

Einführung in die 2D-Spieleprogrammierung mit Pygame

Ralf Adams (**TBS1**, Bochum)

Version 0.12 vom 10. Oktober 2025

Inhaltsverzeichnis

1	Ziele	4
2	Grundlagen	5
2.1	Das erste Beispiel	5
2.2	Grafikprimitive	11
2.2.1	Grundlagen	11
2.2.2	Beispiel: Fontaine	13
2.3	Bitmaps laden und ausgeben	21
2.4	Bitmaps bewegen	28
2.4.1	Grundlagen	28
2.4.2	Geschwindigkeiten normalisieren (<i>deltatime</i>)	32
2.5	Sprite-Klasse	45
2.6	Tastatur	53
2.7	Textausgabe mit Fonts	60
2.7.1	Default-Font	60
2.7.2	Fontliste	63
2.8	Textausgabe mit Bitmaps	67
2.9	Kollisionserkennung	74
2.10	Zeitsteuerung	82
2.11	Animation	89
2.11.1	Die laufende Katze	89
2.11.2	Der explodierende Felsen	94
2.12	Maus	97
2.13	Soundausgaben	103
2.13.1	Hintergrundmusik und Soundereignisse	103
2.13.2	Stereo	107
2.14	Dirty Sprites	114
2.14.1	Einfaches Beispiel	114
2.14.2	Performancemessungen	120
2.14.3	Fazit	123
2.15	Events	126
2.15.1	Welche Infos stecken in einem Event?	126
2.15.2	Wie kann ich selbst ein Event erzeugen?	127
2.15.3	Wie kann ich periodisch Ereignisse erzeugen lassen?	132

3 Beispielprojekte	134
3.1 Pong	134
3.1.1 Requirement 1: Standards	134
3.1.2 Requirement 2: Die Schläger	136
3.1.3 Requirement 3: Der Ball	139
3.1.4 Requirement 4: Punkte	141
3.1.5 Requirement 5: Tennisschlag	143
3.1.6 Requirement 6: Computerspieler	144
3.1.7 Requirement 7: Sound	146
3.1.8 Requirement 8: Pause und Hilfebildschirm	148
3.2 Bubbles	151
3.2.1 Requirement 1: Standards	151
3.2.2 Requirement 2: Blasen erscheinen	154
3.2.3 Requirement 3: Blasenanzahl	157
3.2.4 Requirement 4: Blasenwachstum	158
3.2.5 Requirement 5: Mauscursor	160
3.2.6 Requirement 6: Blasen zerplatzen	162
3.2.7 Requirement 7: Punktestand	163
3.2.8 Requirement 8: Spielende	165
3.2.9 Requirement 9: Zeitanpassungen	167
3.2.10 Requirement 10: Kollision anzeigen	168
3.2.11 Requirement 11: Pause	171
3.2.12 Requirement 12: Neustart	173
3.2.13 Requirement 13: Sound	176
3.3 Moonlander	178
3.3.1 Requirement 1: Standards	178
3.3.2 Requirement 2: Mondoberfläche	181
3.3.3 Requirement 3: Erde	186
3.3.4 Requirement 4: Sterne	187
3.3.5 Requirement 5: Landefähre	189
3.3.6 Requirement 6: Gravitation und Aufsetzten	192
3.3.7 Requirement 7: Gegenschub	194
3.3.8 Requirement 8: Treibstoff	195
3.3.9 Requirement 9: Statusanzeige	196
3.3.10 Requirement 10: Spielende und Neustart	198
3.3.11 Requirement 11: Autopilot	203

1 Ziele

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

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

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

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

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

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

2 Grundlagen

2.1 Das erste Beispiel

Quelltext 2.1: Mein erstes *Spiel*, Version 1.0

```
1 import pygame # Pygame-Modul (auch bei Pygame-ce!)
2
3
4 def main():
5     pygame.init() # Subsystem starten
6     window = pygame.Window(size=(600, 400)) # Fenster erzeugen
7     window.title = "Mein_<erstes_Pygame-Programm" # Fenstertitel
8     window.position = (10, 50) # Fensterposition
9     screen = window.get_surface() # Das Bitmap des Fensters holen
10
11    running = True
12    while running: # Hauptprogrammschleife: start
13        for event in pygame.event.get(): # Ermitteln der Events
14            if event.type == pygame.QUIT: # Fenster X angeklickt?
15                running = False
16            screen.fill((0, 255, 0)) # Spielfläche einfärben
17            window.flip() # Doublebuffer austauschen
18
19    pygame.quit() # Subsystem beenden
20
21
22 if __name__ == "__main__":
23     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 1). Danach stehen uns **Konstanten**, **Funktionen** und **Klassen** des **Namensraums** zur Verfügung.

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 5 wird der ganze Pygame-Motor mit `init()` angeworfen. Man könnte



Abb. 2.1: Spielfläche

`init()`

also nur die Komponenten starten, die gerade gebraucht werden wie beispielsweise die Soundunterstützung mit `pygame.mixer.init()`.

Wir brauchen für unsere Spiele eine *Spieldfläche*/ein Fenster innerhalb dessen sich alles abspielt (*Vorsicht, Wortspiel!*). Die Klasse `pygame.Window` repräsentiert einen solchen Spieldfläche. Der Konstruktor bekommt in Zeile 6 einen Übergabeparameter – nämlich die Breite und die Höhe des Fensters als das 2-Tupel `size`. Unser Fenster ist also 600 *px* breit und 400 *px* (siehe [Pixel \(px\)](#)) hoch. Die Methode `get_surface()` in Zeile 9 liefert mir ein `pygame.Surface`-Objekt, was ungefähr sowas wie ein [Bitmap](#) ist.

In Zeile 9 speichere ich diese Rückgabe in die Variable `screen`.

Dem Fenster kann ich dann noch über das Attribut `Window.title` eine Titelüberschrift verpassen (siehe Zeile 7) und mit dem Attribut `Window.position` die Position des Fensters relativ zum Desktop (siehe Zeile 8).

Hinweis: Sie können auch mehrere Fenster für das Spiel erstellen (siehe <https://pygame/docs/ref/window.html>)

Das Spiel selbst – so wie auch alle zukünftigen Spiele – laufen innerhalb einer [Hauptprogrammschleife](#). Hier startet die Schleife in Zeile 12 und endet in Zeile 19. Innerhalb dieser Schleife werden zukünftig immer drei Dinge passieren:

1. Ereignisse auslesen und verarbeiten: Wie in Zeile 13f. werden Maus-, Tastatur- oder Spielereignisse festgestellt und an die Spielelemente weitergegeben. In unserem Fall wird lediglich das Anklicken des X im Fenster oben rechts registriert.
2. Zustand der Spielelemente aktualisieren: Basierend auf den oben festgestellten Ereignissen und den Zuständen der Spielelemente, 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 Spielelemente malen: Die Spielelemente haben eine neue Position oder ein neues Aussehen und müssen deshalb neu gemalt werden. In diesem Minimalbeispiel wird lediglich Zeile 16 der Hintergrund der Spieldfläche eingefärbt und anschließend in Zeile 17 der `Doublebuffer` mit `Window.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 ab Zeile 13 die Ereignisse durch und picke die für mich interessanten heraus.

Dabei überprüfe ich zuerst, was für ein Ereignistyp (`pygame.event.type`) mir da angeboten wird. Derzeit ist für mich nur der Typ `pygame.QUIT` wichtig. Dieser Typ wird ausgelöst, wenn das Betriebssystem eine *Beenden*-Nachricht an die Anwendung sendet.

Falls ich nun eine solche Nachricht empfange, setzte ich das `Flag` `running` auf `False`, so dass die Hauptprogrammschleife beendet wird.

Falls ich dieses Signal nicht empfange, läuft die Hauptprogrammschleife fröhlich weiter und füllt in Zeile 16 die gesamte Spielfläche mit `screen.fill()` mit einer Farbe – hier grün – ein. Bitte beachten Sie, dass ähnlich wie in Zeile 6 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.

Verbleibt noch Zeile 17: Dort wird die Funktion `pygame.quit()` aufgerufen. Diese Funktion ist quasi das Gegenteil von `pygame.init()` in Zeile 5. 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 oder dem ordnungsgemäßen Lichtausmachen und Türabschließen beim Verlassen der Wohnung.

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



Abbildung 2.2: Ressourcenverbrauch ohne Taktung

Wenn wir uns die Hauptprogrammschleife anschauen, sollte es allerdings nicht wirklich verwundern. Da wird ungebremst ein Bitmap auf den Bildschirm gemalt und das ohne Unterbrechung. Besser wäre es, bei jedem Schleifendurchlauf genügend Zeit zur Verfügung zu stellen, um die Ereignisse einzusammeln, die neuen Zustände zu berechnen und erst dann die Bildschirmausgabe zu generieren. Die Bildschirmausgabe selbst sollte auch nicht beliebig schnell und oft passieren, sondern in der Regel reichen 60 `frames per second (fps)`, um eine Bewegung als flüssig wahrzunehmen.

fps

Quelltext 2.2: Mein erstes *Spiel*, Version 1.1

```

1 import pygame
2
3
4 def main():
5     pygame.init()
6     window = pygame.Window(size=(600, 400),
7                             title="Mein_Erstes_Pygame-Programm",           # per Übergabeparameter
8                             position=(10, 50))
9     screen = window.get_surface()
10
11    clock = pygame.time.Clock()                      # Clock-Objekt
12
13    running = True
14    while running:

```

```

15     for event in pygame.event.get():
16         if event.type == pygame.QUIT:
17             running = False
18             screen.fill((0, 255, 0))
19             window.flip()
20             clock.tick(60)                      # Taktung auf 60 fps
21
22     pygame.quit()
23
24
25 if __name__ == "__main__":
26     main()

```

Clock

tick()

tick_busy_loop()

In Zeile 11 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 20. 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. Events abfragen.
 2. Spielobjekte aktualisieren.
 3. Bildschirminhalt ausgeben.
 4. Schleifendurchläufe takten.
- Beim Verlassen das Pygame-System stoppen.

Es wurden folgende Pygame-Elemente eingeführt:

- `import pygame:`
<https://pyga.me/docs/tutorials/en/import-init.html>
- `pygame.init():`
<https://pyga.me/docs/ref/pygame.html#pygame.init>
- `pygame.quit():`
<https://pyga.me/docs/ref/pygame.html#pygame.quit>
- `pygame.QUIT:`
<https://pyga.me/docs/ref/event.html#pygame.event.EventType.type>
- `pygame.event.get():`
<https://pyga.me/docs/ref/event.html#pygame.event.get>
- `pygame.event.type:`
<https://pyga.me/docs/ref/event.html#pygame.event.EventType.type>
- `pygame.time.Clock:`
<https://pyga.me/docs/ref/time.html#pygame.time.Clock>
- `pygame.time.Clock.tick():`
<https://pyga.me/docs/ref/time.html#pygame.time.Clock.tick>
- `pygame.time.Clock.tick_busy_loop():`
https://pyga.me/docs/ref/time.html#pygame.time.Clock.tick_busy_loop
- `pygame.Surface.fill():`
<https://pyga.me/docs/ref/surface.html#pygame.Surface.fill>
- `pygame.Window:`
<https://pyga.me/docs/ref/window.html>
- `pygame.Window.flip():`
<https://pyga.me/docs/ref/window.html#pygame.Window.flip>

- `pygame.Window.get_surface()`:
https://pyga.me/docs/ref/window.html#pygame.Window.get_surface
- `pygame.Window.title`:
<https://pyga.me/docs/ref/window.html#pygame.Window.title>
- `pygame.Window.position`:
<https://pyga.me/docs/ref/window.html#pygame.Window.position>

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 ganz nützlich sein. Ich will hier deshalb nur einige vorstellen.

Quelltext 2.3: Mein zweites *Spiel*, Version 1.0

```

1  import pygame
2  import pygame.gfxdraw  # Muss sein!
3
4
5  def main():
6      pygame.init()
7      window = pygame.Window( size=(530, 530),
8                               title = "Mein\u00d7zweites\u00d7Pygame-Programm",
9                               position = (10, 50))
10     screen = window.get_surface()
11
12     clock = pygame.time.Clock()
13
14     mygrey = pygame.Color(200, 200, 200)                      # Eigene Farben
15
16     myrectangle1 = pygame.rect.Rect(10, 10, 20, 30)          # Rechteck-Objekt
17     myrectangle2 = pygame.rect.Rect(60, 10, 20, 30)
18     points1 = ((120, 10), (160, 10), (140, 90))           # Punktliste
19     points2 = ((180, 10), (220, 10), (200, 90))
20
21     running = True
22     while running:
23         for event in pygame.event.get():
24             if event.type == pygame.QUIT:
25                 running = False
26             screen.fill(mycgrey)
27             pygame.draw.rect(screen, "red", myrectangle1)          # Gefülltes Rechteck
28             pygame.draw.rect(screen, "red", myrectangle2, 3, 5)      # Rechteck
29             pygame.draw.polygon(screen, "green", points1)          # Gefülltes Polygon
30             pygame.draw.polygon(screen, "green", points2, 1)        # Polygon
31             pygame.draw.line(screen, "red", (5, 230), (240, 230), 3)  # Linie
32             pygame.draw.circle(screen, "blue", (40, 150), 30)        # Gefüllter Kreis
33             pygame.draw.circle(screen, "blue", (110, 150), 30, 2)      # Kreis
34             pygame.draw.circle(screen, "blue", (180, 150), 30, 5, True) # Kreisbogenschnitt
35             for i in range(255):
36                 for j in range(255):
37                     screen.set_at((265+i, 10+j), (255, i, j))          # Punkte Variante 1
38                     screen.fill((i, j, 255), ((10+i, 265+j), (1, 1)))  # Variante 2
39                     pygame.gfxdraw.pixel(screen, 265+i, 265+j, (i, 255, j)) # Variante 3
40
41             window.flip()
42             clock.tick(60)
43
44     pygame.quit()

```

Der Grundaufbau ist der gleiche wie in Quelltext 2.2 auf Seite 7. 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.

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.

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

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

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

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

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

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

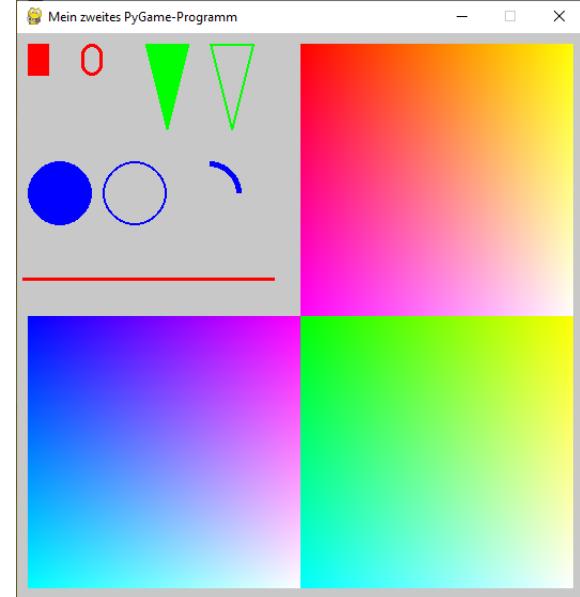


Abb. 2.4: Einige Grafikprimitive

Farbnamen

Rect

rect()

polygon()

line()
lines()

circle()

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

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

In Zeile 37 wird der Punkt durch das Setzen eines einzelnen Farbwertes an einer Position mit `pygame.Surface.set_at()` gezeichnet. Man könnte auch die schon oben verwendete Surface-Funktion `fill()` mit einer Ausdehnung von nur einem Pixel Breite und Höhe verwenden (Zeile 38). Ein Möglichkeit einen Pixel über eine Grafikbibliothek zu setzen, ist die experimentelle `gfxdraw`-Umgebung. In Zeile 39 wird mit `pygame.gfxdraw.pixel()` ein einzelnes Pixel gesetzt. Die `gfxdraw`-Umgebung wird nicht automatisch durch `import pygame` importiert (siehe Zeile 2).

`set_at()`

`pixel()`

2.2.2 Beispiel: Fontaine

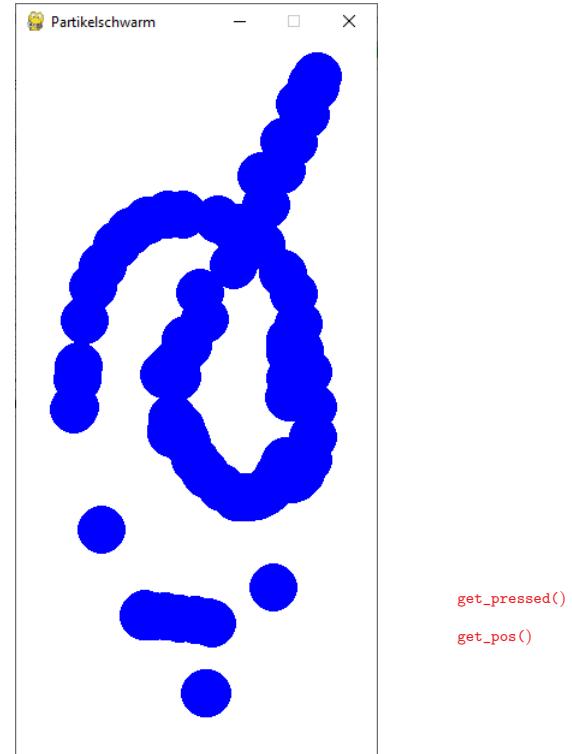
Man kann mit Grafikprimitiven dynamische Effekte einbauen, wie beispielsweise Partikelschwärme. Ich will hier mal ein super einfaches Beispiel für eine mausgesteuerte Fontaine aus Kreisen vorstellen.

Bauen wir zuerst ein kleines Programm, dass an der Mausposition einen Kreis zeichnet. Die Klasse `Circle` (siehe Zeile 6) 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 6 wird die Bildschirmgröße in einer Liste vorgehalten, da wir die Info noch an anderer Stelle als in Zeile 20 brauchen. Darunter wird in Zeile 26 eine Liste für die Aufnahme der Kreise definiert.

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

Das Ergebnis ist noch wenig berauschend (siehe Abb. 2.5: Sicher keine bildung 2.5) und erinnert mehr an ein Malprogramm wie PaintPartikelfontaine



`get_pressed()`
`get_pos()`

Quelltext 2.4: Partikelfontaine, Version 1.0

```

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

```

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

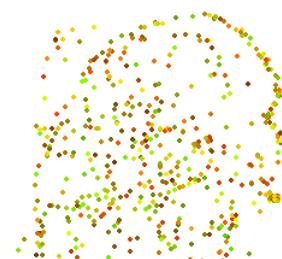


Abb. 2.6: Version 2

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  def __init__(self, pos) -> None:
8      self.posx = pos[0] + randint(-2, 2)
9      self.posy = pos[1] + randint(-2, 2)
10     self.radius = 2
11     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 14). 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 Schwerkraft `GRAVITY` verändert. Damit alle Partikel immer der gleichen Schwerkraft unterliegen, habe ich `GRAVITY` als statisches Attribut definiert (Zeile 7).

Schwerkraft

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

```

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

```

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

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

```

38     if pygame.mouse.get_pressed()[0]:
39         circles.append(Circle(pygame.mouse.get_pos()))
40
41     for p in circles:
42         p.update()

```

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()`

```

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

```

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 16), 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 22 wird überprüft, ob der rechte Rand des Partikels (Mittelpunkt plus Radius), links außerhalb des Bildschirms liegt. Falls ja, muss das Löschflag auf `True` gesetzt werden. Analog werden in Zeile 24 und Zeile 26 der rechte und der untere Rand des Bildschirm überprüft.

Dabei wird mit Hilfe des Attributs `pygame.Window.size` jeweils die Breite bzw. Höhe des Bildschirms ermittelt. Dieses Attribut liefert mir die Bildschirmgröße als 2-Tupel wieder. Der nullte Wert ist dabei die Breite und der erste die Höhe. Ein Test, ob der Partikel nach oben verschwunden ist, wird nicht benötigt, da er ja irgendwann wieder runterfällt und damit wieder sichtbar wird.



Abb. 2.7: Partikelfontaine,
Version 5: fast fertig

Quelltext 2.9: Partikelfontaine, Version 5.0, Klasse Circle

```

6  class Circle:
7      GRAVITY = 0.3
8
9      def __init__(self, pos) -> None:
10         self.posx = pos[0] + randint(-2, 2)
11         self.posy = pos[1] + randint(-2, 2)
12         self.radius = 2
13         self.color = [randint(100, 255), randint(50, 255), 0]
14         self.speedx = randint(-10, 10) / 10.01
15         self.speedy = randint(-100, 0) / 10.01
16         self.todelete = False           # Löschflag
17
18     def update(self, window: pygame.Window) -> None:
19         self.speedy += Circle.GRAVITY
20         self.posx += self.speedx
21         self.posy += self.speedy
22         if self.posx - self.radius < 0:           # links raus
23             self.todelete = True
24         elif self.posx + self.radius > window.size[0]: # rechts raus
25             self.todelete = True
26         elif self.posy - self.radius > window.size[1]: # unten raus
27             self.todelete = True
28
29     def draw(self, screen: pygame.surface.Surface) -> None:
30         pygame.draw.circle(screen, self.color, (self.posx, self.posy), self.radius)

```

Im Hauptprogramm muss ich nun einen passende Löschlogik implementieren. Vorab soll meine Fontaine aber noch mehr *Wumms* bekommen: In Zeile 48 wird nicht ein Partikel erzeugt, sondern immer gleich 5.

In Zeile 51 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 54). Wenn ja, wird dieser Partikel in die Liste `todelete` aufgenommen. Nach Beendigung der Update-Schleife werden die zu löschen Partikel ab Zeile 56 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

