound: `sound_play()` von Game

```

74     def sound_play(self, **kwargs: Any) -> None:
75         if "bubble" in kwargs.keys():
76             self.bubble.play()
77         if "clash" in kwargs.keys():
78             self.clash.play()

```

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 2.93: Sound: `music_start_stop()` von Game

```

80     def music_start_stop(self, **kwargs: Any) -> None:

```

```

81     if "fadeout" in kwargs.keys():
82         pygame.mixer.music.fadeout(kwargs["fadeout"])
83     if "loop" in kwargs.keys():
84         pygame.mixer.music.play(kwargs["loop"], 0.0)

```

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 91 das Flag `pause` umgelegt.

`pause()`
`unpause()`

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

```

86     def pause_alter(self) -> None:
87         if self.pause:
88             pygame.mixer.music.unpause()
89         else:
90             pygame.mixer.music.pause()
91         self.pause = not self.pause           #

```

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` addiert (Bedenken Sie, dass ein negativer Veränderungswert hier die Lautstärke reduziert.). 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 2.95: Sound: `volume_alter()` von Game

```

93     def volume_alter(self, delta: float) -> None:
94         self.volume += delta
95         if self.volume > 1.0:
96             self.volume = 1.0
97         elif self.volume < 0.0:
98             self.volume = 0.0
99         pygame.mixer.music.set_volume(self.volume)

```

Und zum Schluss kommt der gute Rest:

Quelltext 2.96: Sound: `draw()`, `update()`, `run()` und Aufruf von Game

```

101    def draw(self) -> None:
102        self.screen.fill("black")
103
104        volume = self.font_bigsize.render(f"Lautstärke: {self.volume:3.2f}", True, "red")
105        rect = volume.get_rect()
106        rect.center = Settings.WINDOW.center
107        self.screen.blit(volume, rect)
108
109        pygame.display.flip()
110
111    def update(self):
112        pass

```

```

113
114     def run(self):
115         self.running = True
116         while self.running:
117             self.watch_for_events()
118             self.update()
119             self.draw()
120             self.clock.tick(Settings.FPS)
121         pygame.quit()
122
123
124     def main():
125         os.environ["SDL_VIDEO_WINDOW_POS"] = "10, 30"
126         Game().run()
127
128
129     if __name__ == "__main__":
130         main()

```

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 befinden. 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.

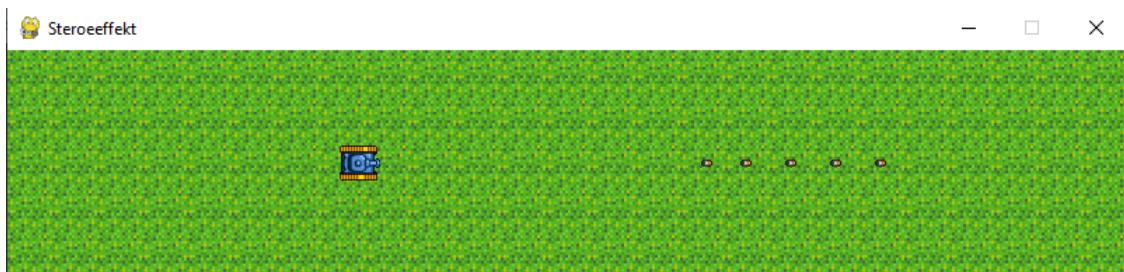


Abbildung 2.43: Sound: Stereoeffekt

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

Quelltext 2.97: 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 `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()`.

Kanal

find_channel()

play()

Quelltext 2.98: 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 = 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.position = pygame.math.Vector2(self.rect.left, self.rect.top)
71         self.speed = 50

```

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

Quelltext 2.99: Sound-Stereo: `Tank.update()`

```

72     def update(self, *args: Any, **kwargs: Any) -> None:
73         if "go" in kwargs.keys():
74             if kwargs["go"]:
75                 self.update_imageindex()
76                 self.image = self.images[self.direction][self.imageindex]
77                 if self.direction == "up" or self.direction == "left":
78                     self.speed = -50
79                 elif self.direction == "down" or self.direction == "right":
80                     self.speed = 50
81                 if self.direction == "up" or self.direction == "down":
82                     self.position.y += (self.speed * Settings.DELTATIME)
83                     self.rect.top = round(self.position.y)
84                     if self.rect.top <= Settings.WINDOW.top:
85                         self.turn("down")
86                     if self.rect.bottom >= Settings.WINDOW.bottom:
87                         self.turn("up")
88                 elif self.direction == "left" or self.direction == "right":
89                     self.position.x += (self.speed * Settings.DELTATIME)
90                     self.rect.left = round(self.position.x)
91                     if self.rect.left <= Settings.WINDOW.left:
92                         self.turn("right")
93                     if self.rect.right >= Settings.WINDOW.right:
94                         self.turn("left")
95                     self.stereo()
96             if "turn" in kwargs.keys():
97                 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.

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 2.100: Sound-Stereo: `Tank.stereo()`

```
99  def stereo(self) -> None:
100     volume_rechts = self.rect.centerx / Settings.WINDOW.width  #
101     volume_links = 1 - volume_rechts
102     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 2.101: Sound-Stereo: `Tank.turn()` und `Tank.update_imageindex()`

```
104  def turn(self, direction: str) -> None:
105      self.direction = direction
106
107  def update_imageindex(self) -> None:
108      if self.speed == 0:
109          self.imageindex = 0
110      else:
111          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 116 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 137 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 2.102: Sound-Stereo: Die Klasse Bullet

```

114 class Bullet(pygame.sprite.Sprite):
115
116     _sound_fire = None                                # Es braucht nur einen
117
118     def __init__(self, tank: Tank) -> None:
119         super().__init__()
120         bulletspeed = 300
121         number: dict[str, int] = {"left": 49, "right": 61, "up": 37, "down": 73}
122         directions = {
123             "left": (-bulletspeed, 0),
124             "right": (bulletspeed, 0),
125             "up": (0, -bulletspeed),
126             "down": (0, bulletspeed),
127         }
128         fullfilename = os.path.join(Settings.PATH["image"],
129                                     f"tankbrigade_part{number[tank.direction]}.png")
130         self.image = pygame.image.load(fullfilename).convert()
131         self.image.set_colorkey("black")
132         self.rect = self.image.get_rect()
133         self.direction = tank.direction
134         self.rect.center = tank.rect.center
135         self.speed = directions[tank.direction]
136         self.position = pygame.math.Vector2(self.rect.left, self.rect.top)
137
138         if Bullet._sound_fire == None:                      # Es braucht nur einen
139             Bullet._sound_fire = pygame.mixer.Sound(Settings.get_sound("tank_fire1.wav"))
140         volume_rechts = self.rect.centerx / Settings.WINDOW.width
141         volume_links = 1 - volume_rechts
142         self.channel: pygame.mixer.Channel = pygame.mixer.find_channel()
143         self.channel.set_volume(volume_links, volume_rechts)
144         self.channel.play(Bullet._sound_fire)

```

Der Rest des Quelltextes wird nur der Vollständigkeit wegen abgedruckt.

Quelltext 2.103: Sound-Stereo: Rest

```

154 class Game:
155
156     def __init__(self) -> None:
157         pygame.init()
158         self._screen = pygame.display.set_mode(Settings.WINDOW.size)
159         pygame.display.set_caption("Stereoeffekt")
160         self.clock = pygame.time.Clock()
161         self.ground = pygame.sprite.GroupSingle(Ground())
162         self.tankreference = Tank()
163         self.tank = pygame.sprite.GroupSingle(self.tankreference)
164         self.all_bullets = pygame.sprite.Group()
165         self.running = True
166
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.key == K_UP:
175                     self.tank.update(turn="up")
176                 elif event.key == K_DOWN:
177                     self.tank.sprite.update(turn="down")
178                 elif event.key == K_LEFT:
179                     self.tank.sprite.update(turn="left")

```

```

180         elif event.key == K_RIGHT:
181             self.tank.sprite.update(turn="right")
182         elif event.key == K_SPACE:
183             self.fire()
184
185     def fire(self) -> None:
186         if len(self.all_bullets) < 5:
187             self.all_bullets.add(Bullet(self.tankreference))
188
189     def draw(self) -> None:
190         self.ground.draw(self._screen)
191         self.tank.draw(self._screen)
192         self.all_bullets.draw(self._screen)
193         pygame.display.flip()
194
195     def update(self) -> None:
196         self.tank.update(go=True)
197         self.all_bullets.update()
198
199     def run(self) -> None:
200         time_previous = time()
201         self.running = True
202         while self.running:
203             self.watch_for_events()
204             self.update()
205             self.draw()
206             self.clock.tick(Settings.FPS)
207             time_current = time()
208             Settings.DELTATIME = time_current - time_previous
209             time_previous = time_current
210         pygame.quit()
211
212
213     def main():
214         os.environ["SDL_VIDEO_WINDOW_POS"] = "10, 30"
215         Game().run()
216
217
218     if __name__ == "__main__":
219         main()

```

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://www.pygame.org/docs/ref/music.html#pygame.mixer.Channel>
- `pygame.mixer.Channel.play()` :
<https://www.pygame.org/docs/ref/mixer.html#pygame.mixer.Channel.play>
- `pygame.mixer.Channel.set_volume()` :
https://www.pygame.org/docs/ref/mixer.html#pygame.mixer.Channel.set_volume