```

41     running = True
42     while running:
43         for event in pygame.event.get():
44             if event.type == pygame.QUIT:
45                 running = False
46
47             if pygame.mouse.get_pressed()[0]:
48                 for i in range(5):           # 5 Partikel gleichzeitig
49                     circles.append(Circle(pygame.mouse.get_pos()))
50
51             todelete = []           # Zwischenspeicher
52             for p in circles:
53                 p.update(window)
54                 if p.todelete:           # Zu löschen?
55                     todelete.append(p)
56             for p in todelete:           # Löschen
57                 circles.remove(p)

```

```

58         screen.fill("white")
59         for p in circles:
60             p.draw(screen)
61
62         window.flip()
63         clock.tick(60)

```

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

```

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

```

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

Quelltext 2.11: Partikelfontaine, Version 6.0

```

1  from random import randint
2
3  import pygame
4
5
6  class Circle:
7      GRAVITY = 0.3
8      RADIUS_INC = -0.1                                # Radiusinkrement
9
10     def __init__(self, pos) -> None:
11         self.posx = pos[0] + randint(-4, 4)            # Veränderte Streuung
12         self.posy = pos[1] + randint(-4, 4)
13         self.radius = 8
14         self.color = [randint(100, 255), randint(50, 255), 0]
15         self.speedx = randint(-15, 15) / 10.01         # Fontaine breiter
16         self.speedy = randint(-100, 0) / 10.01
17         self.todelete = False
18
19     def update(self, window: pygame.Window) -> None:
20         self.speedy -= Circle.GRAVITY
21         self.posx += self.speedx
22         self.posy += self.speedy
23         self.radius += Circle.RADIUS_INC
24         if self.posx - self.radius < 0:
25             self.todelete = True
26         elif self.posx + self.radius > window.size[0]:
27             self.todelete = True
28         elif self.posy - self.radius > window.size[1]:
29             self.todelete = True
30         elif self.radius <= 0.0:                         # Können raus
31             self.todelete = True
32
33     def draw(self, screen: pygame.surface.Surface) -> None:
34         pygame.draw.circle(screen, self.color, (self.posx, self.posy), self.radius)
35
36
37 def main():

```

```

38     size = (300, 600)
39     pygame.init()
40     window = pygame.Window( size=size, title = "Partikelschwarm", position = (10, 50))
41     screen = window.get_surface()
42     clock = pygame.time.Clock()
43     circles = []
44
45     running = True
46     while running:
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):
53                     circles.append(Circle(pygame.mouse.get_pos()))
54
55             todelete = []
56             for p in circles:
57                 p.update(window)
58                 if p.todelete:
59                     todelete.append(p)
60             for p in todelete:
61                 circles.remove(p)
62
63             screen.fill("white")
64             for p in circles:
65                 p.draw(screen)
66
67             window.flip()
68             clock.tick(60)
69
70     pygame.quit()
71
72
73 if __name__ == '__main__':
74     main()

```

Was war neu?

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

Es wurden folgende Pygame-Elemente eingeführt:

- Vordefinierte Farbnamen:
https://pyga.me/docs/ref/color_list.html
- `import pygame.gfxdraw`:
<https://pyga.me/docs/ref/gfxdraw.html>
- `pygame.Color`:
<https://pyga.me/docs/ref/color.html>
- `pygame.draw.circle()`:
<https://pyga.me/docs/ref/draw.html#pygame.draw.circle>

- `pygame.draw.line()`:
<https://pyga.me/docs/ref/draw.html#pygame.draw.line>
- `pygame.draw.lines()`:
<https://pyga.me/docs/ref/draw.html#pygame.draw.lines>
- `pygame.draw.polygon()`:
<https://pyga.me/docs/ref/draw.html#pygame.draw.polygon>
- `pygame.draw.rect()`:
<https://pyga.me/docs/ref/draw.html#pygame.draw.rect>
- `pygame.gfxdraw.pixel()`:
<https://pyga.me/docs/ref/gfxdraw.html#pygame.gfxdraw.pixel>
- `pygame.mouse.get_pos()`:
https://pyga.me/docs/ref/mouse.html#pygame.mouse.get_pos
- `pygame.mouse.get_pressed()`:
https://pyga.me/docs/ref/mouse.html#pygame.mouse.get_pressed
- `pygame.Rect`:
<https://pyga.me/docs/ref/rect.html>
- `pygame.Surface.set_at()`:
https://pyga.me/docs/ref/surface.html#pygame.Surface.set_at
- `pygame.Window.size`:
<https://pyga.me/docs/ref/window.html#pygame.Window.size>

2.3 Bitmaps laden und ausgeben

Quelltext 2.12: Bitmaps laden und ausgeben, Version 1.0

```

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

Wir wollen nun die Bitmaps ein wenig unseren Bedürfnissen anpassen. Zunächst empfiehlt das Handbuch, dass das Bitmap nach dem Laden in ein für Pygame leichter zu

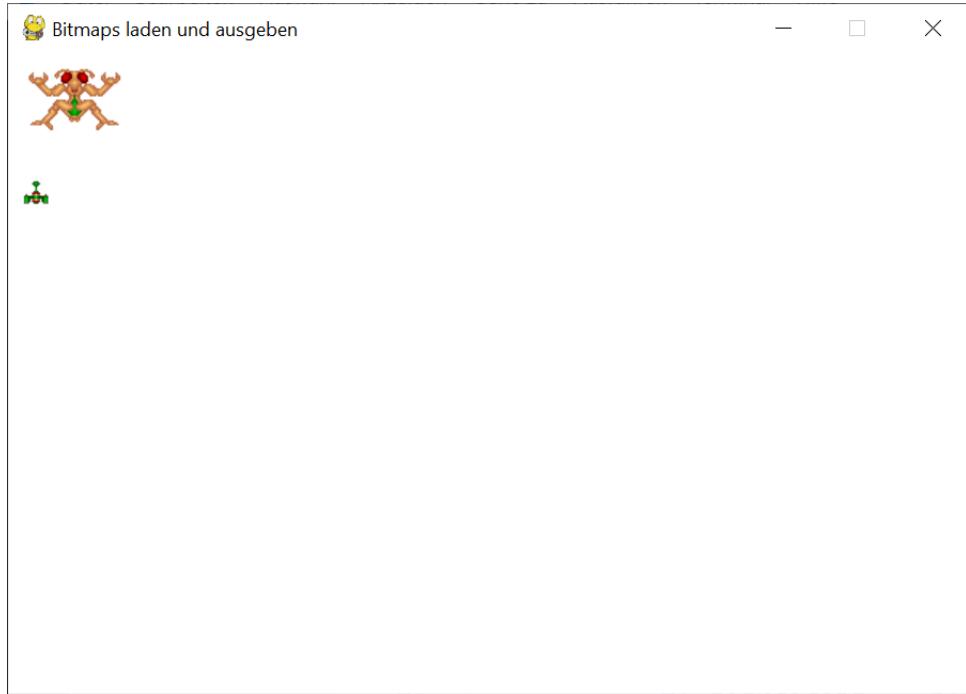


Abbildung 2.8: Bitmaps laden und ausgeben, Version 1.0

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

```

16  screen = window.get_surface()
17  clock = pygame.time.Clock()
18
19  defender_image = pygame.image.load("images/defender01.png").convert()  # konvertieren
20  defender_image = pygame.transform.scale(defender_image, (30, 30))      # skalieren

```

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 19 verwiesen.

Das Verändern der Größe erfolgt durch `pygame.transform.scale()`. In Zeile 20 wird das Image auf die angegeben (*width, height*) in der Maßeinheit Pixel skaliert. Das Ergebnis an Abbildung 2.9 entspricht nicht ganz meinen Erwartungen.

Die Größenverhältnisse gefallen mir zwar jetzt, aber warum erscheint plötzlich ein schwarzer Hintergrund? Die Ursache dafür ist, dass durch die Konvertierung mit `convert()`



Abb. 2.9: Größen OK

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

```
16 screen = window.get_surface()
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))
```

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 nicht 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.



Abb. 2.10: Transparenz OK

äquidistant

Quelltext 2.15: Bitmap: Positionen, Version 1.4

```
17 screen = window.get_surface()
18 clock = pygame.time.Clock()
19
20 defender_image = pygame.image.load("images/defender01.png").convert_alpha()
21 defender_image = pygame.transform.scale(defender_image, (30, 30))
22 defender_pos_left = (Settings.WINDOW_WIDTH - 30) // 2 # linke Koordinate
23 defender_pos_top = Settings.WINDOW_HEIGHT - 30 - 5 # obere Koordinate
```

```

24  defender_pos = (defender_pos_left, defender_pos_top)      # Mache ein 2-Tupel
25
26  alien_image = pygame.image.load("images/alienbig0101.png").convert_alpha()
27  alien_image = pygame.transform.scale(alien_image, (50, 45))
28  space_for_aliens = Settings.ALIENS_NOF * 50                # Verbrauchter Platz
29  space_available = Settings.WINDOW_WIDTH - space_for_aliens # Verfügbarer Platz
30  space_nof = Settings.ALIENS_NOF + 1                          # Anzahl Freiräume
31  space_between_aliens = space_available // space_nof        # Platz Freiräume
32
33  running = True
34  while running:
35      for event in pygame.event.get():
36          if event.type == pygame.QUIT:
37              running = False
38
39      screen.fill("white")
40      alien_top = 10                                         # Abstand von oben
41      for i in range(Settings.ALIENS_NOF):                   # Berechnung/Ausgabe

```

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.WINDOWS_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*:

```
20  defender_pos_top = Settings.WINDOWS_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 schwebe er im Raum:

```
20  defender_pos_top = Settings.WINDOWS_HEIGHT - 30 - 5
```

In Zeile 22 wird der Abstand des linken Rands des Bitmaps vom Spielfeldrand berechnet. Mit

```
19  defender_pos_left = Settings.WINDOWS_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 auf der nächsten Seite).

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*):

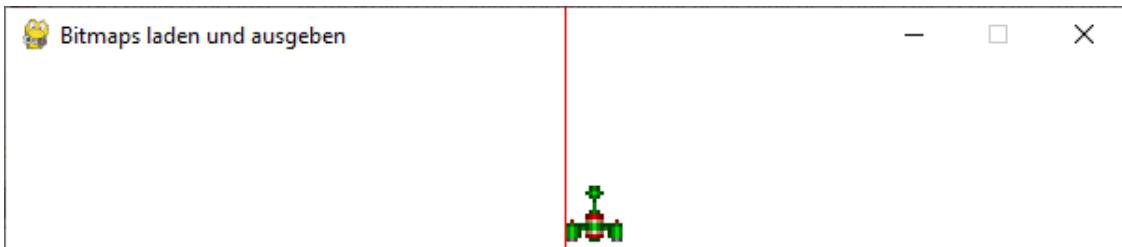


Abbildung 2.11: Bitmaps positionieren (Verteidiger)

```
19     defender_pos_left = Settings.WINDOWS_WIDTH // 2 - 30 // 2
```

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

```
19     defender_pos_left = (Settings.WINDOWS_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:

```
37     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:

```
38     for i in range(Settings.ALIENS_NOF):
39         alien_left = i * 50
40         alien_pos = (alien_left, alien_top)
41         screen.blit(alien_image, alien_pos)
```

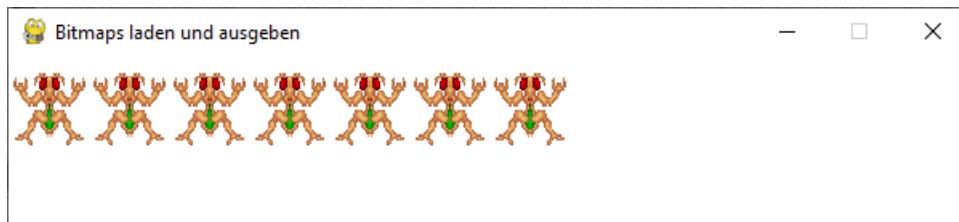


Abbildung 2.12: Bitmaps positionieren (Angreifer, Version 1)

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

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:

```
27     space_nof = 2
```

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

```
27     space_nof = Settings.ALIENS_NOF - 1 + 2
```

also:

```
27     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

```
25     space_for_aliens = Settings.ALIENS_NOF * 50
```

und diesen von der Bildschirmbreite abziehe.

```
26     space_available = Settings.WINDOWS_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:

```
28     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):

```
38     for i in range(Settings.ALIENS_NOF):
39         alien_left = space_between_aliens + i * 50
40         alien_pos = (alien_left, alien_top)
41         screen.blit(alien_image, alien_pos)
```

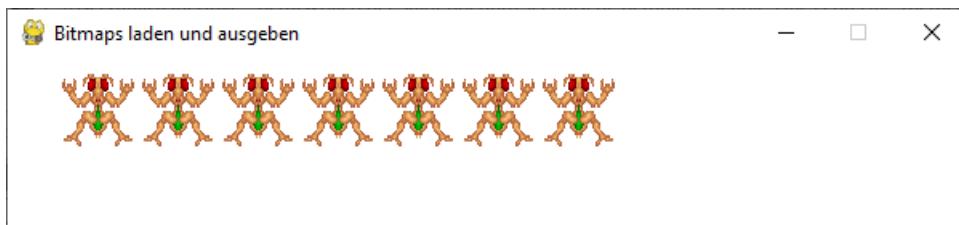


Abbildung 2.13: Bitmaps positionieren (Angreifer, Version 2)

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:

```

38     for i in range(Settings.ALIENS_NOF):
39         alien_left = space_between.aliens + i * (space_between.aliens + 50)
40         alien_pos = (alien_left, alien_top)
41         screen.blit(alien_image, alien_pos)

```

Und schon passt alles (siehe Abbildung 2.14).

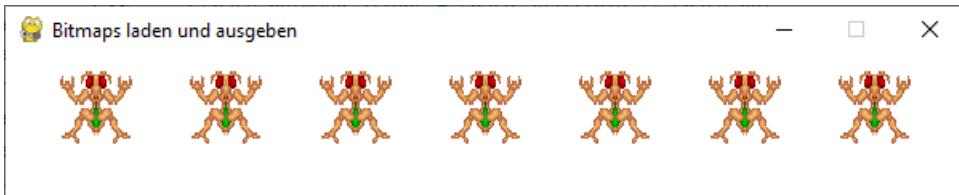


Abbildung 2.14: Bitmaps positionieren (Angreifer, Version 3)

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 oder mit anderen Worten: *Das Koordinatensystem hat seinen 0-Punkt linksoben und nicht linksunten*.

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

Es wurden folgende Pygame-Elemente eingeführt:

- `pygame.image` :
<https://pyga.me/docs/ref/image.html>
- `pygame.image.load()` :
<https://pyga.me/docs/ref/image.html#pygame.image.load>
- `pygame.Surface.blit()` :
<https://pyga.me/docs/ref/surface.html#pygame.Surface.blit>
- `pygame.Surface.convert()` :
<https://pyga.me/docs/ref/surface.html#pygame.Surface.convert>
- `pygame.Surface.convert_alpha()` :
https://pyga.me/docs/ref/surface.html#pygame.Surface.convert_alpha
- `pygame.Surface.set_colorkey()` :
https://pyga.me/docs/ref/surface.html#pygame.Surface.set_colorkey
- `pygame.transform.scale()` :
<https://pyga.me/docs/ref/transform.html#pygame.transform.scale>

2.4 Bitmaps bewegen

2.4.1 Grundlagen

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

Rect
FRect

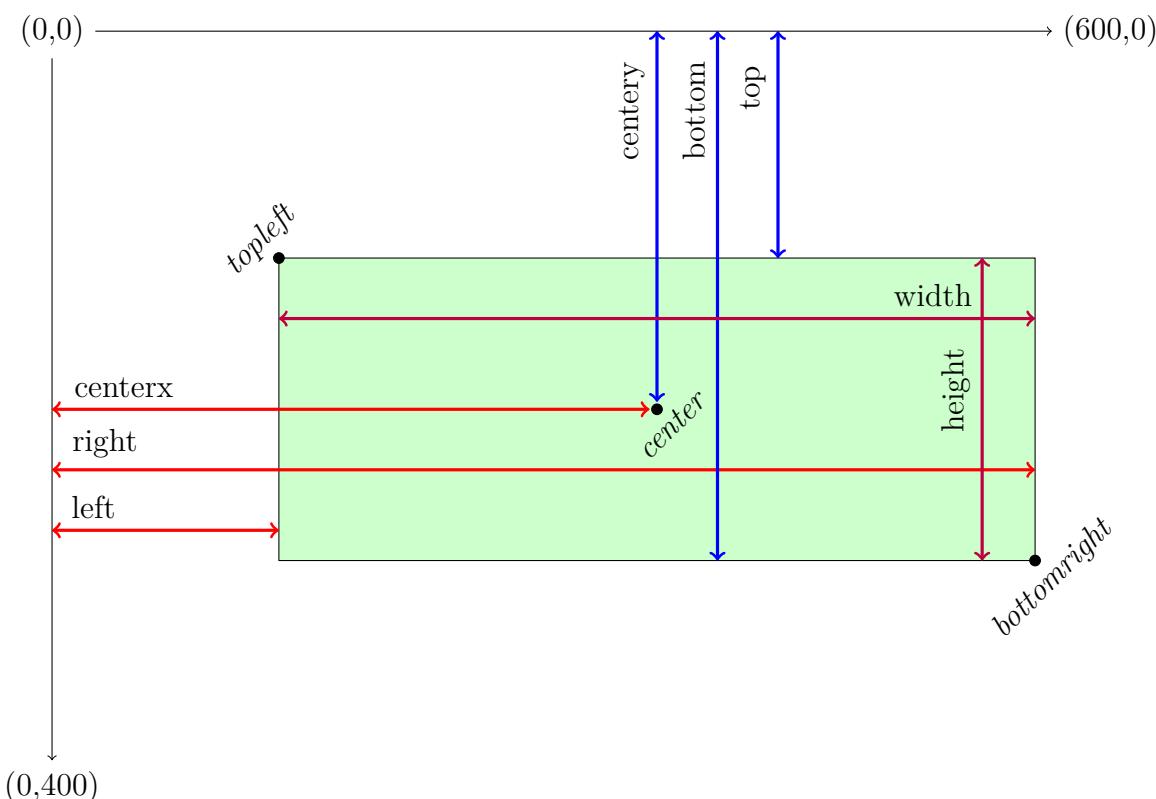


Abbildung 2.15: Elemente eines `Rect`-Objekts

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

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

Schauen wir uns dazu eine reduzierte Version des letzten Quelltextes an. In Quelltext 2.16 wird die `Rect`-Klasse schon verwendet. So werden beispielsweise in Zeile 5 die Fenstermaße in einem `Rect`-Objekt verwaltet. In den Zeilen Zeile 12, Zeile 21 und Zeile 22 können dadurch die Bildschirminformationen bequem und ohne eigene Berechnungen ausgelesen werden.

Quelltext 2.16: Bitmaps bewegen, Version 1.0

```

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

```

Für `Surface`-Objekte können wir sehr bequem mit `pygame.Surface.get_rect()` das `Rect`-Objekt erstellen lassen (Zeile 20). Die Positionierung kann nun leichter über die

`get_rect()`

Attribute erfolgen. Das Zentrum muss beispielsweise nicht mehr in die Berechnung einfließen, ich kann vielmehr das horizontale Zentrum direkt als halbe Fensterbreite festlegen (Zeile 21). 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 22). Und als Sahnehäubchen kann das `Rect`-Objekt auch noch als Parameter der `blit()`-Funktion übergeben werden (Zeile 35).

blit()

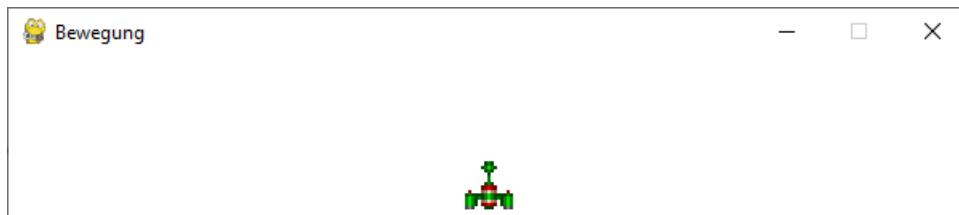


Abbildung 2.16: Bitmaps bewegen, Version 1.0

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`.

32 `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

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

Geschwindigkeit

Quelltext 2.17: Bitmaps bewegen, Version 1.2

15 `defender_image = pygame.image.load("images/defender01.png").convert_alpha()`
16 `defender_image = pygame.transform.scale(defender_image, (30, 30))`
17 `defender_rect = defender_image.get_rect()`
18 `defender_rect.centerx = Settings.WINDOW.centerx`
19 `defender_rect.bottom = Settings.WINDOW.height - 5`
20 `defender_speed = 2 # Geschwindigkeit`

```

21     defender_direction_h = +1  # Richtung
22
23     running = True
24     while running:
25         # Events
26         for event in pygame.event.get():
27             if event.type == pygame.QUIT:
28                 running = False
29
30     # Update
31     defender_rect.left += defender_direction_h * defender_speed  # Flexibler
32
33     # Draw
34     screen.fill("white")
35     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 20 wird die Geschwindigkeit nun durch die Variable `defender_speed` repräsentiert. So könnten wir im Laufe des Spiels die Geschwindigkeit dynamisch gestalten, z.B. bei einer Beschleunigung durch Raketentreibstoffausstoß.