- `pygame.mixer.find_channel()`:
https://www.pygame.org/docs/ref/music.html#pygame.mixer.find_channel
- `pygame.mixer.init()`:
<https://www.pygame.org/docs/ref/mixer.html#pygame.mixer.init>
- `pygame.mixer.music.fadeout()`:
<https://www.pygame.org/docs/ref/music.html#pygame.mixer.music.fadeout>
- `pygame.mixer.music.get_volume()`:
https://www.pygame.org/docs/ref/music.html#pygame.mixer.music.get_volume
- `pygame.mixer.music.load()`:
<https://www.pygame.org/docs/ref/music.html#pygame.mixer.music.load>
- `pygame.mixer.music.pause()`:
<https://www.pygame.org/docs/ref/music.html#pygame.mixer.music.pause>
- `pygame.mixer.music.play()`:
<https://www.pygame.org/docs/ref/music.html#pygame.mixer.music.play>
- `pygame.mixer.music.set_volume()`:
https://www.pygame.org/docs/ref/music.html#pygame.mixer.music.set_volume
- `pygame.mixer.music.unpause()`:
<https://www.pygame.org/docs/ref/music.html#pygame.mixer.music.unpause>
- `pygame.mixer.Sound`:
<https://www.pygame.org/docs/ref/mixer.html#pygame.mixer.Sound>
- `pygame.mixer.Sound.get_volume()`:
https://www.pygame.org/docs/ref/mixer.html#pygame.mixer.Sound.get_volume
- `pygame.mixer.Sound.play()`:
<https://www.pygame.org/docs/ref/mixer.html#pygame.mixer.Sound.play>
- `pygame.mixer.Sound.set_volume()`:
https://www.pygame.org/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, Rechenzeitverschwendug.

Pygame stellt dafür das Konzept der *Dirty Sprites* zur Verfügung. 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 sich bewegt oder verschwindet; wird doch der Hintergrund eben nicht in jedem Frame neu gezeichnet.

Dirty Sprite
`self.dirty`

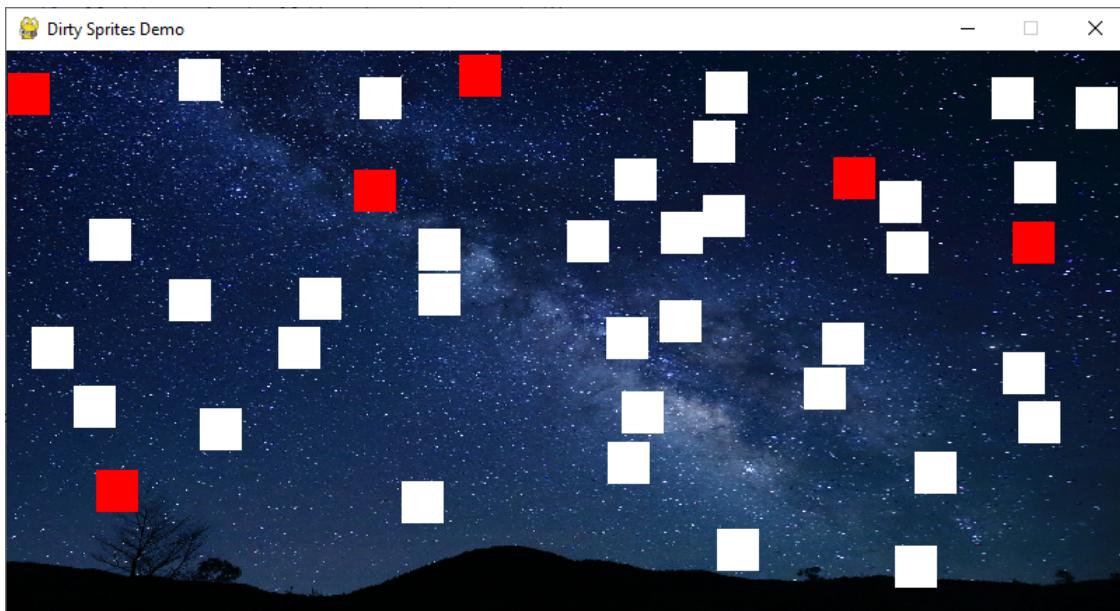


Abbildung 2.44: Dirty Sprite – Demo-Spiel

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.

Zunächst 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 2.104: 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 2.105: 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("Dirty Sprites Demo")
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")
98
99
100    def main():
101        os.environ["SDL_VIDEO_WINDOW_POS"] = "10, 30"
102        Game().run()
103
104
105    if __name__ == "__main__":
106        main()

```

Aber Rettung naht; 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 124 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)². 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

² Der Wert 1 ist die Vorbelegung für dieses Attribut; ein Setzen ist daher nicht zwingend nötig, aber wegen der besseren Verständlichkeit angebracht.

Timer die Farbe wieder von rot nach weiß abändert (siehe Zeile 35). Wie in Tabelle 2.9 auf Seite 124 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 ja der Hintergrund an dieser Stelle nachgezeichnet werden. Wie das geschieht, wird in Game geregelt.

Quelltext 2.106: 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 2.107: 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("Dirty Sprites Demo")
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.

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.

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 2.108: 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 ;-)

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 2.109: 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 2.110: 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 gemessen³. 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 2.111: 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:

```

³ Siehe https://github.com/pygame/pygame/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 unterschiedlich⁴, 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

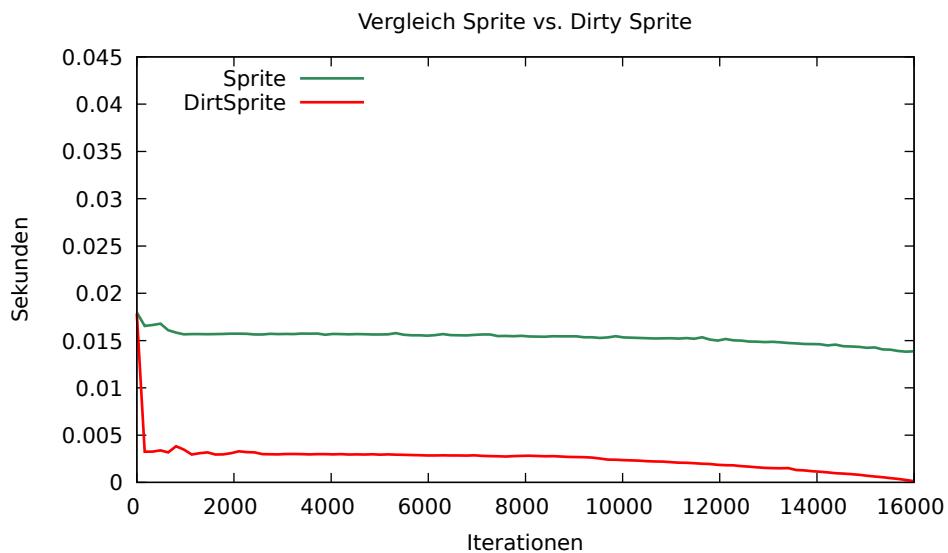


Abbildung 2.45: Performancevergleich mit Testkonfiguration 2

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

⁴ 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 Zeitverbräue 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.

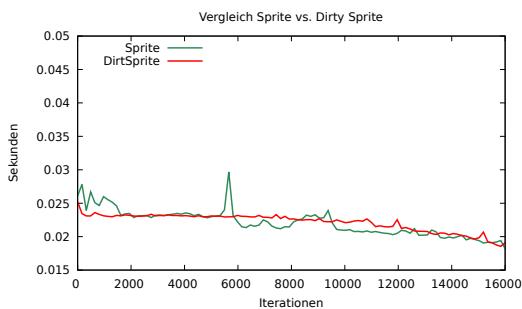


Abbildung 2.46: Testkonfiguration mit privatem PC

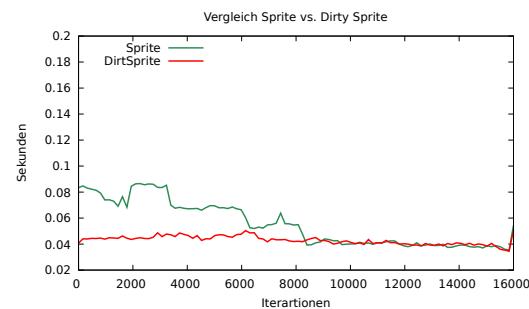


Abbildung 2.47: Testkonfiguration mit Surface Pro

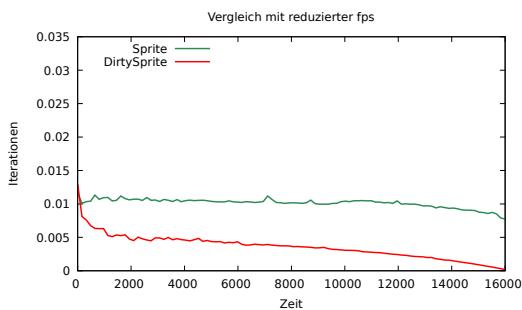


Abbildung 2.48: Testkonfiguration mit privatem PC und reduzierter fps

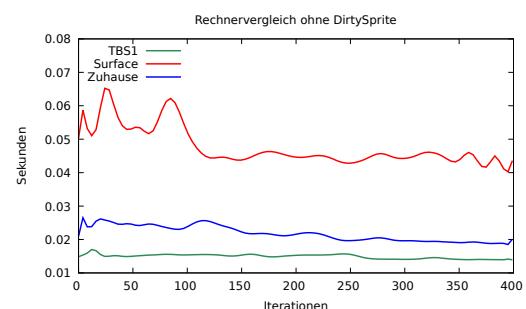


Abbildung 2.49: Bestätigung der Unterschiede

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.

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:

http://www.pygame.org/docs/ref/color_list.html

- `pygame.display.update()`:

<http://www.pygame.org/docs/ref/sprite.html#pygame.display.update>

- `pygame.sprite.DirtySprite`:

<http://www.pygame.org/docs/ref/sprite.html#pygame.sprite.DirtySprite>

- `pygame.sprite.DirtySprite.dirty`:

<http://www.pygame.org/docs/ref/sprite.html#pygame.sprite.DirtySprite>

Bedeutung siehe Tabelle 2.9.

- `pygame.sprite.LayeredDirty`:

<http://www.pygame.org/docs/ref/sprite.html#pygame.sprite.LayeredDirty>

- `pygame.sprite.LayeredDirty.clear()`:

<http://www.pygame.org/docs/ref/sprite.html#pygame.sprite.LayeredDirty.clear>

- `pygame.sprite.LayeredDirty.draw()`:

<http://www.pygame.org/docs/ref/sprite.html#pygame.sprite.LayeredDirty.draw>

- `pygame.sprite.LayeredDirty.set_timing_threshold()`:

http://www.pygame.org/docs/ref/sprite.html#pygame.sprite.LayeredDirty.set_timing_threshold

- `pygame.sprite.LayeredDirty.set_timing_threshold()`:

http://www.pygame.org/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.

Tabelle 2.9: Bedeutung von `dirty` (Fortsetzung)

Konstante	Beschreibung
2	Der Sprite wird immer neu gezeichnet. Sie wird nicht nach dem Zeichnen auf 0 gesetzt.

3 Beispielprojekte

3.1 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.

3.1.1 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 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.

Quelltext 3.1: 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

```

Die statischen Angaben zum Spiel werden hier nicht in einer separaten Klasse `Settings` abgelegt, sondern direkt in `Game`.

Quelltext 3.2: Bubbles (Requirement 1.1) – Statisches in Game

```

22 class Game:
23     window = pygame.Rect(0, 0, 1220, 1002)
24     fps = 60
25     deltatime = 1.0/fps
26     path: Dict[str, str] = {}
27     path["file"] = os.path.dirname(os.path.abspath(__file__))
28     path["image"] = os.path.join(path["file"], "images")
29     path["sound"] = os.path.join(path["file"], "sounds")
30     caption = 'Fingerübung "Bubbles"'
31
32     @staticmethod
33     def get_file(filename: str) -> str:
34         return os.path.join(Game.path["file"], filename)
35
36     @staticmethod
37     def get_image(filename: str) -> str:
38         return os.path.join(Game.path["image"], filename)
39
40     @staticmethod
41     def get_sound(filename: str) -> str:
42         return os.path.join(Game.path["sound"], filename)

```

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.

Quelltext 3.3: Bubbles (Requirement 1.2) – Background

```

9 class Background(pygame.sprite.DirtySprite):
10     def __init__(self) -> None:
11         super().__init__()
12         imagename = Game.get_image("aquarium.png")
13         self.image: pygame.surface.Surface = pygame.image.load(imagename).convert()
14         self.image = pygame.transform.scale(self.image, Game.window.size)
15         self.rect: pygame.rect.Rect = self.image.get_rect()
16         self.dirty = 1
17
18     def draw(self):
19         pygame.display.get_surface().blit(self.image, self.rect) # Holt sich screen

```

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 19. 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 3.1 auf der nächsten Seite zu sehen.

Quelltext 3.4: Bubbles (Requirement 1) – Methoden von `Game`

```

44     def __init__(self) -> None:
45         pygame.init()
46         self._screen = pygame.display.set_mode(Game.window.size)
47         pygame.display.set_caption(Game.caption)
48         self._clock = pygame.time.Clock()
49         self._background = Background()
50         self._running = True
51
52     def watch_for_events(self) -> None:
53         for event in pygame.event.get():
54             if event.type == QUIT:
55                 self._running = False
56             elif event.type == KEYDOWN:
57                 if event.key == K_ESCAPE:
58                     self._running = False
59
60     def draw(self) -> None:
61         self._background.draw()
62         pygame.display.update()
63
64     def update(self) -> None:
65         pass
66
67     def run(self) -> None:
68         time_previous = time()
69         self._running = True
70         while self._running:
71             self.watch_for_events()
72             self.update()
73             self.draw()
74             self._clock.tick(Game.fps)
75             time_current = time()
76             Game.timedelta = time_current - time_previous
77             time_previous = time_current
78         pygame.quit()

```

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.

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 nur noch der Aufruf (siehe Quelltext 3.5). Damit sind alle Unterpunkte von Requirement 1 auf Seite 126, die hier Anwendung finden, erfüllt.

Quelltext 3.5: Bubbles (Requirement 1) – Aufruf

```

81 def main():
82     os.environ["SDL_VIDEO_WINDOW_POS"] = "10, 30"
83     game = Game()
84     game.run()
85
86
87 if __name__ == "__main__":
88     main()

```



Abbildung 3.1: Bubbles: Hintergrundbild (aquarium.png)

3.1.2 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 3.2 verwendet. Die zufällige Position muss noch eingeschränkt werden. Das Aquarium füllt ja nicht den ganzen Bildschirm aus (siehe Abbildung 3.1), sondern steht innerhalb einer Art Fernseher. Wir müssen also eine Spielfläche (*playground*) definieren. Nur innerhalb dieser Spielfläche sollen die Blasen erscheinen.



Abb. 3.2: Blase

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 66 werden

die entsprechenden Werte festgehalten. Der Abstand von Spielfeldrand und der Blasen untereinander wird in Zeile 65 entsprechend Requirement 2.4 mit 10 *px* definiert. Der Startradius – und damit der minimale Radius – wird wegen Requirement 2.3 in Zeile 64 mit 15 *px* festgelegt. Während des Spielens ist mir aufgefallen, dass kleinere Startradien einfach zu schlecht gesehen werden.

Quelltext 3.6: Bubbles (Requirement 2) – Ergänzungen in Game

```

64 radius = {"min": 15}                                     # Radius Startwert
65 distance = 10                                         # Rand-/Blasenabstand
66 playground = pygame.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 3.7: Bubbles (Requirement 2) – Timer

```

10 class Timer:
11     def __init__(self, duration: int, with_start: bool = True) -> None:
12         self.duration = duration
13         if with_start:
14             self._next = pygame.time.get_ticks()
15         else:
16             self._next = pygame.time.get_ticks() + self.duration
17
18     def is_next_stop_reached(self) -> bool:
19         if pygame.time.get_ticks() > self._next:
20             self._next = pygame.time.get_ticks() + self.duration
21             return True
22         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 3.8: Bubbles (Requirement 2) – Bubble

```

35 class Bubble(pygame.sprite.DirtySprite):
36     def __init__(self) -> None:
37         super().__init__()
38         self.radius = Game.radius["min"]
39         imagename = Game.get_image("blase1.png")
40         self.image: pygame.surface.Surface = pygame.image.load(imagename).convert_alpha()
41         self.image = pygame.transform.scale(self.image, (Game.radius["min"],
42                                         Game.radius["min"]))
43         self.rect: pygame.Rect = self.image.get_rect()
44         self.dirty = 1
45
46     def update(self, *args: Any, **kwargs: Any) -> None:
47         pass
48
49     def randompos(self) -> None:

```

```

49     bubbledistance = Game.distance + Game.radius["min"]
50     centerx = randint(Game.playground.left + bubbledistance, Game.playground.right -
51         bubbledistance)
51     centery = randint(Game.playground.top + bubbledistance, Game.playground.bottom -
52         bubbledistance)
52     self.rect.center = (centerx, centery)

```

Die Klasse `Game` muss nun entsprechend erweitert werden. In der Zeile 85 wird das `Background`-Objekt angelegt. Zeile 86 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 geblättert. Mit Hilfe von `pygame.sprite.LayeredDirty.clear()` wird in Zeile 88 das entsprechende Image festgelegt. Die Methode `Background.draw()` kann daher gelöscht werden.

`clear()`

Quelltext 3.9: Bubbles (Requirement 2) – Konstruktor von `Game`

```

80     def __init__(self) -> None:
81         pygame.init()
82         self._screen = pygame.display.set_mode(Game.window.size)
83         pygame.display.set_caption(Game.caption)
84         self._clock = pygame.time.Clock()
85         self._background = Background()                                #
86         self._timer_bubble = Timer(500, False)                         # Timer 500ms
87         self._all_sprites = pygame.sprite.LayeredDirty()              # Alle Blasen
88         self._all_sprites.clear(self._screen, self._background.image) #
89         self._all_sprites.set_timing_threshold(1000.0/Game.fps)
90         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 3.10: Bubbles (Requirement 2) – `draw()` und `update()` von `Game`

```

100    def draw(self) -> None:
101        rects = self._all_sprites.draw(self._screen)
102        pygame.display.update(rects)  # type: ignore
103
104    def update(self) -> None:
105        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 `Endlos-schleife` 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 113) und nach der Kollisionsprüfung wieder auf seinen Ursprungswert reduziert (Zeile 115).

Sie sehen hier ein Beispiel dafür, dass der Methode `pygame.sprite.spritecollide()`

`sprite-
collide()`

eine Methodenreferenz mitgegeben wird – hier `pygame.sprite.collide_circle()` – und somit nicht die übliche Rechtecksprüfung vorgenommen wird.

Quelltext 3.11: Bubbles (Requirement 2) – `spawn_bubble()` von Game