Die Richtung wird in Zeile 21 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 31 zur Berechnung der neuen horizontalen Position genutzt werden.

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

Quelltext 2.18: Bitmaps bewegen, Version 1.3

```

30     # Update
31     defender_rect.left += defender_direction_h * defender_speed
32     if defender_rect.right >= Settings.WINDOW.right:  # Rechter Rand erreicht
33         defender_direction_h *= -1  # Richtungswechsel
34     elif defender_rect.left <= Settings.WINDOW.left:  # Linker Rand erreicht
35         defender_direction_h *= -1

```

Ich hoffe, dass Sie die Idee hinter dem Code erkennen. Nach Berechnung der neuen horizontalen Position, wird in Zeile 32 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.

Richtungswechsel

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

Ein Problem habe ich noch: In Zeile 31 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.

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

`move()`

Quelltext 2.19: Bitmaps bewegen, Version 1.4

```

30     # Update
31     newpos = defender_rect.move(defender_direction_h * defender_speed, 0)  # Testposition
32     if newpos.right >= Settings.WINDOW.right:
33         defender_direction_h *= -1
34         newpos.right = Settings.WINDOW.right # Ausrichten am rechten Rand
35     elif newpos.left <= Settings.WINDOW.left:
36         defender_direction_h *= -1
37         newpos.left = Settings.WINDOW.left # Ausrichten am linken Rand
38     defender_rect = newpos # Übernehme neue Position

```

Die neue Funktion taucht in Zeile 31 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. Ebenso wird der linke Rand des Bitmap am linken Bildschirmrand ausgerichtet und der rechte am rechten. Danach wird `newpos` zu unserem neuen Rechteck für den Verteidiger (Zeile 38).



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 7 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 25 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.

`get_ticks()`

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 36) – 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 28 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

```

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

```

```

36     defender_rect.top += defender_direction_v * defender_speed # Neue Höhe
37     if defender_rect.bottom >= Settings.WINDOW.bottom:
38         defender_direction_v *= -1
39     elif defender_rect.top <= 0:
40         defender_direction_v *= -1
41
42     # Draw
43     screen.fill("white")
44     screen.blit(defender_image, defender_rect)
45     window.flip()
46     clock.tick(Settings.FPS)
47     print(f"top={defender_rect.top}")
48
49     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

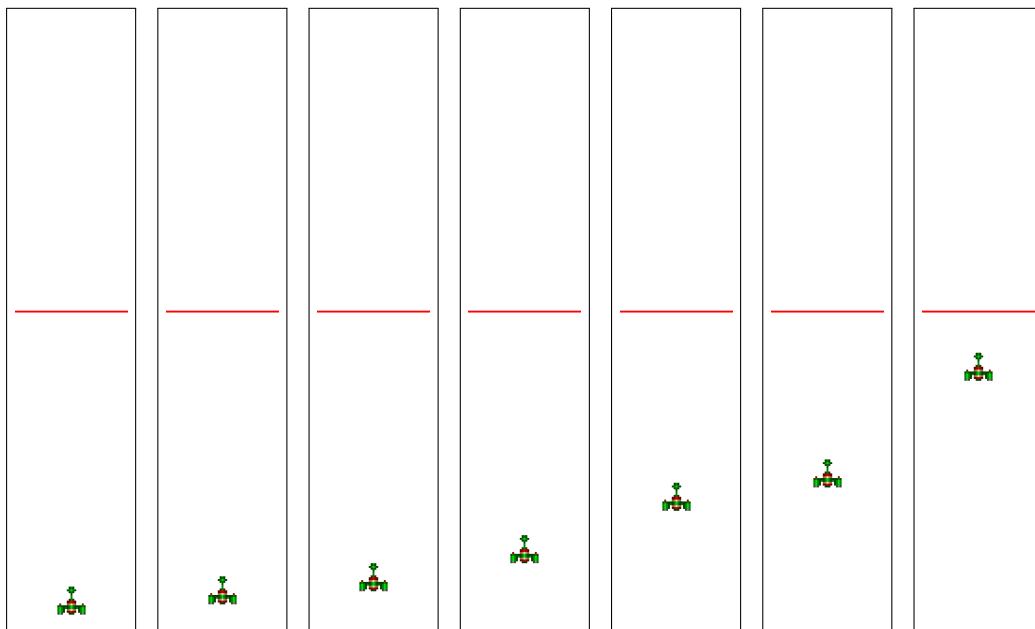


Abbildung 2.18: Nicht normalisierte Bewegung bei gleicher Geschwindigkeit aber unterschiedlichen Frameraten (von links nach rechts: 10, 30, 60, 120, 240, 300, 600)

In der [Tabelle 2.2](#) ist die Strecke, die die Geschwindigkeit bei unterschiedlichen Frameraten auf einer Strecke von 2 px zurücklegt, dargestellt. Die Geschwindigkeit ist hier als $\frac{\text{px}}{\text{s}}$ angegeben. Um die tatsächliche Geschwindigkeit zu erhalten, muss die Framerate $\frac{1}{\text{fps}}$ mit der Geschwindigkeit multipliziert werden. Dieser Korrekturwert wird *deltatime* (dt) 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
 Geschw. $(\frac{\text{px}}{\text{s}}) * \text{fps} (\frac{\text{f}}{\text{s}}) * dt (\frac{\text{s}}{\text{f}}) * \text{Zeit (s)} = \text{Strecke (px)}$

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 habe ich mal berechnet, welche Endposition (.top) wir nach einer halben Sekunde erwarten können. In der linken Hälfte wird die Wegstrecke 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 9 wird der Korrekturwert – wie oben besprochen – als Inverses von der Frame-rate definiert. Die Geschwindigkeit wird von 2 auf 600 in Zeile 23 angepasst und in Zeile 37 wird der Korrekturwert DELTATIME in Berechnung als Faktor eingebaut. Das war's; in Abbildung 2.22 auf Seite 42 können wir das *perfekte* Ergebnis auf einem meiner langsameren Rechner bewundern.

Quelltext 2.21: Normalisierte Bewegung mit 1/fps

```

5  class Settings:
6      WINDOW = pygame.Rect((0, 0), (120, 650))
7      FPS = 300 # 10 30 60 120 240 300 600
8      LIMIT = 500
9      DELTATIME = 1.0 / FPS # Korrekturwert
10
11
12  def main():
13      pygame.init()
14      window = pygame.Window(size=Settings.WINDOW.size, title="Bewegung", position=(10, 50))
15      screen = window.get_surface()
16      clock = pygame.time.Clock()
17
18      defender_image = pygame.image.load("images/defender01.png").convert_alpha()
19      defender_image = pygame.transform.scale(defender_image, (30, 30))
20      defender_rect = defender_image.get_rect()
21      defender_rect.centerx = Settings.WINDOW.centerx
22      defender_rect.bottom = Settings.WINDOW.bottom - 5
23      defender_speed = 600 # Nicht mehr px/f sondern px/s
24      defender_direction_v = -1
25
26      start_time = pygame.time.get_ticks()
27      running = True
28      while running:
29          if pygame.time.get_ticks() > start_time + Settings.LIMIT:

```

```

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

```

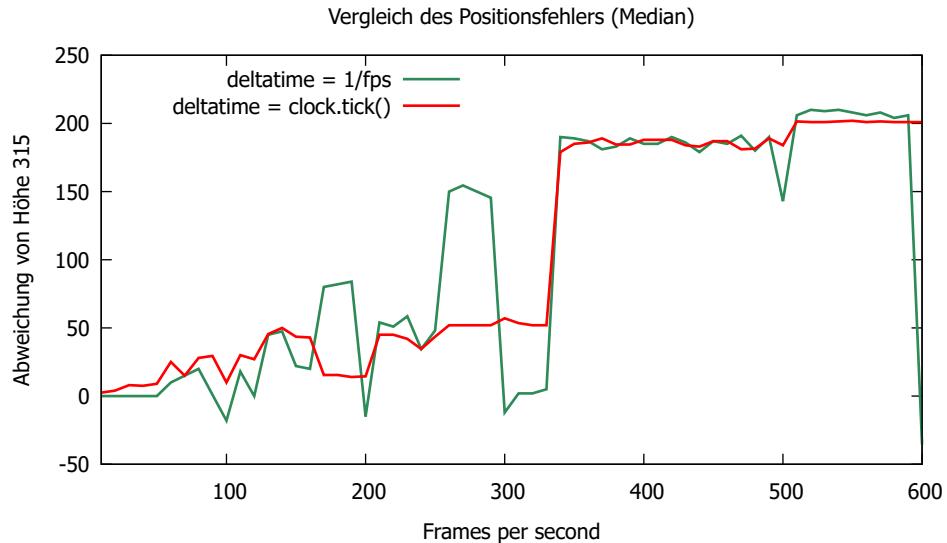
Zwei Probleme verursachen den Fehler in Abbildung 2.22 auf Seite 42:

- 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 der Kodierung einer **Fließkommazahl** ein Wert nahe des tatsächlichen Wertes abgespeichert; also beispielsweise bei $\frac{1.0}{30.0}$ nicht $0.0\overline{3}$, 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.

Rundungs-
fehler

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 48 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 42 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     window.flip()
48     Settings.DELTATIME = clock.tick(Settings.FPS) / 1000.0 # Korrekturwert ermitteln
49     print(f"top={defender_rect.top}")

```

Ursache ist ein Problem, welches wir hätten sofort lösen müssen. Bei der Zuweisung in Zeile 37 in Quelltext 2.21 auf Seite 36 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

FRect

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

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

Quelltext 2.23: Normalisierte Bewegung mit Positionsangaben in float

```

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

```

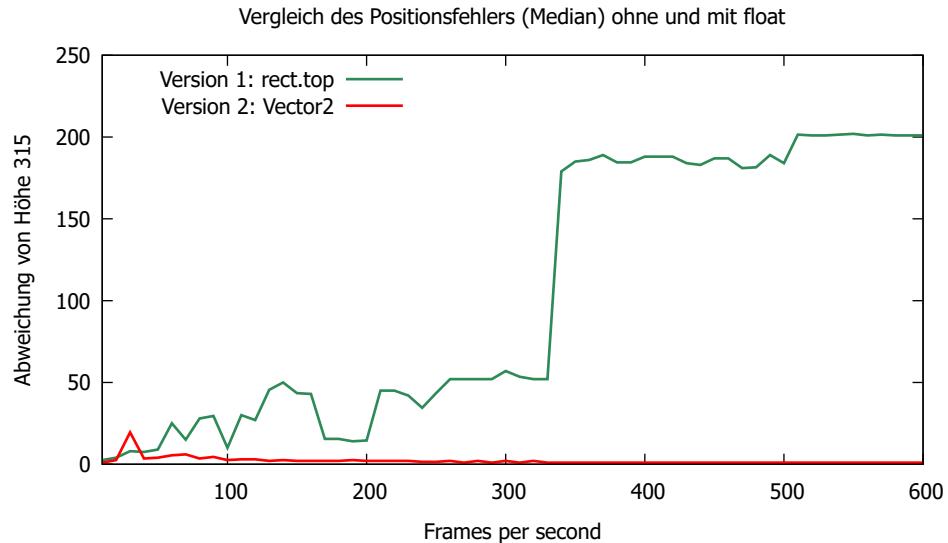
In Abbildung 2.24 auf Seite 43 können wir sehen, dass das Ergebnis schon deutlich besser geworden ist. Auch hat sich die Abweichung von optimalen Wert 315 dramatisch verbessert. In Abbildung 2.20 auf der nächsten Seite wird der Unterschied sichtbar gemacht.

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

In Zeile 29 von Quelltext 2.24 auf Seite 41 wird mit Hilfe von `time.time()` die Anzahl der Sekunden nach dem 01.01.1970 als eine Fließkommazahl (float) zurückgeliefert. Die Nachkommastellen geben dabei die Sekundenbruchteile an. Diese Messung ist genauer als die durch `pygame.clock.tick()` und liefert mir je nach Zeitmessungsmöglichkeiten der Rechnerarchitektur und des Betriebssystems mehr Nachkommastellen – bis in den Nanosekundenbereich.

In Zeile 52 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,

`time()`



nun mit mehr Nachkommastellen. Anschließend wird in Zeile 54 der neue Startzeit des nächsten Frames festgehalten, um nach dem nächsten Frame wieder den Zeitverbrauch ermitteln zu können.

Abbildung 2.25 auf Seite 43 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.

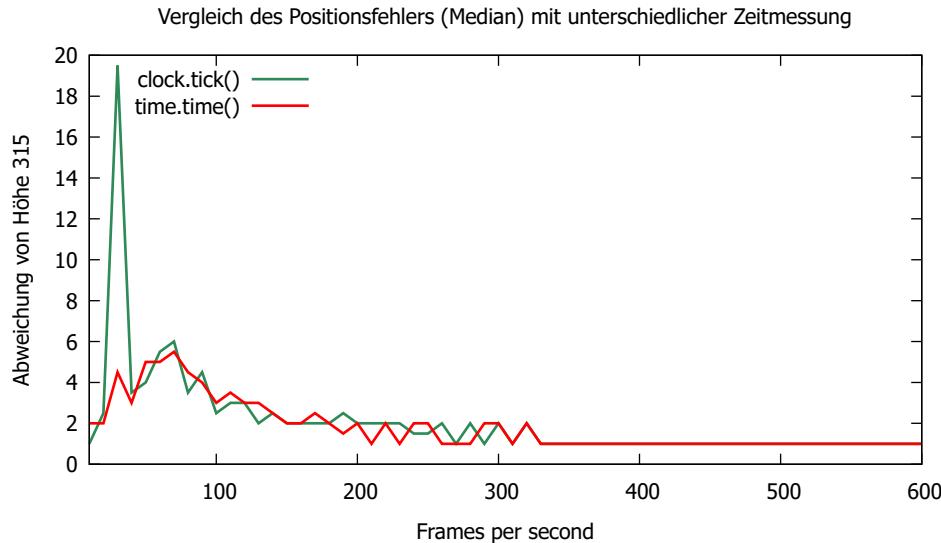


Abbildung 2.21: Vergleich der Positionsfehler mit unterschiedlichen Genauigkeiten

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

```

14 def main():
15     pygame.init()
16     window = pygame.Window(size=Settings.WINDOW.size, title="Bewegung", position=(10, 50))
17     screen = window.get_surface()
18     clock = pygame.time.Clock()
19
20     defender_image = pygame.image.load("images/defender01.png").convert_alpha()
21     defender_image = pygame.transform.scale(defender_image, (30, 30))
22     defender_rect = pygame.rect.FRect(defender_image.get_rect())
23     defender_rect.centerx = Settings.WINDOW.centerx
24     defender_rect.bottom = Settings.WINDOW.height - 5
25     defender_speed = 600
26     defender_direction_v = -1
27
28     start_time = pygame.time.get_ticks()
29     time_previous = time() # Startzeit festhalten
30     running = True
31     while running:
32         if pygame.time.get_ticks() > start_time + Settings.LIMIT:
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 * Settings.DELTATIME
41         if defender_rect.bottom >= Settings.WINDOW.height:
42             defender_direction_v *= -1
43         elif defender_rect.top <= 0:
44             defender_direction_v *= -1
45
46         # Draw
47         screen.fill("white")
48         pygame.draw.line(screen, "red", (0, 315), (Settings.WINDOW.width, 315), 2)

```

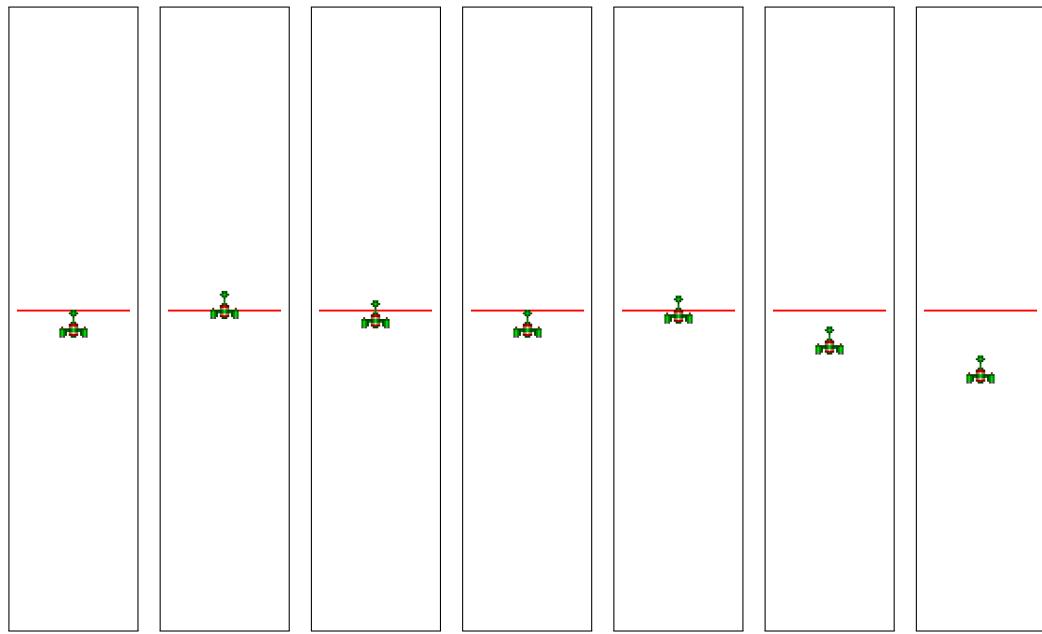


Abbildung 2.22: Normalisierte Bewegung mit $1/fps$ bei gleicher Geschwindigkeit aber unterschiedlichen Frameraten (von links nach rechts: 10, 30, 60, 120, 240, 300, 600)

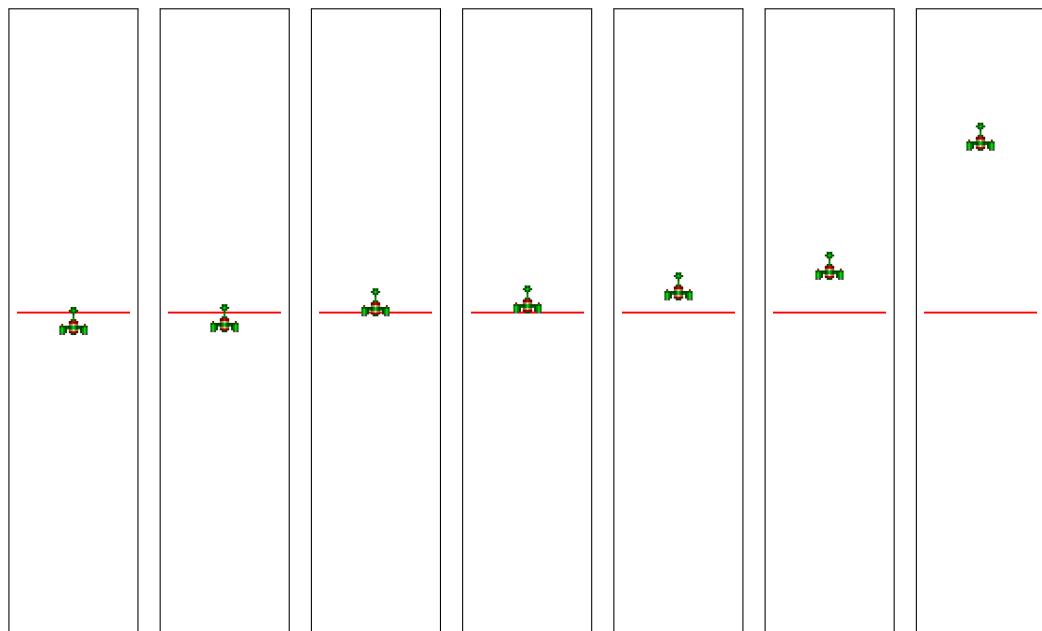


Abbildung 2.23: Normalisierte Bewegung mit `pygame.clock.tick()` bei gleicher Geschwindigkeit aber unterschiedlichen Frameraten (von links nach rechts: 10, 30, 60, 120, 240, 300, 600)