```

107     def spawn_bubble(self) -> None:
108         if self._timer_bubble.is_next_stop_reached():
109             b = Bubble()
110             tries = 100
111             while tries > 0:
112                 b.randompos()
113                 b.radius += Game.distance
114                 collided = pygame.sprite.spritecollide(b, self._all_sprites, False,
115                                               pygame.sprite.collide_circle)
116                 b.radius -= Game.distance
117                 if collided:
118                     tries -= 1
119                 else:
120                     self._all_sprites.add(b)
121                     break

```

Das Ergebnis können Sie in Abbildung 3.3 sehen. Gleichmäßig sind die Bubbles auf der Spielfläche verteilt und der geforderte Abstand zum Rand und zwischen den Blasen ist dabei eingehalten.



Abbildung 3.3: Bubbles: Die Blasen haben beim Start einen Mindestabstand

3.1.3 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 3.12: Bubbles (Requirement 3) – Ergänzung von `Settings`

```
67     max_bubbles = playground.height * playground.width // 10000 # Erfahrungswert
```

Diese Obergrenze aus Zeile 67 wird in Zeile 110 abgefragt. Nur wenn die maximale Anzahl noch nicht erreicht wurde, wird eine neue Blase erzeugt.

Quelltext 3.13: Bubbles (Requirement 3) – Ergänzung von `Game` in `spawn_bubbles()`

```
108     def spawn_bubble(self) -> None:
109         if self._timer_bubble.is_next_stop_reached():
110             if len(self._all_sprites) <= Game.max_bubbles: # Platz?
111                 b = Bubble()
112                 tries = 100
113                 while tries > 0:
114                     b.randompos()
115                     b.radius += Game.distance
116                     collided = pygame.sprite.spritecollide(b, self._all_sprites, False,
117                                                 pygame.sprite.collide_circle)
118                     b.radius -= Game.distance
119                     if collided:
120                         tries -= 1
121                     else:
122                         self._all_sprites.add(b)
123                         break
```

Der Rest des Programmes bleibt unverändert.

3.1.4 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 42). scale()

Die Methode `get()` liefert mir dann zu einem Radius das passend skalierte und schon fertige Image. Vorab wird in den Zeilen 45 und 46 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 3.14: Bubbles (Requirement 4.1) – `BubbleContainer`

```
35 class BubbleContainer:
```

```

36     def __init__(self) -> None:
37         imagename = Game.get_image("blase1.png")
38         image: pygame.surface.Surface = pygame.image.load(imagename).convert_alpha()
39         self._images = {
40             i: pygame.transform.scale(image, (i * 2, i * 2))
41             for i in range(Game.radius["min"], Game.radius["max"] + 1)
42         }
43
44     def get(self, radius: int) -> pygame.surface.Surface:
45         radius = max(Game.radius["min"], radius)           # Untere Grenze
46         radius = min(Game.radius["max"], radius)           # Obere Grenze
47         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 89 passend zu Requirement 4.2 um die Angabe eines maximalen Radius erweitert.

Quelltext 3.15: Bubbles (Requirement 4.2) – Erweiterung von `Game`

```

89     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 55 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 68). Der neue Radius wird in Zeile 65 bestimmt. In der gleichen Zeile wird dieser Wert mit dem maximalen Radius aus `Game` verglichen und das Minimum der beiden ausgewählt. Diese Logik verhindert, dass der Radius zu groß wird.

Was hat es aber mit den Zeilen 67 und 70 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 3.16: Bubbles (Requirement 4) – Ergänzung von `Bubble`

```

50 class Bubble(pygame.sprite.DirtySprite):
51     def __init__(self, bubble_container: BubbleContainer) -> None:
52         super().__init__()
53         self._bubble_container = bubble_container           # Verweis auf Container
54         self.radius = Game.radius["min"]
55         self.image = self._bubble_container.get(self.radius) # Zugriff auf Bubbles
56         self.rect: pygame.rect.Rect = self.image.get_rect()
57         self.dirty = 1
58         self.fradius = float(self.radius)
59         self.speed = 100
60
61     def update(self, *args: Any, **kwargs: Any) -> None:
62         if "action" in kwargs.keys():

```

```

63     if kwargs["action"] == "grow":
64         self.fradius += self.speed * Game.deltatime
65         self.fradius = min(self.fradius, Game.radius["max"]) # Neuer Radius
66         self.radius = round(self.fradius)
67         center = self.rect.center
68         self.image = self._bubble_container.get(self.radius) # Neues Image
69         self.rect = self.image.get_rect()
70         self.rect.center = center # Neuer MP = Alter MP
71         self.dirty = 1
72
73     def randompos(self) -> None:
74         bubbledistance = Game.distance + Game.radius["min"]
75         centerx = randint(Game.playground.left + bubbledistance, Game.playground.right -
76             bubbledistance)
76         centery = randint(Game.playground.top + bubbledistance, Game.playground.bottom -
77             bubbledistance)
77         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 132).

Quelltext 3.17: Bubbles (Requirement 4) – Ergänzung von `update()` in `Game`

```

131     def update(self) -> None:
132         self._all_sprites.update(action="grow") # Bubbles aktualisieren
133         self.spawn_bubble()

```

Der BubbleContainer wird angelegt

Quelltext 3.18: Bubbles (Requirement 4) – Ergänzung im Konstruktor von `Game`

```

110     self._clock = pygame.time.Clock()
111     self._bubble_container = BubbleContainer()
112     self._background = Background()

```

und in der Methode `spawn_bubble()` wird der Aufruf des Konstruktors von `Bubble` um den `BubbleContainer` ergänzt.

Quelltext 3.19: Bubbles (Requirement 4) – Ergänzung von `spawn_bubble()` in `Game`

```

138     b = Bubble(self._bubble_container) # Verweis auf Bubbles

```

Die Blasen wachsen nun um ihren Mittelpunkt herum nach außen. Das Ergebnis könnte dann wie in Abbildung 3.4 auf der nächsten Seite aussehen.

3.1.5 Requirement 5: Mauscursor

Requirement 5 Mauscursor

Befindet sich die Maus innerhalb einer Blase, soll sich das Aussehen ändern.



Abbildung 3.4: Bubbles: Die Blasen sind gewachsen/verwachsen

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 3.5 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.

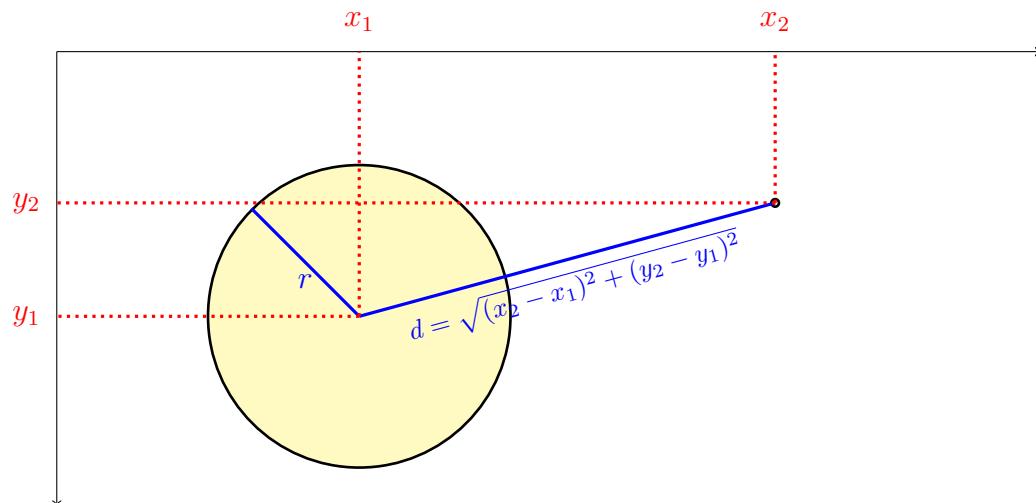


Abbildung 3.5: Kollisionserkennung: Punkt innerhalb des Kreises ([Satz des Pythagoras](#))?

Quelltext 3.20: Bubbles (Requirement 5) – `collidepoint()` in Game

```
153     def collidepoint(self, point: Tuple[int, int], sprite: pygame.sprite.Sprite) -> bool:
154         if hasattr(sprite, 'radius'):
155             deltax = point[0] - sprite.rect.centerx # type: ignore
```

```

156         deltay = point[1] - sprite.rect.centery # type: ignore
157         return (sqrt(deltax * deltax + deltay * deltay) <= sprite.radius) # type: ignore
158     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 164 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 3.21: Bubbles (Requirement 5) – `set_mousecursor()` in Game

```

160     def set_mousecursor(self) -> None:
161         is_over = False
162         pos = pygame.mouse.get_pos()
163         for b in self._all_sprites:
164             if self.collidepoint(pos, b): # Innerhalb?
165                 is_over = True
166                 break
167         if is_over:
168             pygame.mouse.set_cursor(pygame.SYSTEM_CURSOR_HAND)
169         else:
170             pygame.mouse.set_cursor(pygame.SYSTEM_CURSOR_CROSSHAIR)

```

Die Methode `update` in `Game` muss noch um den Aufruf der Überprüfung erweitert werden.

Quelltext 3.22: Bubbles (Requirement 5) – `update()` in Game

```

132     def update(self) -> None:
133         self._all_sprites.update(action="grow")
134         self.spawn_bubble()
135         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.

3.1.6 Requirement 6: Blasen zerplatzen

Requirement 6 Blasen zerplatzen

Bei einem Linksklick innerhalb einer Blase, soll die Blase zerplatzen.

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 127) und die aktuelle Mausposition an die – neu erstellte Methode – `sting()` übergeben (Zeile 129).

MOUSE
BUTTON
DOWN
get_pos()

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 3.23: Bubbles (Requirement 6) – `watch_for_event()` in Game

```

120  def watch_for_events(self) -> None:
121      for event in pygame.event.get():
122          if event.type == QUIT:
123              self._running = False
124          elif event.type == KEYDOWN:
125              if event.key == K_ESCAPE:
126                  self._running = False
127          elif event.type == MOUSEBUTTONDOWN:           # Mausklick?
128              if event.button == 1:                      # left
129                  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 177). Wenn *Ja*, dann wird das entsprechende Objekt aus der `Spritegroup` mit `kill()` entfernt.

kill()

Quelltext 3.24: Bubbles (Requirement 6) – `sting()` in Game

```

175  def sting(self, mousepos: Tuple[int, int]) -> None:
176      for bubble in self._all_sprites:
177          if self.collidepoint(mousepos, bubble):           # Innerhalb?
178              bubble.kill()

```

3.1.7 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 `Game` oder eine globale Variable. Ich bevorzuge Variante 1 (Quelltext 3.25).

Quelltext 3.25: Bubbles (Requirement 7.1) – Erweiterung von Game

```
117     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 84 wird einfach der Radius der Blase auf den Punktestand addiert.

Quelltext 3.26: Bubbles (Requirement 7.2) – `stung()` in `Bubble`

```
82     def stung(self):                                # Increment points
83         self.kill()
84         Game.points += self.radius
```

Der Aufruf von `stung()` erfolgt durch ein angepasstes `update()`.

Quelltext 3.27: Bubbles (Requirement 7.2) – `update()` in `Bubble`

```
62     def update(self, *args: Any, **kwargs: Any) -> None:
63         if "action" in kwargs.keys():
64             if kwargs["action"] == "grow":
65                 self.fradius += self.speed * Game.deltatime
66                 self.fradius = min(self.fradius, Game.radius["max"])
67                 self.radius = round(self.fradius)
68                 center = self.rect.center
69                 self.image = self._bubble_container.get(self.radius)
70                 self.rect = self.image.get_rect()
71                 self.rect.center = center
72                 self.dirty = 1
73             elif kwargs["action"] == "sting":
74                 self.stung()
```

Die Methoden `sting()` und `update()` in `Game` müssen dazu passend verändert werden (Zeile 203 und Zeile 161).

Quelltext 3.28: Bubbles (Requirement 7.2) – `sting()` in `Game`

```
200    def sting(self, mousepos: Tuple[int, int]) -> None:
201        for bubble in self._all_sprites:
202            if self.collidepoint(mousepos, bubble):
203                bubble.update(action="sting")           # Nach Bubble verschoben
```

Quelltext 3.29: Bubbles (Requirement 7.2) – `update()` in `Game`

```
160    def update(self) -> None:
161        self._all_sprites.update(action="grow")          #
162        self.spawn_bubble()
163        self.set_mousecursor()
```

Verbleibt Requirement 7.3. Ähnlich wie für die Spielfläche möchte ich die Maße für den unteren Teil als Ausgabebox in `Game` festlegen.

Rect

Quelltext 3.30: Bubbles (Requirement 7.3) – Erweiterung von `Settings`

```
116    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 3.31: Bubbles (Requirement 7.3) – Points

```

87  class Points(pygame.sprite.DirtySprite):
88      def __init__(self) -> None:
89          super().__init__()
90          self._font = pygame.font.Font(pygame.font.get_default_font(), 18)
91          self.oldpoints = -1
92          self.dirty = 1
93
94      def update(self, *args: Any, **kwargs: Any) -> None:
95          if self.oldpoints != Game.points:
96              self.image = self._font.render(f"Points: {Game.points}", True, "red")
97              self.rect = self.image.get_rect()
98              self.rect.left = Game.box.left
99              self.rect.top = Game.box.top
100             self.dirty = 1

```

Verbleiben einige Erweiterungen in `Game`. Im Konstruktor wird das `Points`-Objekt in das `LayeredDirty`-Objekt gesteckt.

Quelltext 3.32: Bubbles (Requirement 7.3) – Erweiterung des Konstruktors von `Game`

```

141      self._all_sprites.set_timing_threshold(1000.0/Game.fps)
142      self._all_sprites.add(Points())                                # Points
143      self._running = True

```



Abbildung 3.6: Bubbles: Ausgabe Punktestand

In Abbildung 3.6 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.

3.1.8 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 3.33: Bubbles (Requirement 8) – Bubble.speed

60 self.speed = 10

Die Grundstruktur unseres Spiels ermöglicht es, diese Anforderung recht leicht durch eine Erweiterung von `update()` in `Game` zu realisieren.

Quelltext 3.34: Bubbles (Requirement 8) – Erweiterung von update() in Game

```
163     def update(self) -> None:
164         if self.check_bubblecollision():
165             self._running = False
166         else:
167             self._all_sprites.update(action="grow")
168             self.spawn_bubble()
169         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 164) 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 212 durchwandern die Gruppe der Blasen zweimal und vermeiden dabei zwei Dinge:

- 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 versetzten Index erreicht.

In Zeile 217 wird Requirement 8.1 überprüft. Dabei wird die auf der Kreisform basierende Kollisionsprüfung mit `collide_circle()` verwendet. In der Zeile 219 und Zeile 221 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.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.

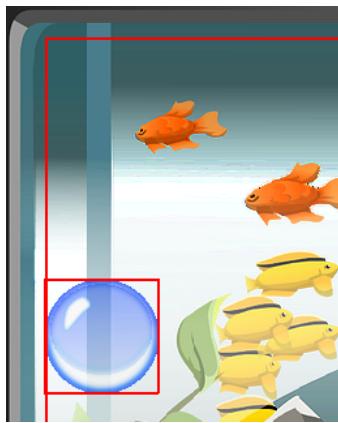


Abbildung 3.7: Bubbles – Kollision mit dem Rand

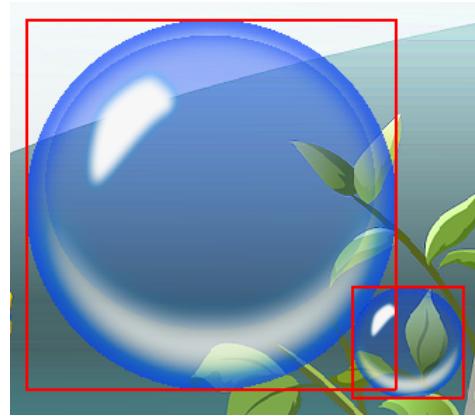


Abbildung 3.8: Bubbles – Kollision der Blasen

Quelltext 3.35: Bubbles (Requirement 8) – `check_bubblecollision()` in Game

```

211 def check_bubblecollision(self) -> bool:
212     for index1 in range(0, len(self._all_sprites) - 1):      # Blasen prüfen
213         for index2 in range(index1 + 1, len(self._all_sprites)):
214             bubble1 = self._all_sprites.sprites()[index1]
215             bubble2 = self._all_sprites.sprites()[index2]
216             if type(bubble1).__name__ == "Bubble" and type(bubble2).__name__ == "Bubble":
217                 if pygame.sprite.collide_circle(bubble1, bubble2): # Blasen kollidieren
218                     return True
219                 if not Game.playground.contains(bubble1):      # Blase1 berührt Rand
220                     return True
221                 if not Game.playground.contains(bubble2):      # Blase2 berührt Rand
222                     return True
223     return False

```

In Abbildung 3.7 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 3.8 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 3.36: Bubbles (Requirement 8) – Hilfslinien in Game

```

156 def draw(self) -> None:
157     rects = self._all_sprites.draw(self._screen)
158     # pygame.draw.rect(self._screen, "red", Game.playground, 2)
159     # for b in self._all_bubbles:
160     #     pygame.draw.rect(self._screen, "red", b.rect, 2) # type: ignore
161     pygame.display.update(rects) # type: ignore

```

3.1.9 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 60 wird dieser Parameter in ein Attribut geparkt.

Quelltext 3.37: Bubbles (Requirement 9) – Bubble

```

51 class Bubble(pygame.sprite.DirtySprite):
52     def __init__(self, bubble_container: BubbleContainer, speed: int) -> None:
53         super().__init__()
54         self._bubble_container = bubble_container
55         self.radius = Game.radius["min"]
56         self.image = self._bubble_container.get(self.radius)
57         self.rect: pygame.rect.Rect = self.image.get_rect()
58         self.dirty = 1
59         self.fradius = float(self.radius)
60         self.speed = speed # Wachstumsgeschwindigkeit

```

Das sind schon alle Anpassungen in `Bubble`, der Rest passiert in `Game`. In Zeile 139 wird ein Timer erstellt, der mir alle 1000 *ms* ein Signal geben wird. Darunter wird die

Timer

anfängliche Wachstumsgeschwindigkeit der Blasen auf 10 *px/s* gestellt.

Quelltext 3.38: Bubbles (Requirement 9) – Konstruktor von `Game`

```

138     self._timer_bubble = Timer(500, False)
139     self._timer_bubble_speed = Timer(1000, False) #
140     self._bubble_speed = 10

```

In `spawn_bubble()` wird der Timer abgefragt und ggf. die Blasenwachstumsgeschwindigkeit¹ erhöht (Zeile 174). 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 3.39: Bubbles (Requirement 9) – `Game.spawn_bubble()`

```

173     def spawn_bubble(self) -> None:
174         if self._timer_bubble_speed.is_next_stop_reached(): #
175             if self._bubble_speed < 100:
176                 self._bubble_speed += 5

```

Wenn Sie jetzt das Spiel ausprobieren (`bubbles09.py`), werden Sie einen leichten Start und eine moderate Steigerung der Spielschwierigkeit bemerken.