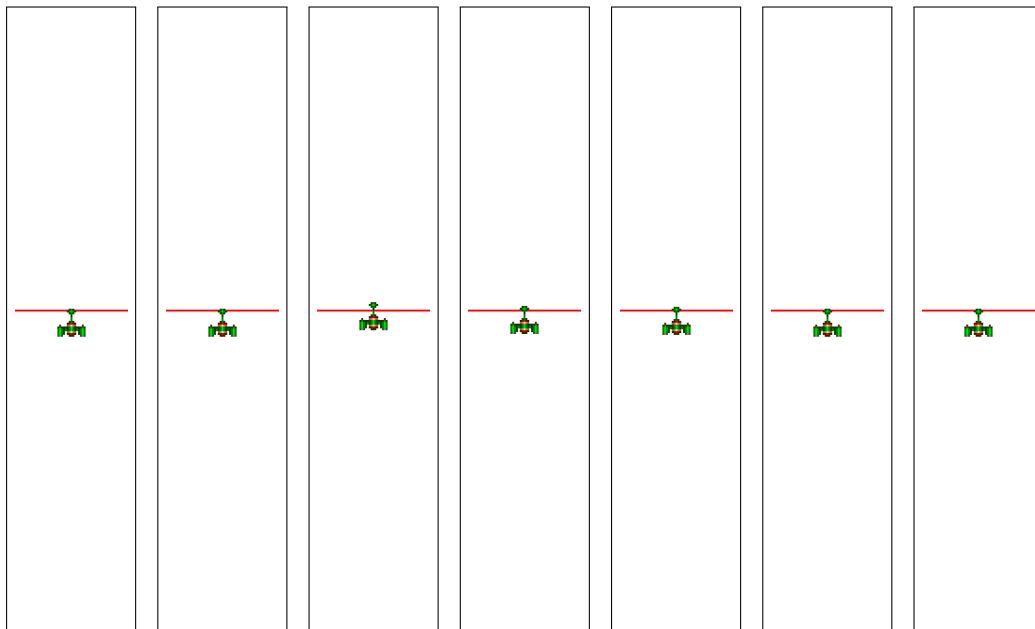


Abbildung 2.24: Normalisierte Bewegung mit `pygame.clock.tick()` (float) bei gleicher Geschwindigkeit aber unterschiedlichen Frameraten (von links nach rechts: 10, 30, 60, 120, 240, 300, 600)

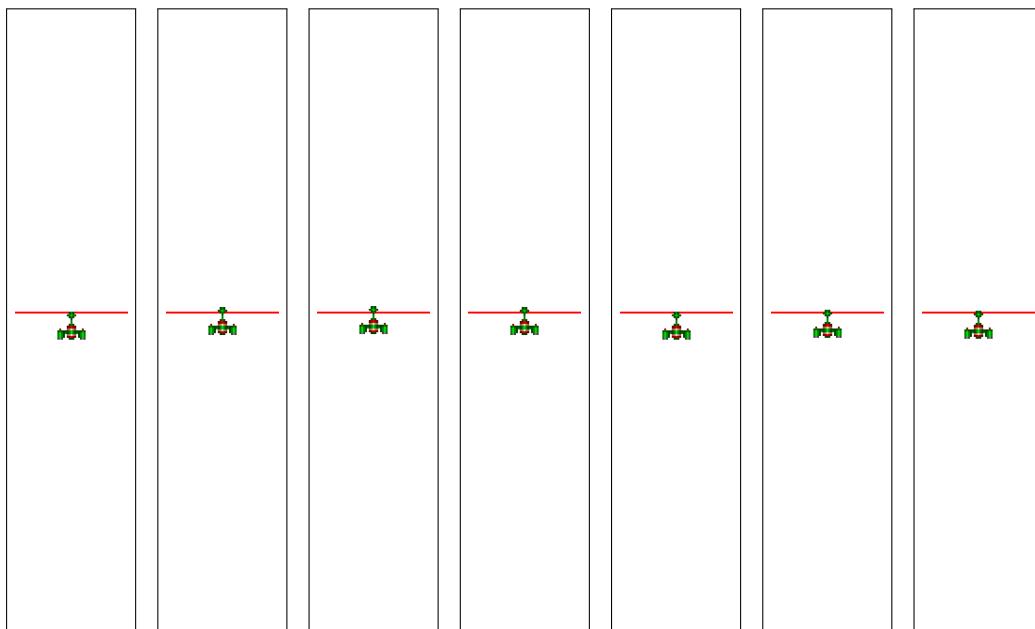


Abbildung 2.25: Normalisierte Bewegung mit `time.time()` bei gleicher Geschwindigkeit aber unterschiedlichen Frameraten (von links nach rechts: 10, 30, 60, 120, 240, 300, 600), Version 3

```
49     screen.blit(defender_image, defender_rect)
50     window.flip()
51     clock.tick(Settings.FPS)
52     time_current = time()  # Aktuelle Zeit festhalten
53     Settings.DELTATIME = time_current - time_previous  # Zeitverbrauch
54     time_previous = time_current  # Neue Startzeit
55     pygame.quit()
```

Was war neu?

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

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

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

Es wurden folgende Pygame-Elemente eingeführt:

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

2.5 Sprite-Klasse

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

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

`Sprite`

Formulieren wir also die Klasse `Defender` als eine Kindklasse von `Sprite` (Zeile 12).

Quelltext 2.25: Sprites (1), Version 1.0

```

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

```

Die Zeilen des Konstruktors (Zeile 14ff.) entsprechen denen der vorherigen Version. Lediglich der Präfix `defender_` wird durch `self.` ersetzt, wodurch die Variablen zu Attributen der Klasse werden. Sie sollten keine Schwierigkeiten haben, diese zu verstehen.

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

`self.rect`
`self.image`

In Zeile 23ff. werden die Randkollisionen und die Zustandsänderungen formuliert. Hier fällt besonders die Berechnung der neuen Position mit `move()` auf.

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

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

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

Quelltext 2.26: Sprites (2), Version 1.0

```

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

```

Die Verwendung der Klasse `Defender` ist nun denkbar einfach geworden. In der Zeile 45 wird ein Objekt der Klasse erzeugt. In Zeile 56 wird `update()` aufgerufen und in Zeile 60 `draw()`.

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

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

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

Bei `move_ip()` wird in Zeile 24 die Methode `pygame.Rect.move_ip()` eingeführt. Sie arbeitet wie `move()`, nur dass hier die Änderung direkt im Rechteck durchgeführt wird; `ip` steht hier für *in place*. Bei `move()` bleibt das ursprüngliche Rechteck unverändert.

Quelltext 2.27: Sprites (1), Version 1.1

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

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

Quelltext 2.28: Sprites (2), Version 1.1

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

Nun erzeuge ich die beiden Ränder:

Quelltext 2.29: Sprites (3), Version 1.1

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

Bisher war alles easy.

Quelltext 2.30: Sprites (4), Version 1.1

```
65     # Update
66     if pygame.sprite.collide_rect(defender, border_left):
67         defender.change_direction()
68     elif pygame.sprite.collide_rect(defender, border_right):
69         defender.change_direction()
70     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

```

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

Die erste Stelle ist Zeile 61 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`- oder `FRect`-Objekt mit dem Namen `rect` als Attribut haben. Der dritte Parameter – hier `False` – steuert, ob das kollidierende Sprite aus der Liste entfernt werden soll. Dieses Feature ist in Spielen recht interessant, will man doch beispielsweise Felsen, die von einem Raumschiff zerschossen wurden, löschen.

Die zweite Stelle ist Zeile 68. 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.

Group

spritecollide()

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 61f.).

`GroupSingle`

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

```

41  class Game(object):
42
43      def __init__(self) -> None:
44          pygame.init()
45          self.window = pygame.Window(size=Settings.WINDOW.size, title="Sprite", position=(10,
46                                         50))
46          self.screen = self.window.get_surface()
47          self.clock = pygame.time.Clock()
48
49          self.defender = pygame.sprite.GroupSingle(Defender())
50          self.all_border = pygame.sprite.Group()
51          self.all_border.add(Border("left"))
52          self.all_border.add(Border("right"))
53          self.running = False
54
55      def run(self) -> None:
56          time_previous = time()
57          self.running = True
58          while self.running:
59              self.watch_for_events()
60              self.update()
61              self.draw()
62              self.clock.tick(Settings.FPS)
63              time_current = time()
64              Settings.DELTATIME = time_current - time_previous
65              time_previous = time_current
66              pygame.quit()
67
68      def watch_for_events(self) -> None:
69          for event in pygame.event.get():
70              if event.type == pygame.QUIT:
71                  self.running = False
72
73      def update(self) -> None:
74          if pygame.sprite.spritecollide(self.defender.sprite, self.all_border, False):
75              self.defender.sprite.change_direction() # Gefällt mir nicht!
76          self.defender.update()
77
78      def draw(self) -> None:
79          self.screen.fill((255, 255, 255))
80          self.defender.draw(self.screen)
81          self.all_border.draw(self.screen)
82          self.window.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 75 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 `update()`
aufzurufen. So wird `change_direction()` nach Zeile 28 durch `update()` aufgerufen und nicht mehr von außen.

Quelltext 2.33: `Defender.update()`

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

Der Aufruf erfolgt dann in Quelltext 2.34 in Zeile 80 indirekt durch die Verwendung des Übergabeparameters.

Quelltext 2.34: `Game.update()`

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

Was war neu?

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

Es wurden folgende Pygame-Elemente eingeführt:

- `pygame.Rect.move()`:
<https://pyga.me/docs/ref/rect.html#pygame.Rect.move>
- `pygame.Rect.move_ip()`:
https://pyga.me/docs/ref/rect.html#pygame.Rect.move_ip
- `pygame.sprite.Group`:
<https://pyga.me/docs/ref/sprite.html#pygame.sprite.Group>
- `pygame.sprite.GroupSingle`:
<https://pyga.me/docs/ref/sprite.html#pygame.sprite.GroupSingle>
- `pygame.sprite.GroupSingle.sprite`:
<https://pyga.me/docs/ref/sprite.html#pygame.sprite.GroupSingle>

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

2.6 Tastatur

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

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

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

```

7  class Settings:
8      WINDOW = pygame.rect.Rect((0, 0), (150, 150))
9      FPS = 60
10     DELTATIME = 1.0 / FPS
11     DIRECTIONS = {
12         "right": pygame.math.Vector2(1, 0),
13         "left": pygame.math.Vector2(-1, 0),
14         "up": pygame.math.Vector2(0, -1),
15         "down": pygame.math.Vector2(0, 1),
16     }

```

Danach bereiten wir die Verteidiger-Klasse vor bzw. wandeln sie ein wenig ab (Quelltext [2.36](#)). Das Sprite wird nun nicht mehr unten sondern mittig platziert (Zeile [26](#)), und das Raumschiff soll sich jetzt auch vertikal bewegen können. Dazu braucht es entweder zwei entsprechende Variablen oder aber ein `Vector2`-Objekt. Ich nehme ein `Vector2`-Objekt (Zeile [27](#)), wobei das erste Element der Richtungsvektor der horizontalen und das zweite der vertikalen Richtung ist. Der jeweilige Richtungsvektor wird dabei entsprechend der schon oben vorgestellten Semantik gesetzt. In der Methode `change_direction()` wird entweder der Richtungsvektor gesetzt oder die Geschwindigkeit angepasst. Bewegen und Stehenbleiben wird einfach dadurch erreicht, dass ich die Geschwindigkeit bei `start` auf 100 bzw. bei `stop` auf 0 setze.

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

```

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

```

```

35         if self.rect.left < Settings.WINDOW.left:
36             self.rect.left = Settings.WINDOW.left + 1
37         elif self.rect.right > Settings.WINDOW.right:
38             self.rect.right = Settings.WINDOW.right - 1
39         if self.rect.top < Settings.WINDOW.top:
40             self.rect.top = Settings.WINDOW.top + 1
41         elif self.rect.bottom > Settings.WINDOW.bottom:
42             self.rect.bottom = Settings.WINDOW.bottom - 1
43     elif kwargs["action"] == "switch":
44         self.direction *= -1
45     elif "direction" in kwargs.keys():
46         self.change_direction(kwargs["direction"])
47
48     def change_direction(self, direction: str) -> None:
49         if direction in Settings.DIRECTIONS.keys():
50             self.direction = Settings.DIRECTIONS[direction]
51         elif direction == "stop":
52             self.speed = 0
53         elif direction == "start":
54             self.speed = 100

```

Kommen wir jetzt zur eigentlichen Tastaturverarbeitung: Das Verwenden einer Taste kann die Ereignistypen `pygame.KEYDOWN` oder `pygame.KEYUP` auslösen. In unserem Beispiel (Zeile 84) 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 59).

Quelltext 2.37: Bewegung durch Tastatur steuern (4), `Game.watch_for_events()`

```

80     def watch_for_events(self) -> None:
81         for event in pygame.event.get():
82             if event.type == pygame.QUIT:
83                 self.running = False
84             elif event.type == pygame.KEYDOWN: # Taste drücken
85                 if event.key == pygame.K_ESCAPE: # Boss-Taste
86                     self.running = False
87                 elif event.key == pygame.K_RIGHT: # Pfeiltasten
88                     self.defender.update(direction="right")
89                 elif event.key == pygame.K_LEFT:
90                     self.defender.update(direction="left")
91                 elif event.key == pygame.K_UP:
92                     self.defender.update(direction="up")
93                 elif event.key == pygame.K_DOWN:
94                     self.defender.update(direction="down")
95                 elif event.key == pygame.K_SPACE: # Leerzeichen-Taste
96                     self.defender.update(action="switch")
97                 elif event.key == pygame.K_r:
98                     if event.mod & pygame.KMOD_LSHIFT: # Shift-Taste
99                         self.defender.update(direction="stop")
100                else:
101                    self.defender.update(direction="start")

```

Fangen wir mit der Boss-Taste an. In Zeile 85 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!

Danach werden mit Hilfe von `K_LEFT`, `K_RIGHT`, `K_UP` und `K_DOWN` ab Zeile 87ff. die vier

`KEYDOWN`
`KEYUP`

`key`

`K_ESCAPE`
`K_LEFT`
`K_RIGHT`
`K_UP`
`K_DOWN`

Pfeiltasten abgefragt und die entsprechende Nachricht an den Verteidiger gesendet.

Mit Hilfe der Leerzeichen-Taste `K_SPACE` wird das Raumschiff in Zeile 95 gestoppt.

`K_SPACE`

Um den Einsatz der Shift-Taste (Umschalttaste) mal zu demonstrieren, habe ich hier das `r` doppelt belegt (Zeile 98). 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.

`event.mod`
`KMOD_LSHIFT`

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

Was war neu?

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

Es wurden folgende Pygame-Elemente eingeführt:

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

Tabelle 2.5: Liste von vordefinierten Tastaturkonstanten

Konstante	Bedeutung	Beschreibung
<code>K_BACKSPACE</code>	<code>\b</code>	Löschen (backspace)
<code>K_TAB</code>	<code>\t</code>	Tabulator
<code>K_CLEAR</code>		Leeren
<code>K_RETURN</code>	<code>\r</code>	Eingabe (return, enter)
<code>K_PAUSE</code>		Pause
<code>K_ESCAPE</code>	<code>^ [</code>	Abbruch (escape)
<code>K_SPACE</code>		Leerzeichen (space)
<code>K_EXCLAIM</code>	<code>!</code>	Ausrufezeichen
<code>K_QUOTEDBL</code>	<code>"</code>	Gänsefüßchen
<code>K_HASH</code>	<code>#</code>	Doppelkreuz (hash)
<code>K_DOLLAR</code>	<code>\$</code>	Dollar
<code>K_AMPERSAND</code>	<code>&</code>	Kaufmannsund
<code>K_QUOTE</code>	<code>'</code>	Hochkomma
<code>K_LEFTPAREN</code>	<code>(</code>	Linke runde Klammer
<code>K_RIGHTPAREN</code>	<code>)</code>	Rechte runde Klammer

Tabelle 2.5: Liste von vordefinierten Tastaturkonstanten (Fortsetzung)

Konstante	Bedeutung	Beschreibung
K_ASTERISK	*	Sternchen
K_PLUS	+	Plus
K_COMMA	,	Komma
K_MINUS	-	Minus
K_PERIOD	.	Punkt
K_SLASH	/	Schrägstrich
K_0	0	0
K_1	1	1
K_2	2	2
K_3	3	3
K_4	4	4
K_5	5	5
K_6	6	6
K_7	7	7
K_8	8	8
K_9	9	9
K_COLON	:	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

Tabelle 2.5: Liste von vordefinierten Tastaturkonstanten (Fortsetzung)

Konstante	Bedeutung	Beschreibung
K_l	l	l
K_m	m	m
K_n	n	n
K_o	o	o
K_p	p	p
K_q	q	q
K_r	r	r
K_s	s	s
K_t	t	t
K_u	u	u
K_v	v	v
K_w	w	w
K_x	x	x
K_y	y	y
K_z	z	z
K_DELETE		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

Tabelle 2.5: Liste von vordefinierten Tastaturkonstanten (Fortsetzung)

Konstante	Bedeutung	Beschreibung
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
K_F14		F14
K_F15		F15
K_NUMLOCK		Umschalten Zahlen
K_CAPSLOCK		Umschalten Großbuchstaben
K_SCROLLLOCK		Umschalten auf scrollen
K_RSHIFT		Rechte Umschalttaste
K_LSHIFT		Linke Umschalttaste
K_RCTRL		Rechte Steuerungstaste
K_LCTRL		Linke Steuerungstaste
K_RALT		Rechte Alternativtaste
K_LALT		Linke Alternativtaste
K_RMETA		Rechte Metataste
K_LMETA		Linke Metataste
K_LSUPER		Linke Windowstaste
K_RSUPER		Rechte Windowstaste
K_MODE		AltGr Umschalter
K_HELP		Hilfe
K_PRINT		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 Alternativtaste
KMOD_RALT	Rechte Alternativtaste
KMOD_ALT	Linke oder rechte Alternativtaste oder beide
KMOD_LMETA	Linke Metataste
KMOD_RMETA	Rechte Metataste
KMOD_META	Linke oder rechte Metataste oder beide
KMOD_CAPS	Umschalten Großbuchstaben
KMOD_NUM	Umschalten Zahlen
KMOD_MODE	AltGr Umschalter

2.7 Textausgabe mit Fonts

2.7.1 Default-Font



Abbildung 2.26: Textausgabe mit Fonts

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

Quelltext 2.38: Text mit Fonts ausgeben (1), Präambel

```

1  from typing import Tuple
2
3  import pygame
4
5
6  class Settings:
7      WINDOW = pygame.rect.Rect((0, 0), (700, 300))
8      FPS = 60

```

Font
get_default_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 30 liefert mir die vom Betriebssystem eingestellte Zeichsatzvorgabe. Die Schriftgröße (**fontsize**) legen wir nach Bedarf einfach fest.

Quelltext 2.39: Text mit Fonts ausgeben (2), `TextSprite`

```

11  class TextSprite(pygame.sprite.Sprite):
12      def __init__(self, fontsize: int, fontcolor: list[int], center: Tuple[int, int], text:
13          str = "HelloWorld!") -> None:
14          super().__init__()
15          self.image = None
16          self.rect = None
17          self.fontsize = fontsize
18          self.fontcolor = fontcolor
19          self.fontsize_update(0) # 0!
20          self.text = text
21          self.center = center
22          self.render() # Alle Infos zusammen
23
23  def render(self) -> None:
24      self.image = self.font.render(self.text, True, self.fontcolor) # Bitmap
25      self.rect = self.image.get_rect()
26      self.rect.center = self.center
27
28  def fontsize_update(self, step: int = 1) -> None:
29      self.fontsize += step
30      self.font = pygame.font.Font(pygame.font.get_default_font(), self.fontsize) #
31
32  def fontcolor_update(self, delta: Tuple[int, int, int]) -> None:
33      for i in range(3):
34          self.fontcolor[i] = (self.fontcolor[i] + delta[i]) % 256
35
36  def update(self) -> None:
37      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 18. 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 21 mit Hilfe von `pygame.font.render()` das Surface-Objekt erzeugen (Zeile 24). 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 46). Darunter soll durch `info` ausgegeben werden, mit welcher Schriftgröße und -farbe der Gruß erzeugt wurde (Zeile 46).