¹ Deutsch ist schon eine coole Sprache ;-)

3.1.10 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 3.9).



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 126 ist dazu ein Dictionary angelegt worden. Unter einem Schlüssel kann ich dort nun beliebige `BubbleContainer`-Objekte ablegen.

Quelltext 3.40: Bubbles (Requirement 10) – Erweiterung von Game

126 `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 3.41: Bubbles (Requirement 10) – Änderung Konstruktor von BubbleContainer

```
36  class BubbleContainer:
37      def __init__(self, filename: str) -> None:                      # Jetzt mit Dateinamen
38          imagename = Game.get_image(filename)
39          image: pygame.surface.Surface = pygame.image.load(imagename).convert_alpha()
40          self._images = {
41              i: pygame.transform.scale(image, (i * 2, i * 2))
42              for i in range(Game.radius["min"], Game.radius["max"] + 1)
43          }
```

Der Konstruktor von `Game` füllt nun das statische Dictionary `bubble_container` auf (Zeile 145 und Zeile 146).

Quelltext 3.42: 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 54) 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 69 in update() erwähnt.

Quelltext 3.43: Bubbles (Requirement 10) – Konstruktor von Bubble und update()

```

51 class Bubble(pygame.sprite.DirtySprite):
52     def __init__(self, speed: int) -> None:
53         super().__init__()
54         self.mode = "blue"                                # Farbmodus
55         self.radius = Game.radius["min"]
56         self.image = Game.bubble_container[self.mode].get(self.radius)
57         self.rect: pygame.rect.Rect = self.image.get_rect()
58         self.dirty = 1
59         self.fradius = float(self.radius)
60         self.speed = speed
61
62     def update(self, *args: Any, **kwargs: Any) -> None:
63         if "action" in kwargs.keys():
64             if kwargs["action"] == "grow":
65                 self.fradius += self.speed * Game.deltatime
66                 self.fradius = min(self.fradius, Game.radius["max"])
67                 self.radius = round(self.fradius)
68                 center = self.rect.center
69                 self.image = Game.bubble_container[self.mode].get(self.radius)  #
70                 self.rect = self.image.get_rect()
71                 self.rect.center = center
72                 self.dirty = 1
73             elif kwargs["action"] == "sting":
74                 self.stung()
75             elif "mode" in kwargs.keys():
76                 self.set_mode(kwargs["mode"])

```

Ändert sich der Modus, muss die andere Farbe nachgeladen werden. Dies erfüllt die Methode `set_mode` in `Bubble`.

Quelltext 3.44: Bubbles (Requirement 10) – `set_mode()` in `Bubble`

```

78     def set_mode(self, mode: str) -> None:
79         if mode != self.mode:
80             self.dirty = 1
81             self.mode = mode
82             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 3.10 auf der nächsten Seite können Sie sehen, wie die beiden kollidierenden Blasen rot erscheinen. Wie das geschieht, können Sie beispielhaft in Zeile 233 sehen.

Quelltext 3.45: Bubbles (Requirement 10) – `check_bubblecollision()` in `Game`

```

226     def check_bubblecollision(self) -> bool:
227         for index1 in range(0, len(self._all_sprites) - 1):
228             for index2 in range(index1 + 1, len(self._all_sprites)):
229                 bubble1 = self._all_sprites.sprites()[index1]
230                 bubble2 = self._all_sprites.sprites()[index2]
231                 if type(bubble1).__name__ == "Bubble" and type(bubble2).__name__ == "Bubble":

```

```

232             if pygame.sprite.collide_circle(bubble1, bubble2):
233                 bubble1.update(mode="red")                      # rot
234                 bubble2.update(mode="red")
235             return True
236         if not Game.playground.contains(bubble1):
237             bubble1.update(mode="red")
238             return True
239         if not Game.playground.contains(bubble2):
240             bubble2.update(mode="red")
241             return True

```

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 255).
wait()

Quelltext 3.46: Bubbles (Requirement 10) – Wartezeit in `run()`

```

244     def run(self) -> None:
245         time_previous = time()
246         self._running = True
247         while self._running:
248             self.watch_for_events()
249             self.update()
250             self.draw()
251             self._clock.tick(Game.fps)
252             time_current = time()
253             Game.deltatime = time_current - time_previous
254             time_previous = time_current
255             pygame.time.wait(2000)                      # Kurz warten
256             pygame.quit()

```

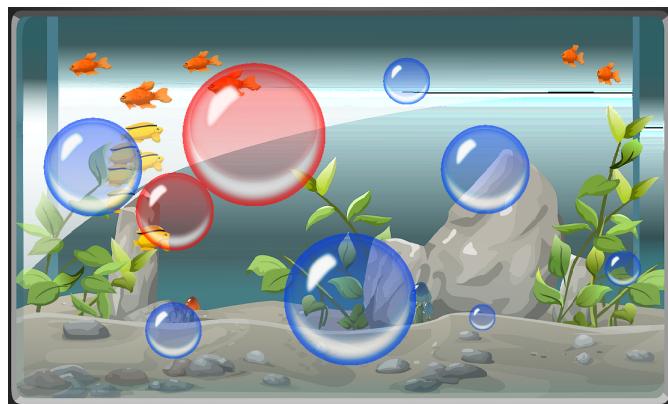


Abbildung 3.10: Bubbles: Kollision anzeigen

3.1.11 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 3.11 können Sie sehen, wie der Pausenbildschirm aussehen sollte. Zunächst werden die Pygame-Konstanten KEYUP, MOUSEBUTTONDOWNUP und K_p importiert (Zeile 8).

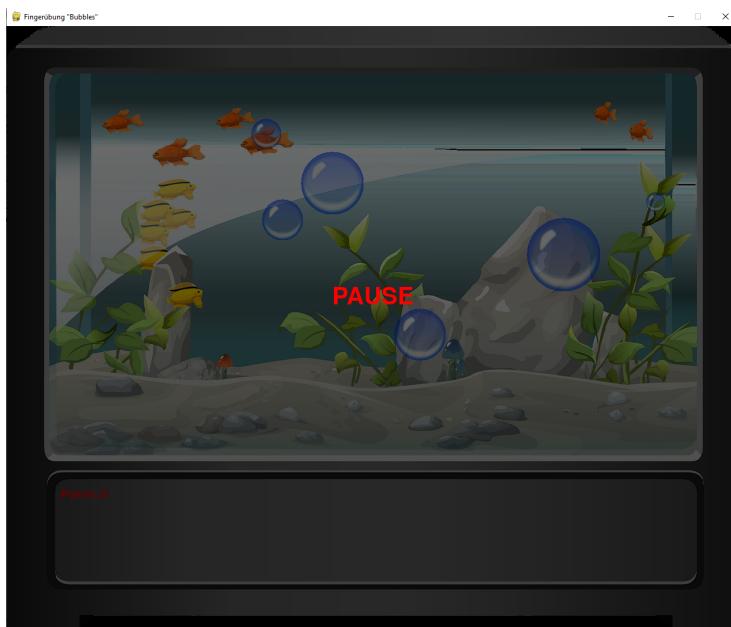


Abbildung 3.11: Bubbles: Pausenbildschirm

Quelltext 3.47: Bubbles (Requirement 11) – Zusätzliche Konstanten

```
8  from pygame.constants import (K_ESCAPE, KEYDOWN, #  
9   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 3.48: Bubbles (Requirement 11) – Konstruktor in Game

```
165  self._pausing = False #
```

In `watch_for_events()` wird nun abgefragt, ob die P-Taste (Zeile 176) oder die rechte Maustaste (Zeile 179) gedrückt wurde. In beiden Fällen wird die neue Methode `setpause()` aufgerufen.

Quelltext 3.49: Bubbles (Requirement 11) – `watch_for_events()` in Game

```

168  def watch_for_events(self) -> None:
169      for event in pygame.event.get():
170          if event.type == QUIT:
171              self._running = False
172          elif event.type == KEYDOWN:
173              if event.key == K_ESCAPE:
174                  self._running = False
175              elif event.type == KEYUP:
176                  if event.key == K_p:                      #
177                      self.setpause()
178                  elif event.type == MOUSEBUTTONUP:
179                      if event.button == 3:                  # right
180                          self.setpause()
181                  elif event.type == MOUSEBUTTONDOWN:
182                      if event.button == 1:                  # left
183                          self.sting(pygame.mouse.get_pos())

```

Für die Darstellung der Pause, habe ich die – vielleicht etwas überflüssige – Klasse `Pause` implementiert.

Quelltext 3.50: Bubbles (Requirement 11) – Pause

```

51  class Pause(pygame.sprite.DirtySprite):
52      def __init__(self) -> None:
53          super().__init__()
54          imagename = Game.get_image("pause.png")
55          self.image: pygame.surface.Surface = pygame.image.load(imagename).convert_alpha()
56          self.rect = self.image.get_rect()
57          self.dirty = 1

```

Im Konstruktor von `Game` wird ein Objekt der Klasse `Pause` angelegt, damit es in `draw()` verwendet werden kann.

Quelltext 3.51: Bubbles (Requirement 11) – Konstruktor in Game

```

166      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 3.52: Bubbles (Requirement 11) – `setpause()` in Game

```

201  def setpause(self):
202      if not self._pausing:
203          self._all_sprites.add(self._pause)
204      else:
205          self._pause.kill()
206          self._pausing = not self._pausing

```

Mehr ist nicht nötig, da der Rest von den üblichen `update()`- und `draw()`-Mechanismen erledigt wird.

3.1.12 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 3.53: Bubbles (Requirement 12) – Erweiterung des Imports

```
8 from pygame.constants import (K_ESCAPE, KEYDOWN, #
9                                KEYUP, MOUSEBUTTONDOWN, MOUSEBUTTONUP, QUIT, K_j,
10                               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 53).

Quelltext 3.54: Bubbles (Requirement 12) – von Pause zu Message

```
52 class Message(pygame.sprite.DirtySprite):
53     def __init__(self, filename: str) -> None:      #
54         super().__init__()
55         imagename = Game.get_image(filename)
56         self.image: pygame.surface.Surface = pygame.image.load(imagename).convert_alpha()
57         self.rect = self.image.get_rect()
58         self.dirty = 1
```

In `Game` werden im Konstruktor ab Zeile 163 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.

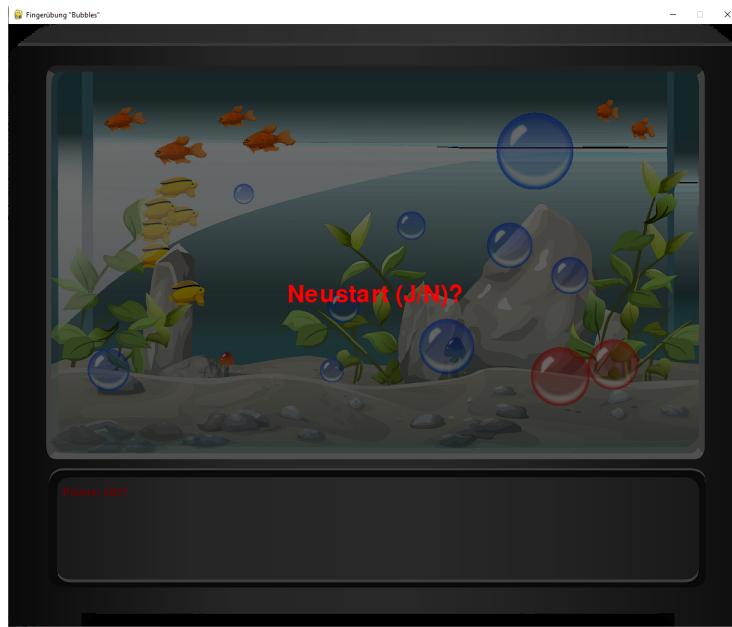


Abbildung 3.12: Bubbles: Neustartbildschirm

Quelltext 3.55: Bubbles (Requirement 12) – Umbau Konstruktor von Game

```

150  def __init__(self) -> None:
151      pygame.init()
152      self._screen = pygame.display.set_mode(Game.window.size)
153      pygame.display.set_caption(Game.caption)
154      self._clock = pygame.time.Clock()
155      Game.bubble_container["blue"] = BubbleContainer("blase1.png")
156      Game.bubble_container["red"] = BubbleContainer("blase2.png")
157      self._background = Background()
158      self._all_sprites = pygame.sprite.LayeredDirty()
159      self._all_sprites.clear(self._screen, self._background.image)
160      self._all_sprites.set_timing_threshold(1000.0/Game.fps)
161      self._running = True
162      self._pausing = False
163      self._pause = Message("pause.png") # 
164      self._restart = Message("neustart.png")
165      self.restart()

```

Der Punkttestand 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 3.56: Bubbles (Requirement 12) – `restart()` in Game

```

208  def restart(self):
209      Game.points = 0
210      self._all_sprites.empty()
211      self._all_sprites.add(Points())
212      self._bubble_speed = 10
213      self._timer_bubble = Timer(500, False)
214      self._timer_bubble_speed = Timer(10000, False)
215      self._do_start = False

```

216

```
        self._restarting = False
```

Aufgerufen wird die Methode in `update()`, wenn das entsprechende Flag `do_start` gesetzt 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 3.57: Bubbles (Requirement 12) – `update()` in `Game`

```
195  def update(self) -> None:
196      if self._do_start:                                     # Neustart?
197          self.restart()
198      if not self._pausing and self._running:
199          if self.check_bubblecollision():
200              if not self._restarting:
201                  self._all_sprites.add(self._restart)
202                  self._restarting = True
203              else:
204                  self._all_sprites.update(action="grow")
205                  self.spawn_bubble()
206                  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 177), 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 179).

Quelltext 3.58: Bubbles (Requirement 12) – Erweiterung von `watch_for_events()`

```
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.key == K_j:          #
178                      self._do_start = True
179                  elif event.key == K_n:          #
180                      self._running = False
181              elif event.type == MOUSEBUTTONUP:
182                  if event.button == 3:
183                      self.setpause()
184              elif event.type == MOUSEBUTTONDOWN:
185                  if event.button == 1:          # left
186                      self.sting(pygame.mouse.get_pos())
```

Da wir nun am Spielende eine halbtransparente Vordergrundausgabe haben, brauchen wir keine zweisekündige Pause, um die kollidierenden Blasen anzusehen (siehe Abbildung 3.12 auf der vorherigen Seite). Daher kann Zeile 297 auskommentiert werden.

Quelltext 3.59: Bubbles (Requirement 12) – run() in Game

```

286     def run(self) -> None:
287         time_previous = time()
288         self._running = True
289         while self._running:
290             self.watch_for_events()
291             self.update()
292             self.draw()
293             self._clock.tick(Game.fps)
294             time_current = time()
295             Game.deltatime = time_current - time_previous
296             time_previous = time_current
297             # pygame.time.wait(2000)           # Neustart?
298             pygame.quit()

```

3.1.13 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 136).

Quelltext 3.60: Bubbles (Requirement 13) – SoundContainer

```

120     class Game:
121         window = pygame.Rect(0, 0, 1220, 1002)
122         fps = 60
123         deltatime = 1.0/fps
124         path: Dict[str, str] = {}
125         path["file"] = os.path.dirname(os.path.abspath(__file__))
126         path["image"] = os.path.join(path["file"], "images")
127         path["sound"] = os.path.join(path["file"], "sounds")
128         caption = 'Fingerübung "Bubbles"'
129         radius = {"min": 15, "max": 240}
130         distance = 50
131         playground = pygame.Rect(90, 90, 1055, 615)
132         max_bubbles = playground.height * playground.width // 10000
133         box = pygame.Rect(90, 770, 1055, 1300)
134         points = 0
135         bubble_container: Dict[str, BubbleContainer] = {}
136         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 157ff.).

Sound

Quelltext 3.61: Bubbles (Requirement 13) – SoundContainer auffüllen

```

150  def __init__(self) -> None:
151      pygame.init()
152      self._screen = pygame.display.set_mode(Game.window.size)
153      pygame.display.set_caption(Game.caption)
154      self._clock = pygame.time.Clock()
155      Game.bubble_container["blue"] = BubbleContainer("blase1.png")
156      Game.bubble_container["red"] = BubbleContainer("blase2.png")
157      Game.sound_container["bubble"] = pygame.mixer.Sound(Game.get_sound("plopp1.mp3")) #
158      Game.sound_container["burst"] = pygame.mixer.Sound(Game.get_sound("burst.mp3"))
159      Game.sound_container["clash"] = pygame.mixer.Sound(Game.get_sound("glas.wav"))
160      self._background = Background()
161      self._all_sprites = pygame.sprite.LayeredDirty()
162      self._all_sprites.clear(self._screen, self._background.image)
163      self._all_sprites.set_timing_threshold(1000.0/Game.fps)
164      self._running = True
165      self._pausing = False
166      self._pause = Message("pause.png")
167      self._restart = Message("neustart.png")

```

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 247.

Quelltext 3.62: Bubbles (Requirement 13.1) – `spawn_bubble()`

```

230  def spawn_bubble(self) -> None:
231      if self._timer_bubble_speed.is_next_stop_reached():
232          if self._bubble_speed < 100:
233              self._bubble_speed += 5
234      if self._timer_bubble.is_next_stop_reached():
235          if len(self._all_sprites) <= Game.max_bubbles:
236              b = Bubble(self._bubble_speed)
237              tries = 100
238              while tries > 0:
239                  b.randompos()
240                  b.radius += Game.distance
241                  collided = pygame.sprite.spritecollide(b, self._all_sprites, False,
242                                              pygame.sprite.collide_circle)
243                  b.radius -= Game.distance
244                  if collided:
245                      tries -= 1
246                  else:
247                      self._all_sprites.add(b)
248                      Game.sound_container["bubble"].play() #
249                      break

```

Dann, wenn in `sting()` eine Blase zerplatzt (Zeile 272):

Quelltext 3.63: Bubbles (Requirement 13.2) – `sting()`

```

269  def sting(self, mousepos: Tuple[int, int]) -> None:
270      for bubble in self._all_sprites:
271          if self.collidepoint(mousepos, bubble):
272              Game.sound_container["burst"].play() #
273              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 Neu-

start anzeigt. Wenn *Ja*, darf der Sound nicht noch einmal abgespielt werden; ansonsten würde permanent der Berühren-Sound abgespielt.

Quelltext 3.64: Bubbles (Requirement [13.3](#)) – `update()`