Quelltext 2.40: Text mit Fonts ausgeben (3), Hauptprogramm

```

40 def main():
41     pygame.init()
42     window = pygame.Window(size=Settings.WINDOW.size, title="Textausgabe mit Fonts",
43                             position=(10, 50))
44     screen = window.get_surface()
45     clock = pygame.time.Clock()
46
47     hello = TextSprite(24, [255, 255, 255], (Settings.WINDOW.center)) # Gruß
48     info = TextSprite(12, [255, 0, 0], (Settings.WINDOW.centerx, Settings.WINDOW.bottom -
49                                     20)) # Fontinfo
50     all_sprites = pygame.sprite.Group()
51     all_sprites.add(hello, info)
52
53     running = True
54     while running:
55         for event in pygame.event.get():
56             if event.type == pygame.QUIT:
57                 running = False
58             elif event.type == pygame.KEYDOWN:
59                 if event.key == pygame.K_ESCAPE:
60                     running = False
61                 elif event.key == pygame.K_KP_PLUS or event.key == pygame.K_PLUS: # Größer
62                     hello.fontsize_update(+1)
63                 elif event.key == pygame.K_KP_MINUS or event.key == pygame.K_MINUS: # Kleiner
64                     hello.fontsize_update(-1)
65                 elif event.key == pygame.K_r:
66                     if event.mod & pygame.KMOD_SHIFT:
67                         hello.fontcolor_update((-1, 0, 0)) # Weniger Rot
68                     else:
69                         hello.fontcolor_update((+1, 0, 0)) # Mehr Rot
70                 elif event.key == pygame.K_g:
71                     if event.mod & pygame.KMOD_SHIFT:
72                         hello.fontcolor_update((0, -1, 0)) # Weniger Grün
73                     else:
74                         hello.fontcolor_update((0, +1, 0)) # Mehr Grün
75                 elif event.key == pygame.K_b:
76                     if event.mod & pygame.KMOD_SHIFT:
77                         hello.fontcolor_update((0, 0, -1)) # Weniger Blau
78                     else:
79                         hello.fontcolor_update((0, 0, +1)) # Mehr Blau
80
81             info.text = f"size={hello.fontsize}, r={hello.fontcolor[0]}, g={hello.fontcolor[1]}, b={hello.fontcolor[2]}"
82             all_sprites.update()
83             screen.fill((200, 200, 200))
84             all_sprites.draw(screen)
85             window.flip()
86             clock.tick(Settings.FPS)
87
88     pygame.quit()

```

Dieser Gruß kann durch die Plus- und Minus-Tasten in seiner Größe verändert werden (Zeile 59ff.). Die Tasten **r**, **g** und **b** werden dazu verwendet, den jeweiligen Farbkanal zu manipulieren. Der Großbuchstabe erhöht den Wert (z.B. in Zeile 65), der Kleinbuchstabe reduziert ihn (z.B. in Zeile 67).

In Abbildung 2.26 auf Seite 60 können Sie eine mögliche Darstellung sehen.

2.7.2 Fontliste

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



Abbildung 2.27: Fontliste

Quelltext 2.41: Fontliste (1), Präambel, Settings und TextSprite

```

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

```

```

27
28     def fontcolor_update(self, delta: list[int]) -> None:
29         for i in range(3):
30             self.fontcolor[i] = (self.fontcolor[i] + delta[i]) % 256
31
32     def update(self) -> None:
33         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 46) 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.

Quelltext 2.42: Fontliste (2), BigImage

```

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

```

Und jetzt das Hauptprogramm. Im ersten Teil wird über `pygame.font.get_fonts()` (Zeile 64) eine Liste aller installierten Fontnamen ermittelt. Dieser Name wird dem Konstruktur von `TestSprite` übergeben. Mit Hilfe der Methode `pygame.font.match_font()` (Zeile 26) 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 2.43: Fontliste (3), Hauptprogramm (1)

```

58 def main():
59     pygame.init()
60     window = pygame.Window(size=Settings.WINDOW.size, title="Fontliste", position=(10, 50))
61     screen = window.get_surface()
62     clock = pygame.time.Clock()

```

```

63
64     fonts = pygame.font.get_fonts()  # Ermittle installierte Fonts
65
66     list_of_fontsprites = pygame.sprite.Group()
67     height = 0
68     width = 0
69     for name in fonts:
70         try:
71             t = TextSprite(name, 24, [0, 0, 255])
72             t.rect.top = height
73             height += t.rect.height
74             width = t.rect.width if t.rect.width > width else width
75             list_of_fontsprites.add(t)
76         except OSError as err:
77             print(f"OS error: {err}")
78         except pygame.error as perr:
79             print(f"Pygame error: {perr} with font {name}")
80
81     bigimage = pygame.sprite.GroupSingle(BigImage())
82     bigimage.sprite.create_image(width, height)
83     list_of_fontsprites.draw(bigimage.sprite.image_total)  #

```

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 83).

Quelltext 2.44: Fontliste, Hauptprogramm (2)

```

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

```

Die Hauptprogrammschleife übernimmt nun nur noch das Blättern (jeweils um ein Drittel der Bildschirmhöhe) und das Programmende.

Was war neu?

Es wurden folgende Pygame-Elemente eingeführt:

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

- `pygame.font.get_default_font()`:
https://pyga.me/docs/ref/font.html#pygame.font.get_default_font
- `pygame.font.get_fonts()`:
https://pyga.me/docs/ref/font.html#pygame.font.get_fonts
- `pygame.font.match_font()`:
https://pyga.me/docs/ref/font.html#pygame.font.match_font
- `pygame.font.Font.render()`:
<https://pyga.me/docs/ref/font.html#pygame.font.Font.render>
`pygame.Surface.subsurface()`:
<https://pyga.me/docs/ref/surface.html#pygame.Surface.subsurface>

2.8 Textausgabe mit Bitmaps

Oft erfolgen Textausgaben nicht über Fonts, sondern über eine [Spritelib](#). In einer solchen befinden sich dann Schriftzeichen, Symbole oder Ziffern, die dann meist auch in einem besonderen dem Spiel angepassten Design sind. In Abbildung 2.28 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 2.30 auf Seite [72](#)).

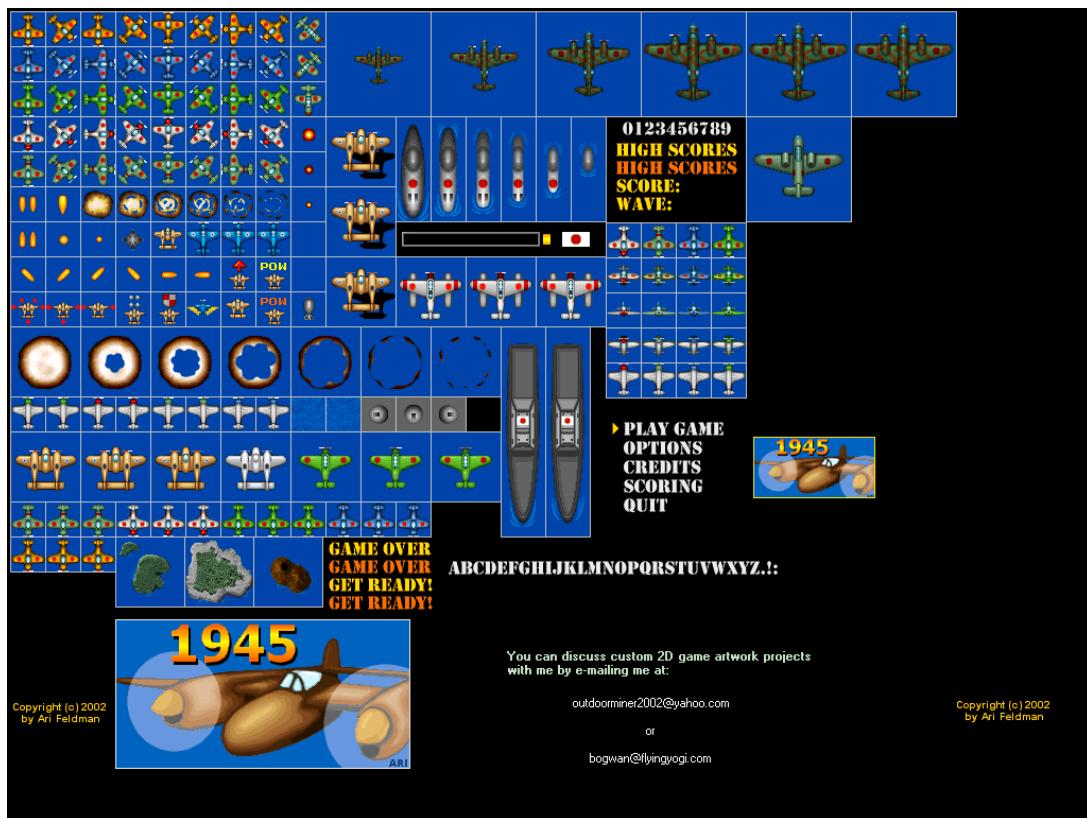


Abbildung 2.28: Beispiel für eine Spritelib

Der erste Teil von Quelltext 2.45 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 2.45: Textbitmaps (1), Präambel und `Settings`

```

1 import os
2
3 import pygame
4
5
6 class Settings:
7
8     WINDOW = pygame.Rect((0, 0), (700, 650))
9     PATH: dict[str, str] = {}
10    PATH["file"] = os.path.dirname(os.path.abspath(__file__))
11    PATH["image"] = os.path.join(PATH["file"], "images")
12    FPS = 60
13
14    @staticmethod
15    def filepath(name: str) -> str:
16        return os.path.join(Settings.PATH["file"], name)
17
18    @staticmethod
19    def imagepath(name: str) -> str:
20        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 2.29 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 2.29 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 2.29 können Sie die Abstände sehen. Hier sind es jeweils 14 *px*.

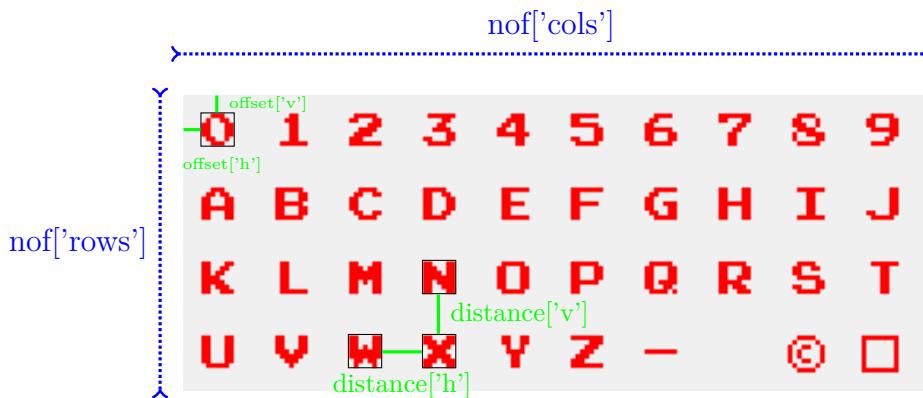


Abbildung 2.29: Bedeutung der Angaben in Spritelib

Quelltext 2.46: Textbitmaps (2), Spritelib

```

23 class Spritelib(pygame.sprite.Sprite):
24
25     def __init__(self, filename: str) -> None:
26         super().__init__()
27         self.image = pygame.image.load(Settings.imagepath(filename)).convert()
28         self.rect = self.image.get_rect()
29         self.nof = {"rows": 4, "cols": 10}
30         self.letter = {"width": 18, "height": 18}
31         self.offset = {"h": 6, "v": 6}
32         self.distance = {"h": 14, "v": 14}
33
34     def draw(self, screen: pygame.surface.Surface) -> None:
35         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 2.47: Textbitmaps (3): Konstruktor von `Letters`

```

38 class Letters(object):
39
40     def __init__(self, spritelib: Spritelib, colornumber: int) -> None:
41         super().__init__()
42         self.spritelib = spritelib
43         self.letters: dict[str, pygame.surface.Surface] = {}
44         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 88 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 93 wird die Position, also die Pixelkoordinaten des ersten Sprites ausgerechnet. Versuchen Sie doch selbst anhand der Angaben in Abbildung 2.29 auf der vorherigen Seite die Arithmetik nachzuvollziehen! Nur Mut, sie ist nicht schwierig, sondern nur lang.

Ab Zeile 94 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 95 wird die obere Koordinate und in Zeile 97 die linke Koordinate der Position berechnet. Wenn Sie Zeile 93 verstanden haben, sollten diese beiden Berechnungen keine Schwierigkeiten mehr bereiten. Höhe und Breite in Zeile 97 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 2.48: Textbitmaps (4): `create_letter_bitmap()` von Letters

```
46     def create_letter_bitmap(self, colordnumber: int):
47         lettername = (
48             "0",
49             "1", # Die Zeilen zwischen 59 und 82 werden übersprungen!
50             "y",
51             "z",
52             "-",
53             "u",
54             "copy",
55             "square",
56         ) #
57         index = 0
58         startpos = (
59             self.sprite.lib.offset["h"],
60             self.sprite.lib.offset["v"] + colordnumber * self.sprite.lib.nof["rows"] *
61             (self.sprite.lib.letter["height"] + self.sprite.lib.distance["v"]),
62         ) #
63         for row in range(self.sprite.lib.nof["rows"]): # Zeilen
64             for col in range(self.sprite.lib.nof["cols"]): # Spalten
65                 left = startpos[0] + col * (self.sprite.lib.letter["width"] +
66                     self.sprite.lib.distance["h"]) # 
67                 top = startpos[1] + row * (self.sprite.lib.letter["height"] +
68                     self.sprite.lib.distance["v"]) # 
69                 width, height = self.sprite.lib.letter.values() # Größe
70                 r = pygame.Rect(left, top, width, height)
71                 self.letters[lettername[index]] = self.sprite.lib.image.subsurface(r)
72                 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 2.49: Textbitmaps (5): `get_letter()` und `get_text()` von Letters

```

103     def get_letter(self, letter: str) -> pygame.surface.Surface:
104         if letter in self.letters:
105             return self.letters[letter]
106         else:
107             return self.letters["square"]
108
109     def get_text(self, text: str) -> pygame.surface.Surface:
110         l = len(text) * self.sprite.lib.letter["width"]
111         h = self.sprite.lib.letter["height"]
112         bitmap = pygame.Surface((l, h))
113         bitmap.set_colorkey((0, 0, 0))
114         for a in range(len(text)):
115             bitmap.blit(self.get_letter(text[a]), (a * self.sprite.lib.letter["width"], 0))
116         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 139: 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 141: 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 2.50: Textbitmaps (6): `TextBitmaps`

```

119 class TextBitmaps(object):
120
121     def __init__(self) -> None:
122         pygame.init()
123         self.window = pygame.Window(size=Settings.WINDOW.size, title="Textausgabe mit"
124                                     Bitmaps")
125         self.screen = self.window.get_surface()
126         self.clock = pygame.time.Clock()
127
128         self.filename = "chars.png"
129         self.running = False
130         self.input = ""
131
132     def watch_for_events(self) -> None:
133         for event in pygame.event.get():
134             if event.type == pygame.QUIT:
135                 self.running = False
136             elif event.type == pygame.KEYDOWN:
137                 if event.key == pygame.K_ESCAPE:
138                     self.running = False
139                 elif event.key == pygame.K_BACKSPACE:
140                     self.input = self.input[:-1] # Letztes Zeichen abschneiden
141                 else:
142                     self.input += event.unicode # Tastaturwert als unicode-Zeichen
143
144     def run(self) -> None:
145         sprite.lib = Spritelib(self.filename)
146         letters = Letters(sprite.lib, 2)
147         self.running = True

```

```
147     while self.running:
148         self.watch_for_events()
149         self.screen.fill((200, 200, 200))
150         self.screen.blit(letters.get_text(self.input), (400, 200))
151         spritelib.draw(self.screen)
152         self.window.flip()
153         self.clock.tick(Settings.FPS)
154
155     pygame.quit()
```

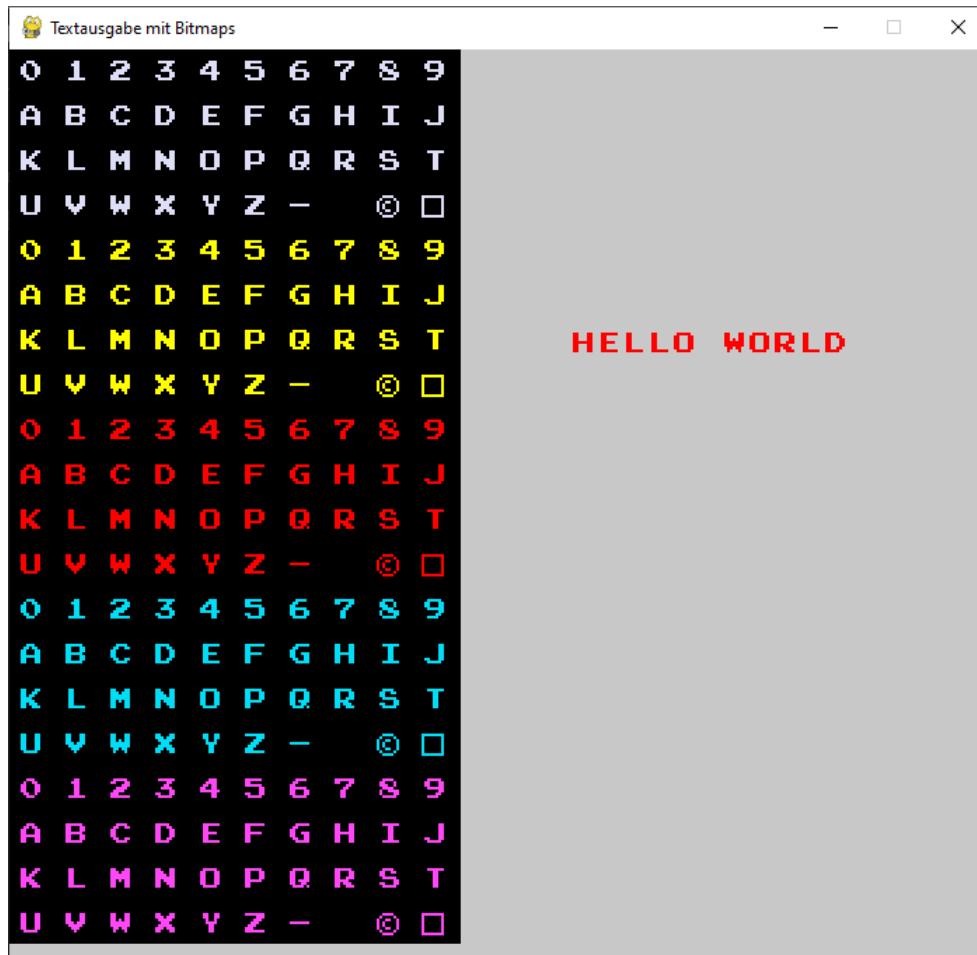


Abbildung 2.30: Textausgabe mit Bitmaps

Was war neu?

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

Es wurden folgende Pygame-Elemente eingeführt:

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

2.9 Kollisionserkennung

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

```

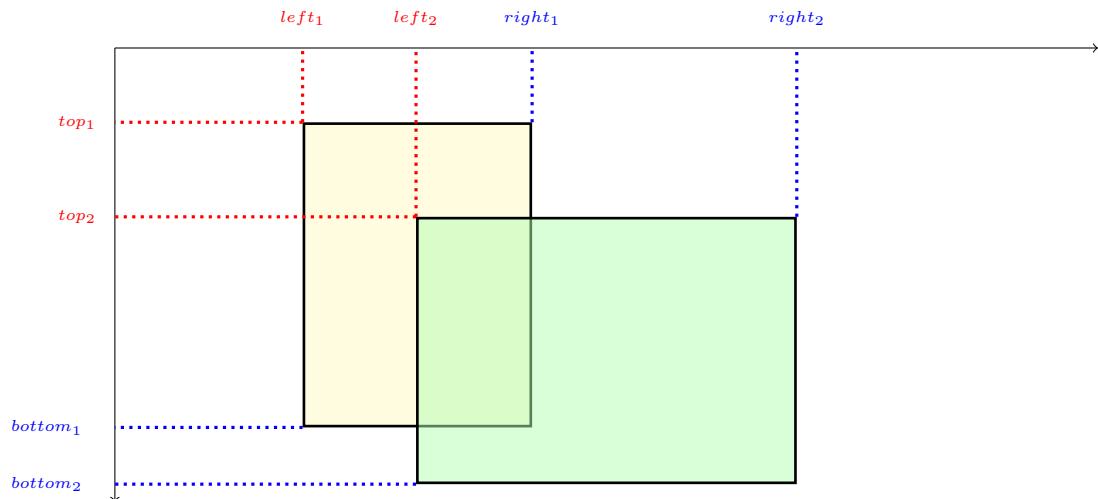


Abbildung 2.31: Kollisionserkennung mit Rechtecken

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

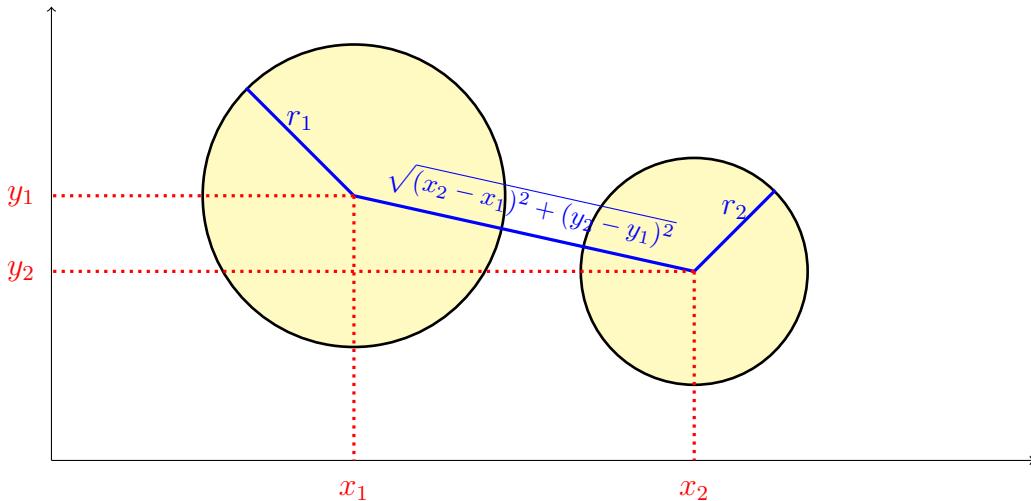


Abbildung 2.32: Kollisionserkennung mit Kreisen

- **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 innenhalb des Schnittmengen-Rechtecks liegen und die Pixel-Prüfung kann auf diesen in der Regel viel kleineren Bereich eingeschränkt werden. Ein weiteres Problem bei der Pixelprüfung ist, Hintergrund von Vordergrund zu unterscheiden. Woher soll die Pixelprüfung wissen, ob die Farbe Blau nun ein Teil des Objektes oder des Hintergrunds ist? Dazu gibt es mehrere Ansätze. Der einfachste ist, zu jedem Sprite ein schwarz/weiß-Bild zu erstellen (ein **Maske**); die weißen Pixel sind wichtig, die schwarzen können ignoriert werden. Nun wird die Pixelprüfung nur noch auf den Masken durchgeführt.

Maske

Schauen wir uns das Kollisionsverhalten mal im Detail an. In Abbildung 2.33 auf der nächsten Seite sehen wir vier Sprites: eine Mauer, ein Raumschiff, ein Monster und ein Geschoss. Keine der Sprites berühren sich.

In Abbildung 2.34 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.

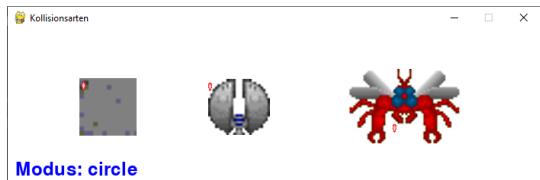


Abbildung 2.33: Vier Sprites

Abbildung 2.34: Rechtecksprüfung
(Montage)

Anders sieht es aus, wenn wir die Kollision durch die Innenkreise bestimmen lassen (Abbildung 2.35). 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 2.36). 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 2.35: Kreisprüfung
(Montage)Abbildung 2.36: 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 2.51: Kollisionsarten (1): Präambel und **Settings**

```

1 import os
2 from typing import Any
3
4 import pygame
5
6
7 class Settings(object):
8
9     WINDOW = pygame.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`.

get_rect()
self.rect
from_surface()

self.mask

self.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 2.52: 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 2.53: 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 2.54: Kollisionsarten (4): Konstruktor von `Game`, Konstruktor

```

68  class Game(object):
69
70      def __init__(self) -> None:
71          pygame.init()
72          self.window = pygame.Window(size=Settings.WINDOW.size, title=Settings.TITLE)
73          self.screen = self.window.get_surface()
74          self.clock = pygame.time.Clock()
75
76          self.font = pygame.font.Font(pygame.font.get_default_font(), 24)
77          self.bullet = pygame.sprite.GroupSingle(Bullet("shoot.png"))
78          self.all_obstacles = pygame.sprite.Group()
79          self.all_obstacles.add(Obstacle("brick1.png", "brick2.png"))
80          self.all_obstacles.add(Obstacle("raumschiff1.png", "raumschiff2.png"))
81          self.all_obstacles.add(Obstacle("alienbig1.png", "alienbig2.png"))
82          self.running = False

```

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

Quelltext 2.55: Kollisionsarten (5): `run()` und `watch_for_events()` von `Game`

```

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

```

```

95     for event in pygame.event.get():
96         if event.type == pygame.QUIT:
97             self.running = False
98         elif event.type == pygame.KEYDOWN:
99             if event.key == pygame.K_ESCAPE:
100                 self.running = False
101             elif event.key == pygame.K_DOWN:
102                 self.bullet.sprite.update(direction="down")
103             elif event.key == pygame.K_UP:
104                 self.bullet.sprite.update(direction="up")
105             elif event.key == pygame.K_LEFT:
106                 self.bullet.sprite.update(direction="left")
107             elif event.key == pygame.K_RIGHT:
108                 self.bullet.sprite.update(direction="right")
109             elif event.key == pygame.K_r:
110                 Settings.MODUS = "rect"
111             elif event.key == pygame.K_c:
112                 Settings.MODUS = "circle"
113             elif event.key == pygame.K_m:
114                 Settings.MODUS = "mask"
115             elif event.type == pygame.KEYUP:

```

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

Quelltext 2.56: Kollisionsarten (6): `update()` und `draw()` von Game

```

118     def update(self) -> None:
119         self.check_for_collision()
120         self.bullet.update(action="move")
121         self.all_obstacles.update()
122
123     def draw(self) -> None:
124         self.screen.fill("white")
125         self.all_obstacles.draw(self.screen)
126         self.bullet.draw(self.screen)
127         text_surface_modus = self.font.render(f"Modus: {Settings.MODUS}", True, "blue")
128         self.screen.blit(text_surface_modus, dest=(10, Settings.WINDOW.bottom - 30))
129         self.window.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 135 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 2.57: Kollisionsarten (7): `resize()` von Game

```

131     def resize(self) -> None:
132         total_width = 0
133         for s in self.all_obstacles:
134             total_width += s.rect.width
135         padding = (Settings.WINDOW.width - total_width) // 4  # Abstand

```

```

136     for i in range(len(self.all_obstacles)):
137         if i == 0:
138             self.all_obstacles.sprites()[i].rect.left = padding
139         else:
140             self.all_obstacles.sprites()[i].rect.left = 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 2.58: Kollisionsarten (8): `check_for_collision()` von Game

```

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

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.

spritecollide()

Quelltext 2.59: Kollisionsarten (9): Variante von `check_for_collision()` von Game

```

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

```

Und zu guter letzt noch der Aufruf:

Quelltext 2.60: 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://pyga.me/docs/ref/mask.html#pygame.mask.from_surface
- `pygame.sprite.collide_circle()`:
https://pyga.me/docs/ref/sprite.html#pygame.sprite.collide_circle
- `pygame.sprite.collide_mask()`:
https://pyga.me/docs/ref/sprite.html#pygame.sprite.collide_mask
- `pygame.sprite.collide_rect()`:
https://pyga.me/docs/ref/sprite.html#pygame.sprite.collide_rect
- `pygame.sprite.spritecollide()`:
<https://pyga.me/docs/ref/sprite.html#pygame.sprite.spritecollide>

2.10 Zeitsteuerung

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

Schauen wir uns zunächst die Bildschirmausgabe von Quelltext 2.61ff. in Abbildung 2.37 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 2.37: 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 2.61: 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((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 2.62: Zeitsteuerung (2), Version 1.0: `Enemy`

```

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

```

Auch `Bullet` ist in weiten Teilen eine Wiederholung. Interessant dürfte Zeile 57 sein. Die Methode `pygame.sprite.Sprite.kill()` ist nicht wirklich eine Selbstzerstörung. Vielmehr entfernt diese Methode das `Sprite`-Objekt aus allen `Spritegroups`. Wenn damit auch alle Referenzen verloren gehen, wird natürlich auch dieses Objekt zerstört, besteht aber noch irgendwo eine Referenz, bleibt das Objekt erhalten. In der Regel werden `Sprite`-Objekte aber in Gruppen (also in `pygame.sprite.Group`-Objekten) verwaltet und somit durch `kill()` zerstört. Sie können das in Abbildung 2.37 auf der vorherigen Seite dadurch erkennen, dass 30 *px* vor dem unteren Bildschirmrand der Feuerball verschwindet.

`kill()`

Quelltext 2.63: Zeitsteuerung (3), Version 1.0: `Bullet`

```

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

```

Im Konstruktor Der Klasse `Game` wird eine `Spritegroup` für die Feuerbälle angelegt und ein `GroupSingle`-Objekt für den `Enemy`. In `run()` erfolgt die übliche Abarbeitung der Teilaufgaben durch entsprechende Funktionsaufrufe. Ein kurzes Augenmerk möchte ich auf Zeile 79ff. lenken. Durch den Aufruf von `pygame.time.Clock.tick()` wird das Spiel getaktet – hier auf das 1/60 einer Sekunde und anschließend die *Deltatime* berechnet.

`tick()`

`deltatime`

Quelltext 2.64: Zeitsteuerung (4), Version 1.0: Konstruktor und `run()` von `Game`

```

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

```

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

Quelltext 2.65: Zeitsteuerung (5), Version 1.0: `watch_for_events()` und `draw()` von `Game`

```

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

```

Die Methode `update()` ist nur bzgl. Zeile 100 erwähnenswert, da dort ein neuer Feuerball erzeugt/abgeworfen wird, indem die Methode `new_bullet()` aufgerufen wird. Die Startposition ergibt sich aus der aktuellen Position des `Enemy`s. Das horizontale Zentrum von Feuerball und `Enemy` soll gleich sein. Das vertikale Zentrum ist etwas nach unten verschoben; sieht besser aus.

Quelltext 2.66: Zeitsteuerung (6), Version 1.0: `update()` und `new_bullet()` von `Game`

```

99  def update(self) -> None:
100     self.new_bullet()  # Feuerballabwurf
101     self.all_bullets.update()
102     self.enemy.update()
103
104  def new_bullet(self) -> None:
105      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 79 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 70 und Zeile 71).

Quelltext 2.67: Zeitsteuerung (7), Version 1.1: Konstruktor von `Game`

```

69  self.all_bullets = pygame.sprite.Group()
70  self.time_counter = 0  # Zähler
71  self.time_range = 30  # Obergrenze
72  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 2.38 auf der nächsten Seite: Es sind nur noch zwei Feuerbälle sichtbar.

Quelltext 2.68: Zeitsteuerung (8), Version 1.1: `new_bullet()` von `Game`

```

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

```

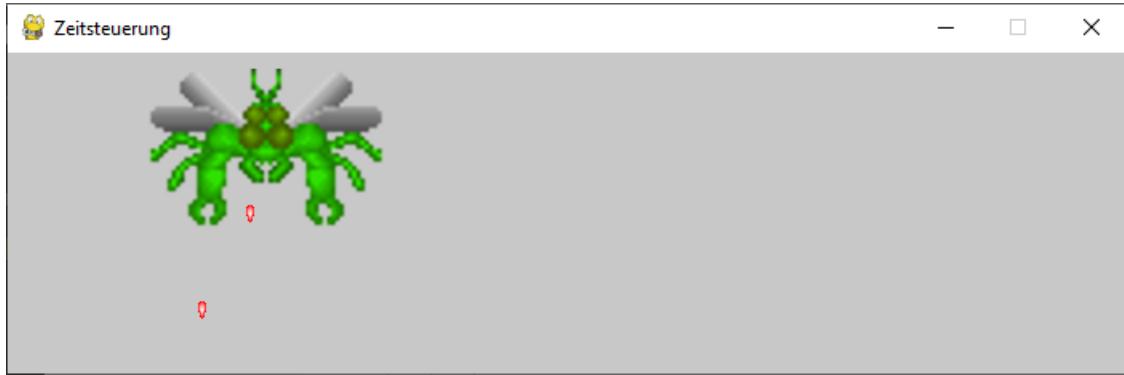


Abbildung 2.38: Feuerball mit Zeitsteuerung

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.

Nun kann man den Quelltext umbauen. Zuerst wird in Zeile 70 die aktuelle Anzahl der *ms* seit Programmstart gemessen und in Zeile 71 wird festgehalten, wie viele *ms* ein Zeitintervall dauern soll; wir wollen alle halbe Sekunde einen Feuerball abwerfen, also 500.

Quelltext 2.69: Zeitsteuerung (9), Version 1.2: Konstruktor von Game

```

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

```

Danach wird in `new_bullet()` abgeprüft, ob das Intervallende erreicht wurde. In Zeile 107 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 107.

Quelltext 2.70: Zeitsteuerung (10), Version 1.2: `new_bullet()` von Game

```

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

```

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.

Timer

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 2.71: 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 86); die beiden Variablen von eben werden nicht mehr gebraucht.

Quelltext 2.72: Zeitsteuerung (12), Version 1.3: Timer-Objekt erzeugen

```

85     self.all_bullets = pygame.sprite.Group()
86     self.bullet_timer = Timer(500) # Timer ohne Verzögerung
87     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 122 abgefragt, ob das Intervallende erreicht wurde und fertig!

Quelltext 2.73: Zeitsteuerung (13), Version 1.3: Timer-Objekt verwenden

```
122     if self.bullet_timer.is_next_stop_reached(): # Wenn Intervallgrenze erreicht
123         self.all_bullets.add(Bullet("shoot.png", self.enemy.sprite.rect.move(0,
20).center))
```

Hinweis: Eine Zeitsteuerung über Ereignisse wird in Kapitel 2.15.3 auf Seite 132 vorgestellt.

Was war neu?

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

Es wurden folgende Pygame-Elemente eingeführt:

- `pygame.time.get_ticks()`:
https://pyga.me/docs/ref/time.html#pygame.time.get_ticks
- `pygame.sprite.Sprite.kill()`:
<https://pyga.me/docs/ref/sprite.html#pygame.sprite.Sprite.kill>

2.11 Animation

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

2.11.1 Die laufende Katze



Abbildung 2.39: Animation einer Katze: Einzelsprites

Die Einzelbilder des Bewegungsbeispiels können Sie in Abbildung 2.39 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 2.74 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 2.74: Animation einer Katze (1), Version 1.0: Präambel, Timer und `Settings`

```

1 import os
2 from time import time
3 from typing import Any
4
5 import pygame
6
7
8 class Settings:
9     WINDOW = pygame.rect.Rect((0, 0), (300, 200))
10    FPS = 60
11    DELTATIME = 1.0 / FPS
12    TITLE = "Animation"
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)
24
25
26 class Timer:
27
28     def __init__(self, duration: int, with_start: bool = True):
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
40
41     def change_duration(self, delta: int = 10):
42         self.duration += delta
43         if self.duration < 0:
44             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 52 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 Überabeparameter einfach nur an das Timer-Objekt weiter. Damit sind eigentlich alle vorbereitenden Aktivitäten abgeschlossen.

Quelltext 2.75: Animation einer Katze (2), Version 1.0: Cat

```

47 class Cat(pygame.sprite.Sprite):
48
49     def __init__(self) -> None:
50         super().__init__()
51         self.images: list[pygame.surface.Surface] = []
52         for i in range(6): # Animations-Sprites laden
53             bitmap = pygame.image.load(Settings.imagepath(f"cat{i}.bmp")).convert()
54             bitmap.set_colorkey("black")
55             self.images.append(bitmap)
56         self.imageindex = 0

```

```

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

```

Die Klasse `CatAnimation` ist nur die übliche Kapselung des Hauptprogramms. In Zeile 85 wird das `Cat`-Objekt erzeugt und in ein `GroupSingle` gestopft.

Quelltext 2.76: Animation einer Katze (3), Version 1.0: Konstruktor und `run()`

```

76 class CatAnimation:
77
78     def __init__(self) -> None:
79         pygame.init()
80         self.window = pygame.Window(size=Settings.WINDOW.size, title=Settings.TITLE)
81         self.screen = self.window.get_surface()
82         self.clock = pygame.time.Clock()
83
84         self.font = pygame.font.Font(pygame.font.get_default_font(), 12)
85         self.cat = pygame.sprite.GroupSingle(Cat()) # Meine Katze
86         self.running = False
87
88     def run(self) -> None:
89         time_previous = time()
90         self.running = True
91         while self.running:
92             self.watch_for_events()
93             self.update()
94             self.draw()
95             self.clock.tick(Settings.FPS)
96             time_current = time()
97             Settings.DELTATIME = time_current - time_previous
98             time_previous = time_current
99         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 2.77: Animation einer Katze (4), Version 1.0: `watch_for_events()`

```

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

```

Der restliche Quelltext (Quelltext 2.78) 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 2.78: Animation einer Katze (5), Version 1.0: `update()` und `draw()`

```

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

```

Wie bei der Zeitsteuerung stört mich, dass die Animationslogik über die Klasse `Cat` verteilt ist, was meiner Ansicht nach ein Verstoß gegen das SRP ist. Bauen wir doch einfach eine Animationsklasse (siehe Quelltext 2.79 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 2.79: 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         return self.images[self.imageindex]
52
53     def is-ended(self) -> bool:
54         if self.endless:
55             return False
56         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 durchnummert 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 2.80: Animation einer Katze (7), Version 1.1: Cat

```

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

```

2.11.2 Der explodierende Felsen

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

Schauen wir uns zuerst die Klasse `Rock` an. In Zeile 84 wird eine Zufallszahl ermittelt, die ich in der darauffolgenden Zeile brauche, um einen von vier möglichen Felsenbitmaps zu laden. Danach werden die Koordinaten des Mittelpunkts des Felsens per Zufallszahlen-generator geraten, wobei ein gewisser Abstand zu den Rändern gewahrt wird. In Zeile 89 wird das `Animation`-Objekt erzeugt. Dabei werden die Dateinamen der Animationsbitmaps wieder in der Reihenfolge der Animation eingelesen. Die Bitmaps können Sie in Abbildung 2.40 auf der nächsten Seite sehen.

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

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

Was hat es mit den drei Zeilen ab Zeile 98 auf sich? Sie dienen rein optischen Zwecken. Die Abmaße der Explosionssprites sind nicht immer gleich und werden durch das `rect`-

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

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.