```
199     def update(self) -> None:
200         if self._do_start:
201             self.restart()
202         if not self._pausing and self._running:
203             if self.check_bubblecollision():
204                 if not self._restarting:
205                     Game.sound_container["clash"].play()          #
206                     self._all_sprites.add(self._restart)
207                     self._restarting = True
208             else:
209                 self._all_sprites.update(action="grow")
210                 self.spawn_bubble()
211                 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 ohne und mit <code>Vector2</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
3.1	Bubbles: Hintergrundbild (aquarium.png)	129
3.2	Blase	129
3.3	Bubbles: Die Blasen haben beim Start einen Mindestabstand	132
3.4	Bubbles: Die Blasen sind gewachsen/verwachsen	136
3.5	Kollisionserkennung: Punkt innerhalb des Kreises?	136
3.6	Bubbles: Ausgabe Punktestand	140
3.7	Bubbles – Kollision mit dem Rand	142
3.8	Bubbles – Kollision der Blasen	142
3.9	Blase 2	144
3.10	Bubbles: Kollision anzeigen	146
3.11	Bubbles: Pausenbildschirm	147
3.12	Bubbles: Neustartbildschirm	150

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 Spielemente 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. [162](#)

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:`. [131](#)

fade Kommt vom englischen *to fade* für *verbllassen*. 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: $6,75 = 2^2 + 2^1 + 2^0 - 1 + 2^{-2}$. 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](#), [162](#)

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 Ganzzommazahl 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](#)

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 fortgefahrene 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. [158](#), [162](#)

Joint Photographic Experts Group Verlustbehaftete komprimierte Bildinformationen. [157](#), [162](#)

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](#), [160](#)

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](#), [162](#)

mp3 Abkürzung von *ISO MPEG Audio Layer 3*. Eine im wesentlichen vom deutschen Elektrotechningenieur 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. [162](#)

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](#), [162](#)

Polygon Ein geschlossener [Linienzug](#). Wird meist durch eine Folge von Punkten definiert, wobei der letzte Punkt mit dem ersten verbunden wird. [12](#), [159](#)

Portable Network Graphics Verlustfrei komprimierte Bildinformationen. [157](#), [162](#)

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 wentslichen Features sind die Typüberwachung und Auto vervollständigung. [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 $2\pi rad$. [102](#)

Red Green Blue Additive Farbkodierung. [162](#)

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 Hypotenusequadrat: $c^2 = \sqrt{a^2 + b^2}$. Der Satz ist nach dem Mathematiker *Pythagoras von Samos* (um 570 v.Chr. bis um 510 v.Chr.) benannt. [136](#)

Semantik Bedeutung einer Angabe. Wird meist in Abgrenzung zu [Syntax](#) einer Angabe verwendet. [12](#), [161](#)

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 Überabeparameter. [52](#)

Simple Direct Media Layer Eine plattformunabhängige API für die Programmierung von Grafiken, Sounds und Eingabegräte. [5, 162](#)

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, 162](#)

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. [162](#)

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, 161](#)

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. [162](#)

Syntax Form oder Grammatik einer Angabe. Wird meist in Abgrenzung zur **Semantik** einer Angabe verwendet. [160](#)

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*. [148](#)

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 mitgepflegt 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. [162](#)

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. [162](#)

Windows Bitmap Format Bildinformationen im Windows Bitmap-Format. [157](#), [162](#)

Akronyme

bmp Windows Bitmap Format. [157](#), [162](#), *Glossar: Windows Bitmap Format*

fps frames per second. [8](#), [162](#), *Glossar: frames per second*

IDE Integrated Development Environment. [158](#), [162](#), *Glossar: Integrated Development Environment*

jpeg Joint Photographic Experts Group. [157](#), [162](#), *Glossar: Joint Photographic Experts Group*

ms Millisekunden. [87](#), [162](#), *Glossar: Millisekunden*

OO Objektorientiert. [62](#), [162](#), *Glossar: Objektorientiert*

png Portable Network Graphics. [157](#), [162](#), *Glossar: Portable Network Graphics*

pt DTP-Punkt. [62](#), [162](#), *Glossar: DTP-Punkt*

px Pixel. [6](#), [162](#), *Glossar: Pixel*

RGB Red Green Blue. [7](#), [157](#), [162](#), *Glossar: Red Green Blue*

SDL Simple Direct Media Layer. [5](#), [162](#), *Glossar: Simple Direct Media Layer*

SRP Single Responsibility Principle. [50](#), [162](#), *Glossar: Single Responsibility Principle*

SSD Solid-State-Drive. [7](#), [162](#), *Glossar: Solid-State-Drive*

Stereo Stereofonie. [108](#), [162](#), *Glossar: Stereofonie*

ttf True Type Font. [66](#), [162](#), *Glossar: True Type Font*

USB Universal Serial Bus. [7](#), [162](#), *Glossar: Universal Serial Bus*

Index

äquidistant, 23
__main__, 51
SDL_VIDEO_WINDOW_POS, 5, 9
round, 40
screen, 6
time
 time(), 40

Alpha-Kanal, 11, 23
Animation, 90
assoziatives Array, 157

Bitmap, 21
 ausgeben, 21
 bewegen, 29
 laden, 21

Blasen erscheinen, 129
Blasen zerplatzen, 137
Blasenanzahl, 132
Blasenwachstum, 133
Bubbles, 126

Dictionary, 71
Dirty Sprite, 115
Doublebuffer, 6
deltatime, 33, 35, 41, 85

Farbnamen, 19, 124
Farbnamen, 12, 116
Flag, 6, 149
Font, 62
Frame, 115
Framerate, 123
fade, 105
float, 39

Geschwindigkeit, 31

Grafikprimitive, 11
gravity, 15

Hashtable, 157
Hauptprogrammschleife, 6
Hintergrundmusik, 105

int, 39

Kanal, 109
Kollision, 75
Kollision anzeigen, 144
Kollisionserkennung
 Kreis, 75
 Pixel, 76
 Rechteck, 75

Maske, 76
Maus, 98
Mauscursor, 135
Mausrad, 99
main loop, 6
mp3, 105

Neustart, 149

ogg, 105

Pause, 147
Punktestand, 138
Pythagoras, Satz von, 136

Rechteck, 129
Rendern, 62
Richtung, 31
Richtungswechsel, 32

Schwerkraft, 15
Sound, 152

Soundausgaben, 104
Soundeffekte, 105
Spielende, 141
Sprite, 46
Standardfunktionalität, 126
self.image, 46
self.mask, 78
self.radius, 78, 130
self.rect, 46, 78, 129

Tastatur, 54
Timer, 88, 130, 143
Transparenz, 23

Zeitanpassungen, 143
Zeitsteuerung, 83
Zuordnungstabelle, 157

Index für den Namensraum pygame

Color, [11](#), [19](#)
KEYDOWN, [56](#), [57](#)
KEYUP, [56](#), [57](#), [147](#)
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](#), [138](#)
MOUSEBUTTONUP, [99](#), [102](#), [147](#)
QUIT, [7](#), [9](#)
Rect, [12](#), [29](#), [45](#), [139](#)
 bottomright, [29](#)
 bottom, [29](#)
 centerx, [29](#)
 centery, [29](#)
 center, [29](#)
 collidepoint(), [100](#), [102](#)
 contains(), [141](#)
 height, [29](#), [129](#)
 left, [29](#), [129](#)
 move(), [33](#), [45](#), [52](#)
 move_ip(), [52](#)
 right, [29](#)
 topleft, [29](#)
 top, [29](#), [129](#)
 width, [29](#), [129](#)
SYSTEM_CURSOR_CROSSHAIR, [137](#)
SYSTEM_CURSOR_HAND, [137](#)
Surface, [6](#)
 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](#)
clock
 tick(), [38](#), [40](#)
display
 flip(), [6](#), [9](#), [120](#)
 get_surface(), [6](#), [9](#), [128](#)
 get_window_size(), [16](#), [20](#)
 set_caption(), [6](#), [9](#), [128](#)
 set_mode(), [6](#), [9](#), [128](#)
 update(), [7](#), [9](#), [120](#), [124](#)
draw
 circle(), [12](#), [19](#)
 line(), [12](#), [19](#)
 lines(), [12](#), [19](#)
 polygon(), [12](#), [19](#)
 rect(), [12](#), [20](#)
event
 Event
 unicode, [73](#), [74](#)
 button, [99](#), [102](#)
 get(), [7](#), [9](#)
 key, [56](#)
 mod, [56](#)
 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, 128
key, 56
mask
 from_surface(), 78, 82
math
 Vector2, 39, 45, 46
 Vector3, 39, 45
mixer
 Channel, 109, 113
 play(), 109, 113
 set_volume(), 110, 113
 Sound, 105, 109, 114, 152
 get_volume(), 105, 114
 play(), 106, 109, 114, 153
 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, 137, 138
 get_pressed(), 13, 20
 set_visible(), 100, 102
quit(), 7, 9
rect
 Rect, 20
sprite
 DirtySprite, 118, 124
 dirty, 115, 118, 124
 GroupSingle, 50, 52
 sprite, 50, 52
 Group, 50, 52, 119
 sprites(), 141
LayeredDirty, 119, 124, 140
 clear(), 120, 124, 131
 draw(), 120, 124
 set_timing_threshold(), 121, 124
 set_timing_threshold(), 124
Sprite, 46, 53
 kill(), 84, 89, 96, 138
 update(), 52
collide_circle(), 81, 82, 132
collide_mask(), 81, 82
collide_rect(), 49, 53, 81, 82
spritecollide(), 50, 53, 81, 82, 131
DirtySprite, 126
LayeredGroup, 126
time
 Clock, 8, 10, 128
 tick(), 8, 10, 85
 tick_busy_loop(), 8, 10
 get_ticks(), 34, 45, 87, 89
 wait(), 146
transform
 rotate(), 102, 102
 scale(), 22, 28, 133