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

`kill()`

Quelltext 2.81: 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, Settings.WINDOW.width -
89                                              self.rect.width)
89          self.rect.centery = random.randint(self.rect.height, Settings.WINDOW.height -
90                                              self.rect.height)
90          self.anim = Animation([f"explosion0{i}.png" for i in range(1, 5)], False, 50)  #
91          self.timer_lifetime = Timer(random.randint(100, 2000), False)  # Lebenszeit
92          self.bumm = False
93
94      def update(self, *args: Any, **kwargs: Any) -> None:
95          if self.timer_lifetime.is_next_stop_reached():
96              self.bumm = True
97          if self.bumm:
98              self.image = self.anim.next()
99              c = self.rect.center  # Zentrum
100             self.rect = self.image.get_rect()
101             self.rect.center = c
102             if self.anim.isEnded():
103                 self.kill()

```



Abbildung 2.40: 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 114 wird ein `Timer`-Objekt angelegt, welches zwei Felsen pro Sekunde erstellen soll und in Zeile 139 wird dieser abgefragt.

Quelltext 2.82: Animation einer Explosion (2): ExplosionAnimation

```

105  class ExplosionAnimation(object):
106
107      def __init__(self) -> None:
108          pygame.init()
109          self.window = pygame.Window(size=Settings.WINDOW.size, title=Settings.TITLE)
110          self.screen = self.window.get_surface()
111          self.clock = pygame.time.Clock()

```

```

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

```

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.

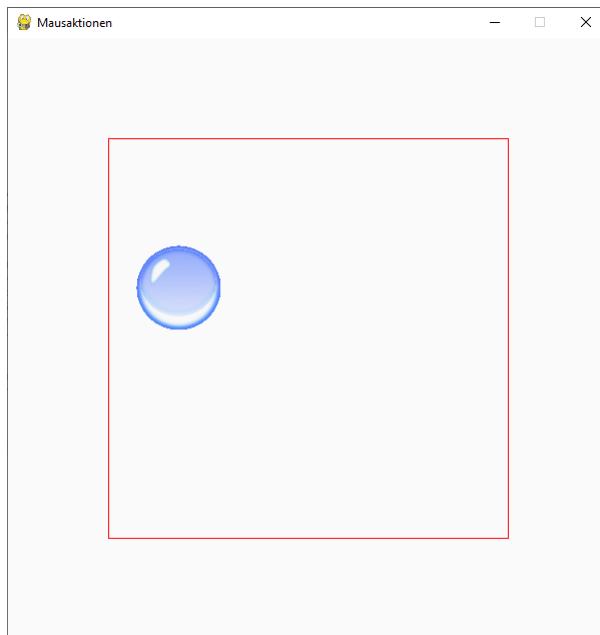


Abbildung 2.41: Mausaktionen

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 2.83: Mausaktionen: Statics und Konstruktor von Game

```

58  class Game:
59      WINDOW = pygame.Rect((0, 0), (600, 600))
60      INNER_RECT = pygame.Rect(100, 100, WINDOW.width - 200, WINDOW.height - 200)
61      FPS = 60
62      DELTATIME = 1.0 / FPS
63
64      def __init__(self) -> None:
65          pygame.init()
66          self.window = pygame.Window(size=Game.WINDOW.size, title="Mausaktionen")
67          self.screen = self.window.get_surface()
68          self.clock = pygame.time.Clock()
69
70          self.ball = Ball() # Ball-Objekt
71          self._running = True

```

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

Quelltext 2.84: 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.FPS)
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.

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.

MOUSE-
BUTTON-
DOWN
MOUSE-
BUTTONUP
event.button
event.pos

Quelltext 2.85: 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: # Maustaste gedrückt
93              if event.button == 1: # left
94                  self.ball.update(rotate=90)
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 107 wird dies durch die Methode `pygame.mouse.set_visible()` erreicht. Diese steuert, ob der Systemmauszeiger – welcher Ausprägung auch immer – angezeigt werden soll oder nicht.

Als Entscheider dient dabei, ob die aktuelle Mausposition innerhalb des inneren Rechtecks liegt. Die Methode `pygame.mouse.get_pos()` liefert mir dazu die aktuelle Mausposition. Diese wird nun einfach in eine schon vorhandene Kollisionsprüfung gesteckt: `pygame.Rect.collidepoint()`. Ist die Mausposition innerhalb des Rechtecks, liefert diese den Wert `True`, ansonsten `False`.

`set_visible()``get_pos()``collidepoint()`Quelltext 2.86: Mausaktionen: `Game.update()` und `Game.draw()`

```

104  def update(self):
105      newpos = pygame.mouse.get_pos()
106      self.ball.update(center=newpos)
107      if Game.INNER_RECT.collidepoint(pygame.mouse.get_pos()): # Unsichtbar?
108          pygame.mouse.set_visible(False)
109      else:
110          pygame.mouse.set_visible(True)
111      self.ball.update(go=True)
112
113  def draw(self) -> None:
114      self.screen.fill((250, 250, 250))
115      pygame.draw.rect(self.screen, "red", Game.INNER_RECT, 1)
116      self.ball.draw(self.screen)
117      self.window.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 50. 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 21, Zeile 32, Zeile 35 oder Zeile 38. 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 2.87: Mausaktionen: Ball

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

```

34     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         self.scale += delta
48         if self.scale > Game.INNER_RECT.width:
49             self.scale = Game.INNER_RECT.width
50         elif self.scale < 5:
51             self.scale = 5
52
53     def set_center(self, center: Tuple[int, int]) -> None:
54         self.rect.center = center

```

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.

rotate()

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.constants`:
<https://pyga.me/docs/ref/locals.html>
- `pygame.MOUSEBUTTONDOWN`, `pygame.MOUSEBUTTONUP`:
<https://pyga.me/docs/ref/event.html>
- Liste der Mausbuttons: Tabelle 2.7 auf der nächsten Seite
- `pygame.mouse.get_pos()`:
https://pyga.me/docs/ref/mouse.html#pygame.mouse.get_pos
- `pygame.mouse.set_visible()`:
https://pyga.me/docs/ref/mouse.html#pygame.mouse.set_visible
- `pygame.Rect.collidepoint()`:
<https://pyga.me/docs/ref/rect.html#pygame.Rect.collidepoint>
- `pygame.transform.rotate()`:
<https://pyga.me/docs/ref/transform.html#pygame.transform.rotate>

Tabelle 2.7: Liste der Mausbuttons

Konstante	Beschreibung
0	nicht definiert
1	linke Maustaste
2	mittlere Maustaste/Mausrad
3	rechte Maustaste
4	Mausrad zu sich drehen (up)
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 2.88: Sound: Präambel und `Settings`

```

1 import os
2 from typing import Any
3
4 import pygame
5 from pygame.constants import (K_ESCAPE, K_MINUS, K_PLUS, KEYDOWN, KEYUP, QUIT,
6                               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:

```

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

`init()`

get_volumeme()

Hintergrundmusik

set_volumeme()
play()

fade

Soundeffekte

Quelltext in Zeile 34 implizit durch `pygame.init()`. In Zeile 41 wird die aktuelle Lautstärke in einem Attribut abgespeichert. Eigentlich ist dies nicht nötig, da man die aktuelle Lautstärke der Hintergrundmusik immer mit `pygame.mixer.music.get_volume()` und die eines Effekts mit `pygame.mixer.Sound.get_volume()` ermitteln kann.

In der Methode `sounds()` sind die vorbereitenden Aktionen zur Soundausgabe gekapselt. Eine Hintergrundmusik wird über `pygame.mixer.music.load()` in den internen Speicher des Mixers geladen. Dadurch wird die Hintergrundmusik aber noch nicht abgespielt. Dies geschieht, nachdem die Lautstärke in Zeile 46 mit `pygame.mixer.music.set_volume()` festgelegt wurde, in der Zeile 47. Die entsprechende Methode `pygame.mixer.music.play()` hat dazu drei Parameter: Der erste Parameter `loops` steuert die Anzahl der Wiederholungen; der Wert `-1` meint dabei, dass die Musik endlos wiederholt wird. Der zweite, `start`, gibt einen Position an, wo die Musik starten soll; der Default ist `0.0`. Soll die Musik leise starten und dann lauter werden (`fade`), kann dies mit dem dritten Parameter `fade` erfolgen; damit können Sie angeben, wie viele Millisekunden dem Lauterwerden zur Verfügung hat; wird nichts angegeben, wird sofort mit der Ziellautstärke gestartet.

Für Soundeffekte wird jeweils ein eigenes `Sound`-Objekt angelegt (Zeile 48f.). 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 2.89: Sound: Konstruktor und `sounds()` von Game

```

30
31 class Game:
32     def __init__(self) -> None:
33         pygame.init()
34         self.window = pygame.Window(size=Settings.WINDOW.size, title='Fingerübung "Sound"')
35         # Auch mixer
36         self.screen = self.window.get_surface()
37         self.clock = pygame.time.Clock()
38
39         self.font_bigsize = pygame.font.Font(pygame.font.get_default_font(), 40)
40         self.running = True
41         self.pause = False
42         self.volume: float = 0.1 # Lautstärke
43         self.sounds()
44
45     def sounds(self) -> None:
46         pygame.mixer.music.load(Settings.get_sound("Lucifer.mid"))
47         pygame.mixer.music.set_volume(self.volume) # Lautstärke
48         pygame.mixer.music.play(-1, 0.0) # Endlos abspielen
49         self.bubble: pygame.mixer.Sound =
50             pygame.mixer.Sound(Settings.get_sound("plopp1.mp3")) #

```

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

Quelltext 2.90: Sound: `watch_for_events()` von Game

```

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

```

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 Fadeln hat; die Vorbelegung ist `0`.

`play()`

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 2.91: Sound: `sound_play()` von Game

```

78
79     def sound_play(self, **kwargs: Any) -> None:
80         if "bubble" in kwargs.keys():
81             self.bubble.play()
82         if "clash" in kwargs.keys():

```

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.

`fadeout()`

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

```

84
85     def music_start_stop(self, **kwargs: Any) -> None:
86         if "fadeout" in kwargs.keys():
87             pygame.mixer.music.fadeout(kwargs["fadeout"])
88         if "loop" in kwargs.keys():

```

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 ?? das Flag `pause` umgelegt.

`pause()`
`unpause()`

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

```

90
91     def pause_alter(self) -> None:
92         if self.pause:
93             pygame.mixer.music.unpause()
94         else:
95             pygame.mixer.music.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.

`set_volume()`

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

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

```

97
98     def volume_alter(self, delta: float) -> None:
99         self.volume += delta
100        if self.volume > 1.0:
101            self.volume = 1.0
102        elif self.volume < 0.0:
103            self.volume = 0.0

```

Und zum Schluss kommt der gute Rest:

Quelltext 2.95: Sound: `draw()`, `update()`, `run()` und Aufruf von Game

```

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

```

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

```

113
114     self.window.flip()
115
116     def update(self):
117         pass
118
119     def run(self):
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         pygame.quit()
127
128
129     def main():
130         Game().run()
131
132
133 if __name__ == "__main__":
134     main()

```

2.13.2 Stereo

Das zweite Beispiel soll die Funktion von Kanälen und **Stereo**effekte ausleuchten. Das Thema ist für eine vollständige Darstellung zu umfangreich, aber ich hoffe, dass dieses Kapitel einen hilfreichen Einstieg bietet.

In Abbildung 2.42 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 untermalt, 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.

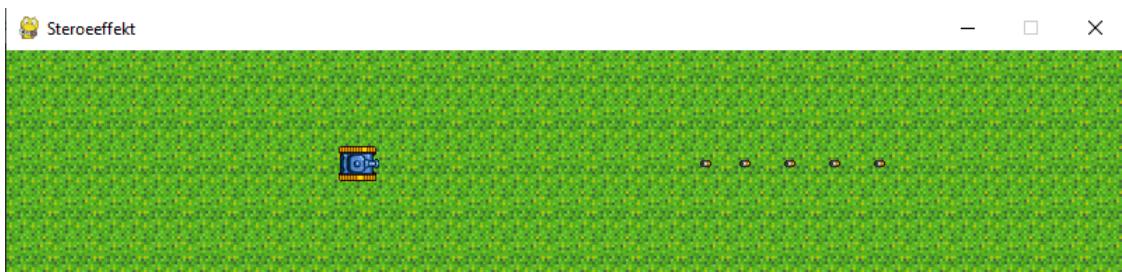


Abbildung 2.42: Sound: Stereoeffekt

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

Quelltext 2.96: Sound-Stereo: Präamble, **Settings** und **Ground**

```

1 import os
2 from time import time
3 from typing import Any
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))

```

Sound-
Objekt

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 Fahrgeräusches in einer Endlosschleife (Zeile 68). Dabei fällt auf, dass hier die Ausgabe nicht über `pygame.mixer.Sound.play()` erfolgt.

Kanal

`find_channel()`

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()`.

`play()`

Quelltext 2.97: Sound-Stereo: Konstruktor von Tank

```

45
46 class Tank(pygame.sprite.Sprite):
47
48     def __init__(self) -> None:
49         super().__init__()
50         self.image_filename = (209, 190, 202, 214, 226, 238, 250, 262)
51         self.images: dict[str, list[pygame.surface.Surface]] = {"up": [], "down": [],
52             "left": [], "right": []}
53         for number in self.image_filename:
54             fullfilename = Settings.get_image(f"tankbrigade_part{number}.png")
55             picture = pygame.image.load(fullfilename).convert()
56             picture.set_colorkey("black")
57             self.images["up"].append(picture)
58             self.images["down"].append(pygame.transform.rotate(picture, 180))
59             self.images["left"].append(pygame.transform.rotate(picture, +90))
60             self.images["right"].append(pygame.transform.rotate(picture, -90))
61         self.direction = "right"
62         self.imageindex = 0
63         self.image = self.images[self.direction][self.imageindex]
64         self.rect = pygame.Rect(self.image.get_rect())
65         self.rect.left, self.rect.top = 3 * self.rect.width, 2 * self.rect.height
66         self.sound_drive = pygame.mixer.Sound(Settings.get_sound("tank_drive1.wav")) #
67         self.channel = pygame.mixer.find_channel() # Sound-Kanal finden
68         self.stereo() #
69         self.channel.play(self.sound_drive, -1) #

```

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

Quelltext 2.98: Sound-Stereo: Tank.update()

```

70
71     def update(self, *args: Any, **kwargs: Any) -> None:
72         if "go" in kwargs.keys():
73             if kwargs["go"]:
74                 self.update_imageindex()
75                 self.image = self.images[self.direction][self.imageindex]
76                 if self.direction == "up" or self.direction == "left":
77                     self.speed = -50
78                 elif self.direction == "down" or self.direction == "right":
79                     self.speed = 50
80                 if self.direction == "up" or self.direction == "down":
81                     self.rect.move_ip(0, self.speed * Settings.DELTATIME)
82                     if self.rect.top <= Settings.WINDOW.top:
83                         self.turn("down")
84                     if self.rect.bottom >= Settings.WINDOW.bottom:
85                         self.turn("up")
86                 elif self.direction == "left" or self.direction == "right":
87                     self.rect.move_ip(self.speed * Settings.DELTATIME, 0)
88                     if self.rect.left <= Settings.WINDOW.left:
89                         self.turn("right")
90                     if self.rect.right >= Settings.WINDOW.right:
91                         self.turn("left")
92             self.stereo()
93         if "turn" in kwargs.keys():

```

Die Methode `stereo()` ist überraschend simpel. Die Methode `pygame.mixer.Channel.set_volume()` stellt nämlich zwei Übergabeparameter zur Verfügung: `left` und `right`.

`set_volume()`

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

```
95
96     def stereo(self) -> None:
97         volume_rechts = self.rect.centerx / Settings.WINDOW.width # 
98         volume_links = 1 - 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 2.100: Sound-Stereo: `Tank.turn()` und `Tank.update_imageindex()`

```
100
101    def turn(self, direction: str) -> None:
102        self.direction = direction
103
104    def update_imageindex(self) -> None:
105        if self.speed == 0:
106            self.imageindex = 0
107        else:
```

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

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

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

Quelltext 2.101: Sound-Stereo: Die Klasse Bullet

```

110
111 class Bullet(pygame.sprite.Sprite):
112
113     SOUND_FIRE = None # Es braucht nur einen
114
115     def __init__(self, tank: Tank) -> None:
116         super().__init__()
117         bulletspeed = 300
118         number: dict[str, int] = {"left": 49, "right": 61, "up": 37, "down": 73}
119         directions = {
120             "left": pygame.Vector2(-bulletspeed, 0),
121             "right": pygame.Vector2(bulletspeed, 0),
122             "up": pygame.Vector2(0, -bulletspeed),
123             "down": pygame.Vector2(0, bulletspeed),
124         }
125         fullfilename = os.path.join(Settings.PATH["image"],
126             f"tankbrigade_part{number[tank.direction]}.png")
127         self.image = pygame.image.load(fullfilename).convert()
128         self.image.set_colorkey("black")
129         self.rect = self.image.get_rect()
130         self.direction = tank.direction
131         self.rect.center = tank.rect.center
132         self.speed = directions[tank.direction]
133
134         if Bullet.SOUND_FIRE == None: # Es braucht nur einen
135             Bullet.SOUND_FIRE = pygame.mixer.Sound(Settings.get_sound("tank_fire1.wav"))
136             volume_rechts = self.rect.centerx / Settings.WINDOW.width
137             volume_links = 1 - volume_rechts
138             self.channel: pygame.mixer.Channel = pygame.mixer.find_channel()
139             self.channel.set_volume(volume_links, volume_rechts)
140             self.channel.play(Bullet.SOUND_FIRE)
141
142     def update(self, *args: Any, **kwargs: Any) -> None:
143         self.rect.move_ip(self.speed * Settings.DELTATIME)
144         if not Settings.WINDOW.contains(self.rect):

```

Der Rest des Quelltextes wird nur der Vollständigkeit wegen abgedruckt.

Quelltext 2.102: Sound-Stereo: Rest

```

146
147 class Game:
148
149     def __init__(self) -> None:
150         pygame.init()
151         self.window = pygame.Window(size=Settings.WINDOW.size, title="Stereoeffekt")
152         self.screen = self.window.get_surface()
153         self.clock = pygame.time.Clock()
154
155         self.ground = pygame.sprite.GroupSingle(Ground())
156         self.tankreference = Tank()
157         self.tank = pygame.sprite.GroupSingle(self.tankreference)
158         self.all_bullets = pygame.sprite.Group()
159         self.running = True
160
161     def watch_for_events(self) -> None:
162         for event in pygame.event.get():
163             if event.type == QUIT:
164                 self.running = False
165             elif event.type == KEYDOWN:
166                 if event.key == K_ESCAPE:
167                     self.running = False

```

```

168         elif event.key == K_UP:
169             self.tank.update(turn="up")
170         elif event.key == K_DOWN:
171             self.tank.sprite.update(turn="down")
172         elif event.key == K_LEFT:
173             self.tank.sprite.update(turn="left")
174         elif event.key == K_RIGHT:
175             self.tank.sprite.update(turn="right")
176         elif event.key == K_SPACE:
177             self.fire()
178
179     def fire(self) -> None:
180         if len(self.all_bullets) < 5:
181             self.all_bullets.add(Bullet(self.tankreference))
182
183     def draw(self) -> None:
184         self.ground.draw(self.screen)
185         self.tank.draw(self.screen)
186         self.all_bullets.draw(self.screen)
187         self.window.flip()
188
189     def update(self) -> None:
190         self.tank.update(go=True)
191         self.all_bullets.update()
192
193     def run(self) -> None:
194         time_previous = time()
195         self.running = True
196         while self.running:
197             self.watch_for_events()
198             self.update()
199             self.draw()
200             self.clock.tick(Settings.FPS)
201             time_current = time()
202             Settings.DELTATIME = time_current - time_previous
203             time_previous = time_current
204             pygame.quit()
205
206
207     def main():
208         Game().run()
209
210
211 if __name__ == "__main__":
212     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://pyga.me/docs/ref/music.html#pygame.mixer.Channel>
- `pygame.mixer.Channel.play()` :
<https://pyga.me/docs/ref/mixer.html#pygame.mixer.Channel.play>

- `pygame.mixer.Channel.set_volume()`:
https://pyga.me/docs/ref/mixer.html#pygame.mixer.Channel.set_volume
- `pygame.mixer.find_channel()` :
https://pyga.me/docs/ref/music.html#pygame.mixer.find_channel
- `pygame.mixer.init()`:
<https://pyga.me/docs/ref/mixer.html#pygame.mixer.init>
- `pygame.mixer.music.fadeout()`:
<https://pyga.me/docs/ref/music.html#pygame.mixer.music.fadeout>
- `pygame.mixer.music.get_volume()`:
https://pyga.me/docs/ref/music.html#pygame.mixer.music.get_volume
- `pygame.mixer.music.load()`:
<https://pyga.me/docs/ref/music.html#pygame.mixer.music.load>
- `pygame.mixer.music.pause()`:
<https://pyga.me/docs/ref/music.html#pygame.mixer.music.pause>
- `pygame.mixer.music.play()`:
<https://pyga.me/docs/ref/music.html#pygame.mixer.music.play>
- `pygame.mixer.music.set_volume()`:
https://pyga.me/docs/ref/music.html#pygame.mixer.music.set_volume
- `pygame.mixer.music.unpause()`:
<https://pyga.me/docs/ref/music.html#pygame.mixer.music.unpause>
- `pygame.mixer.Sound`:
<https://pyga.me/docs/ref/mixer.html#pygame.mixer.Sound>
- `pygame.mixer.Sound.get_volume()`:
https://pyga.me/docs/ref/mixer.html#pygame.mixer.Sound.get_volume
- `pygame.mixer.Sound.play()`:
<https://pyga.me/docs/ref/mixer.html#pygame.mixer.Sound.play>
- `pygame.mixer.Sound.set_volume()`:
https://pyga.me/docs/ref/mixer.html#pygame.mixer.Sound.set_volume

2.14 Dirty Sprites

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

Dirty Sprite
self.dirty

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



Abbildung 2.43: Dirty Sprite – Demo-Spiel

2.14.1 Einfaches Beispiel

Wir haben hier ein kleines, relativ simples Spiel (siehe Abbildung 2.43) 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 wegzuklicken.

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

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

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

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

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

Farbnamen

Quelltext 2.103: 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 2.104: Dirty Sprite – `Game` (unverändertes Demo)

```

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

```

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 125 werden die Bedeutungen angegeben.

`DirtySprite`

`self.dirty`

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

In unserem Beispiel wird an zwei Stellen das Aussehen verändert. Zum einen, wenn das Signal 'switch' gesendet wird (siehe Zeile 29) und zum anderen, wenn der interne Timer die Farbe wieder von rot nach weiß abändert (siehe Zeile 35). Wie in Tabelle 2.9 auf Seite 125 beschrieben, wird der Wert automatisch nach der Ausgabe wieder auf 0 gesetzt.

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

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

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

Quelltext 2.105: 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

```

LayeredDirty

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.

Quelltext 2.106: Dirty Sprite – Konstruktor von `Game` (Umbau)

```

38 class Game:
39     def __init__(self) -> None:
40         pygame.init()
41         self.screen = pygame.display.set_mode(Settings.WINDOW.size)
42         pygame.display.set_caption("DirtySpritesDemo")
43         self.clock = pygame.time.Clock()
44         self.background_image = pygame.image.load(Settings.get_image("background.png"))
45         self.background_image = pygame.transform.scale(self.background_image,
46             (Settings.WINDOW.size))
47         self.all_tiles = pygame.sprite.LayeredDirty() # Gruppenklasse für DirtySprite
48         self.create_playground()
49         self.timer = Timer(1000, False)
50         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 blitzten 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 2.107: 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`.

`clear()`

Quelltext 2.108: 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 2.109: Dirty Sprite – `Game.draw()` (Umbau)

```

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

```

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

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

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 2.110: Performancevergleich – Messung

```

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

```

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

```

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

```

Dann habe ich Testläufe mit den Parametern aus Tabelle 2.8 auf einem Schul-PC laufen lassen. Die Werte waren unterschiedlich⁶, aber die Tendenz immer die gleiche. Ich habe mal ein Ergebnis in einer Grafik visualisiert (siehe Abbildung 2.44). 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

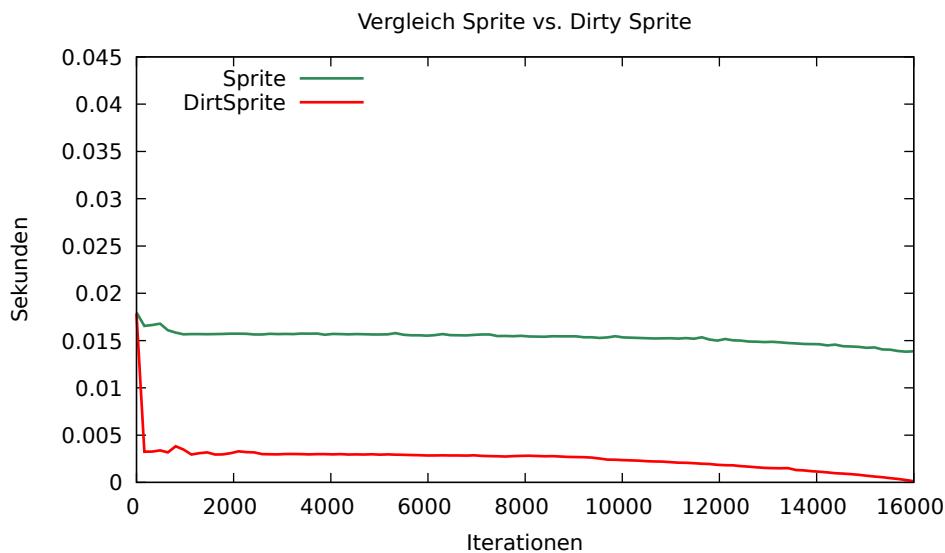


Abbildung 2.44: Performancevergleich mit Testkonfiguration 2

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 2.45 auf der nächsten Seite raus. Natürlich fieberhaft nach einem Fehler gesucht, schließlich hatte ich ja Kleinigkeiten verändert. Test auf einem Surface Pro wiederholt

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

und schon wieder recht schlechte Ergebnisse (Abbildung 2.46). Hier war es sogar noch verwirrender, da die ersten rund 8.000 Iterationen einen Vorteil für DirtySprites ergaben und danach die Zeitverbräuche sich angleichten.

Nach vielen Wiederholungstest auf den drei Rechnern, kam mir der Gedanke, dass der PC zu Hause und der Surface Pro vielleicht nicht die Framerate von 60 *fps* schaffen und daher das Zeitfenster zu klein ist. Also den Test mit kleinerer Framerate wiederholt und siehe da, dann waren die Ergebnisse wieder eindeutig (siehe Abbildung 2.47). Ein Performancetest zwischen den einzelnen Rechnern bestätigte im Nachgang diese Vermutung. In Abbildung 2.48 sehen Sie, das der Schul-PC definitiv die beste Grafikverarbeitungsgeschwindigkeit hat.

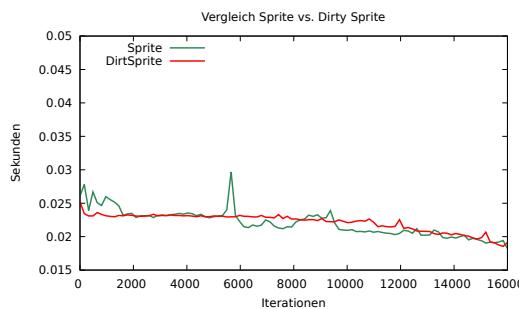


Abbildung 2.45: Testkonfiguration mit privatem PC 2

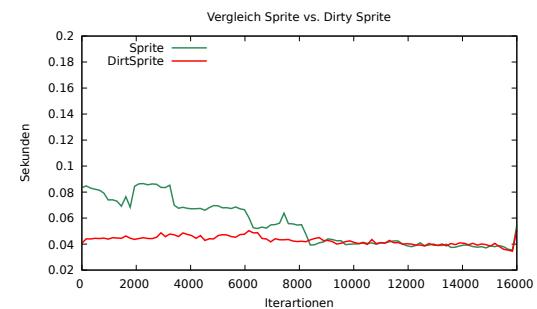


Abbildung 2.46: Testkonfiguration mit Surface Pro 2

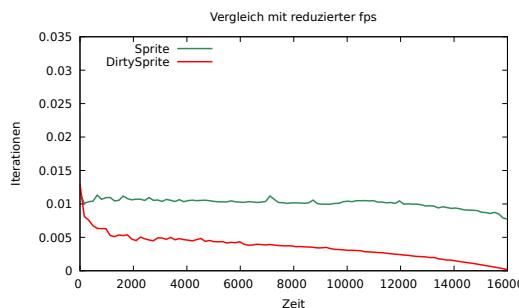


Abbildung 2.47: Testkonfiguration mit privatem PC und reduzierter fps 2

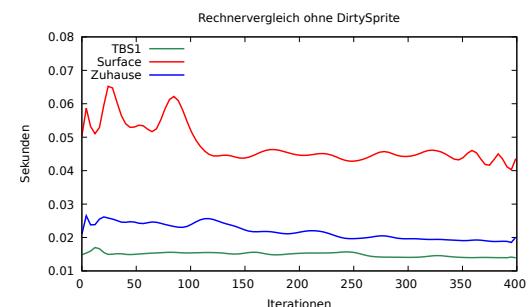


Abbildung 2.48: Bestätigung der Unterschiede

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

2.14.3 Fazit

Die obigen Messungen und Probleme bzgl. sich bewegender Sprites haben mich persönlich nicht von dieser DirtySprite-Implementierung überzeugt. Ein einfacher Ball, der an den Wänden abprallt (Quelltext 2.111), hinterlässt Artefakte oder flackert recht heftig, was ich nicht erwartet hätte.

Quelltext 2.111: Einfacher Ball mit DirtsSprite

```

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

```

```

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

```

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

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

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

Was war neu?

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

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

Es wurden folgende Pygame-Elemente eingeführt:

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

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

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

Tabelle 2.9: Bedeutung von `dirty`

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

2.15 Events

Wir haben Ereignisse ([Event](#)) schon an zwei Stellen verwendet, ohne sie näher betrachtet zu haben. Zum einen als wir in Kapitel [2.6](#) auf Seite [53](#) über die Tastatur und zum anderen als wir in Kapitel [2.12](#) auf Seite [97](#) über die Maus gesprochen haben.

Wir werden hier drei Aspekte näher beleuchten:

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

2.15.1 Welche Infos stecken in einem Event?

Das Programm zu Quelltext [2.112](#) erstellt lediglich ein graues Fenster und gibt mit `print` in Zeile [36](#) das Event in der Konsole aus.

Quelltext 2.112: Events – Informationen ausgeben

```

34     def watch_for_events(self) -> None:
35         for event in pygame.event.get():
36             print(event) # Eventinfo ausgeben
37             if event.type == QUIT:
38                 self.running = False
39             elif event.type == KEYDOWN:
40                 if event.key == K_ESCAPE:
41                     self.running = False

```

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

Quelltext 2.113: Events – Konsolenausgabe

```

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

```

```

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

```

Zunächst fällt auf, dass die Eventinformationen in Form eines Dictionarys zur Verfügung gestellt werden. Den ersten Eintrag (die Nummer mit dem Bindestrich und anschließendem Namen) kann über `event.type` abgefragt werden. Damit man sich diese Nummern nicht auswendig merken muss, werden von Pygame entsprechende Konstanten angeboten; für die Tastatur finden Sie in Tabelle 2.5 auf Seite 55 und für die Maus in Tabelle 2.7 auf Seite 102 eine Übersicht.

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

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

2.15.2 Wie kann ich selbst ein Event erzeugen?

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

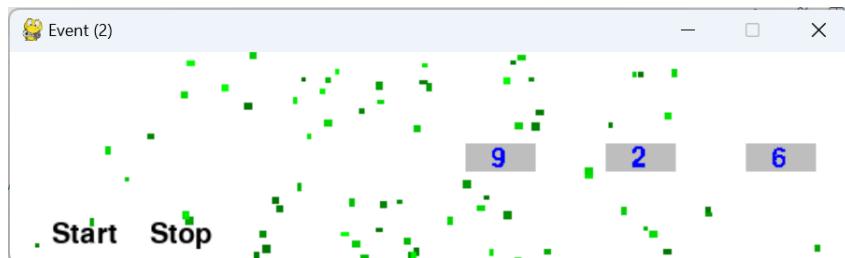


Abbildung 2.49: Selbst erstellte Events

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

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

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

Das Ganze nun im Detail. In der Konsolenausgabe oben (siehe Seite 126) ist für jedes Event so eine eindeutige Nummer zu sehen, über die man das Event identifizieren kann. Pygame reserviert mir einen Nummernbereich für eigene Events zwischen den Konstanten `pygame.USEREVENT` und `pygame.NUMEVENTS - 1`. Für jedes selbst erstellte Event muss man nun eine solche eindeutige Nummer vergeben. Am einfachsten ist es, zentral diese durch `USEREVENT + n` zu definieren. In Zeile 11 und Zeile 12 finden Sie entsprechende Beispiele. Ich kapsle diese Definitionen in eine statische Klasse aus keinem anderen Grund, als dass ich dann die Auto vervollständigung des Editors gut nutzen kann (Zeile 10). Die Klasse `Settings` sollte selbsterklärend sein.

Quelltext 2.114: Events (2) – Präambel

```

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

```

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

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

Quelltext 2.115: Events (2) – Klasse Button

```

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

```

```

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

```

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

Quelltext 2.116: Events (2) – Klasse Particle

```

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

```

Mit `Box` wird so eine Ziffernbox implementiert. Dem Konstruktor wird dabei eine Position und ein Index mitgegeben. Die Bedeutung des Paramters `position` sollte klar sein. Mit

Hilfe von `index` kann später ermittelt werden, welche Box einen Überlauf zur nächst höheren Zehnerpotenz hatte.

In `update()` wird der internen Zähler `_count` immer um 1 erhöht. Erreiche ich dabei die 10 (Zeile 98), wird das Event erzeugt und der Index als Eventinfo übergeben. Damit kann das Hauptprogramm ermitteln, welcher Box er nun ein `update()` verpassen muss.

Quelltext 2.117: Events (2) – Klasse Box

```

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

```

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

Quelltext 2.118: Events (2) – Konstruktor von Game

```

111  class Game:
112
113      def __init__(self) -> None:
114          pygame.init()
115          self.window = pygame.Window(size=Settings.WINDOW.size, title="Event_(1)",
116                                      position=WINDOWPOS_CENTERED)
116          self.screen = self.window.get_surface()
117          self.clock = pygame.time.Clock()
118          self.running = True
119          self.all_sprites = pygame.sprite.Group()
120          self.all_particles = pygame.sprite.Group()
121          self.generate_particles(Settings.STARTNOFPARTICLES)
122          self.all_buttons = pygame.sprite.Group()
123          self.all_buttons.add(Button("Start", (30, Settings.WINDOW.bottom - 30),
124                                      self.all_sprites))
124          self.all_buttons.add(Button("Stop", (100, Settings.WINDOW.bottom - 30),
125                                      self.all_sprites))
125          self.all_boxes = pygame.sprite.Group()
126          self.generate_boxes(Settings.NOFBOXES)

```

127

```
    self.running = True
```

Die Methode `run()` ist nahezu langweilig.

Quelltext 2.119: Events (2) – `Game.run()`

```
129  def run(self) -> None:
130      time_previous = time()
131      while self.running:
132          self.watch_for_events()
133          self.update()
134          self.draw()
135          self.clock.tick(Settings.FPS)
136          time_current = time()
137          Settings.DELTATIME = time_current - time_previous
138          time_previous = time_current
139      pygame.quit()
```

Benutzerdefinierte Events werden genauso behandelt wie vordefinierte. Zuerst erfragt man den `type` und dann verarbeitet man die Eventinfos. In Zeile 151 wird nachgeschaut, ob einer der beiden Buttons gedrückt wird. Anschließend wird über die Eventinfo `text` an die Partikel die Nachricht weitergeleitet, ob sie stehenbleiben oder weiterlaufen sollen. Analoges ab Zeile 153. Zuerst wird überprüft, ob eine Box einen Überlauf hatte und dann wird mit Hilfe des Eventinfo `index` die nächste Box darüber informiert, dass sie sich um 1 erhöhen muss.

Quelltext 2.120: Events (2) – `Game.watch_for_events()`

```
141  def watch_for_events(self) -> None:
142      for event in pygame.event.get():
143          if event.type == QUIT:
144              self.running = False
145          elif event.type == KEYDOWN:
146              if event.key == K_ESCAPE:
147                  self.running = False
148          elif event.type == MOUSEBUTTONDOWN:
149              if event.button == 1:
150                  self._check_button_pressed(event.pos)
151          elif event.type == MyEvents.BUTTONPRESSED: #
152              self.all_particles.update(action=event.text)
153          elif event.type == MyEvents.OVERFLOW: #
154              if event.index < Settings.NOFBOXES - 1:
155                  self.all_boxes.sprites()[event.index + 1].update(counter="inc")
```

Der Rest wird hier der Vollständigkeit abgedruckt.

Quelltext 2.121: Events (2) – Der Rest von `Game`

```
157  def update(self):
158      self.all_buttons.update()
159      self.all_particles.update(action="move")
160      self._check_boxcollision()
161
162  def draw(self) -> None:
163      self.screen.fill("white")
164      self.all_sprites.draw(self.screen)
```

```

165         self.window.flip()
166
167     def generate_boxes(self, number: int) -> None:
168         for i in range(number):
169             self.all_boxes.add(Box(i, (Settings.WINDOW.right - 50 - i * 100,
170                                     Settings.WINDOW.centery), self.all_sprites))
171
172     def generate_particles(self, number: int) -> None:
173         for i in range(number):
174             self.all_particles.add(Particle(self.all_sprites))
175
176     def _check_button_pressed(self, position: Tuple[int]) -> None:
177         for b in self.all_buttons.sprites():
178             if b.rect.collidepoint(position):
179                 b.update(action="pressed")
180
181     def _check_boxcollision(self) -> None:
182         for p in self.all_particles.sprites():
183             for b in self.all_boxes.sprites():
184                 if p.rect.colliderect(b):
185                     self.all_boxes.sprites()[0].update(counter="inc")
186                     p.kill()

```

2.15.3 Wie kann ich periodisch Ereignisse erzeugen lassen?

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

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

Quelltext 2.122: Events (3) – Präambel

```

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

```

Im Konstruktor von **Game** wird in Zeile 141 mit Hilfe von `pygame.time.set_timer()` ein periodischer Timer dazu gesetzt. Dieser schießt alle *500 ms* die entsprechende EventID ab.

Quelltext 2.123: Events (3) – Timer

```

140         self.generate_boxes(Settings.NOFBOXES)
141         pygame.time.set_timer(MyEvents.NEWPARTICLES, 500) # Periodischer Timer
142         self.running = True

```

Wie die anderen Events wird dieses nun in `watch_for_event()` abgefangen (Zeile 171) und verarbeitet. Hier indem die Methode `generate_particles()` aufgerufen wird.

Quelltext 2.124: Events (3) – Event

```

171     elif event.type == MyEvents.NEWPARTICLES: # Periodisches Event
172         self.generate_particles(Settings.NEWNOFPARTICLES)

```

Was war neu?

Der Vorteil von benutzerdefinierten Ereignissen wird hier gut deutlich. Wollte man dies anders implementieren, müssen die Objekte von einander wissen. Alle Boxen müssten ihren Vorgänger oder Nachfolger beispielsweise als Referenz kennen, um den Überlauf bekannt zu geben. Dies kann auch eine gute Methode sein, durch ein Event werden die Klassen aber entkoppelt und das Hauptprogramm kann die Informationsweiterleitung durch die Eventinfo gesteuert organisierten.

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

Es wurden folgende Pygame-Elemente eingeführt:

- USEREVENT :
<https://pyga.me/docs/ref/event.html#pygame.event>
- NUMEVENTS :
<https://pyga.me/docs/ref/event.html#pygame.event>
- pygame.event.Event:
<https://pyga.me/docs/ref/event.html#pygame.event.Event>
- pygame.event.get() :
<https://pyga.me/docs/ref/event.html#pygame.event.get>
- pygame.event.post():
<https://pyga.me/docs/ref/event.html#pygame.event.post>
- pygame.time.set_timer():
https://pyga.me/docs/ref/time.html#pygame.time.set_timer
- pygame.WINDOWPOS_CENTERED:
<https://pyga.me/docs/ref/window.html#pygame.Window.position>

3 Beispielprojekte

3.1 Pong

Der Anfängerklassiker überhaupt. Seit 1972 wird dieses Spiel in immer wieder neuen Varianten gespielt. Da die Regeln recht einfach sind, eignet es sich gut als Anfängerprojekt. Wir werden dieses Spiel systematisch Schritt für Schritt entwickeln, wobei ich davon ausgehen werde, dass die Techniken aus Kapitel 2 bekannt sind. Ich werde auf Docstring-Kommentare im Quelltext verzichten, da hier im Text alles erklärt wird und die Listings sich dadurch unnötig verlängern. In der finalen Version sind sie eingetragen.

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

3.1.1 Requirement 1: Standards

Requirement 1 Standardfunktionalität

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

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

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

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

Der Background wird hier nicht als Bitmap geladen, sondern erzeugt. Es gibt dafür keinen besonderen Grund, außer zu demonstrieren, dass Bitmaps auch dynamisch erstellt werden können. Dafür wird als erstes ein `Surface`-Objekt in der Größe des Bildschirms erstellt. Dann wird es dunkelrot ausgefüllt, was einen Sandplatz simulieren soll. In `paint_net()` ab Zeile 26 wird das Netz als eine Folge von weißen Rechtecken gemalt.

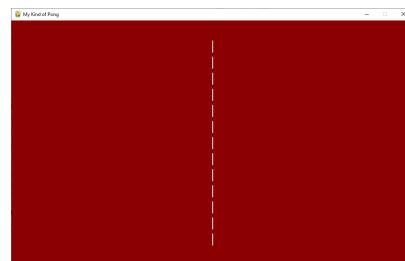


Abb. 3.1: Pong: der Hintergrund

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

```

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

```

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

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

```

31 class Game:
32     def __init__(self):
33         self.window = pygame.Window(size=Settings.WINDOW.size, title="My Kind of Pong",
34                                     position=pygame.WINDOWPOS_CENTERED)
35         self.screen = self.window.get_surface()
36         self.clock = pygame.time.Clock()
37         self.background = pygame.sprite.GroupSingle(Background())
38         self.running = True
39
40     def run(self):
41         time_previous = time()
42         while self.running:
43             self.watch_for_events()
44             self.update()
45             self.draw()
46             self.clock.tick(Settings.FPS)
47             time_current = time()
48             Settings.DELTATIME = time_current - time_previous
49             time_previous = time_current
50         pygame.quit()

```

```

50
51     def update(self):
52         pass
53
54     def draw(self):
55         self.background.draw(self.screen)
56         self.window.flip()
57
58     def watch_for_events(self):
59         for event in pygame.event.get():
60             if event.type == pygame.QUIT:
61                 self.running = False
62             elif event.type == pygame.KEYDOWN:
63                 if event.key == pygame.K_ESCAPE:
64                     self.running = False

```

Der Vollständigkeit halber:

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

```

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

```

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

3.1.2 Requirement 2: Die Schläger

Requirement 2 Schläger

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

In Zeile 37 wird die Größe des Schlägers ermittelt (Requirements 2.1 und 2.2). Ab Zeile 38 wird die Position der Schläger festgelegt. Die vertikale Startposition ist dabei immer

die Bildschirmmitte. Die horizontale Startposition ist davon abhängig, ob es der rechte oder linke Schläger ist. Beide werden entsprechend Requirement 2.4 etwas vom Rand abgesetzt.

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

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

```

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

```

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

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

```

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

```

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

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

```

57     def move(self) -> None:
58         if self.direction != Paddle.DIRECTION["halt"]:
59             self.rect.move_ip(0, self.speed * self.direction * Settings.DELTATIME) #
60             if self.direction == Paddle.DIRECTION["up"]:
61                 self.rect.top = max(self.rect.top, Paddle.BORDERDISTANCE["vertical"])
62             elif self.direction == Paddle.DIRECTION["down"]:
63                 self.rect.bottom = min(self.rect.bottom, Settings.WINDOW.height -
64                                     Paddle.BORDERDISTANCE["vertical"])

```

Nun müssen die Schläger in Game eingepflegt werden. In Zeile 73 wird zunächst eine Spritegroup erstellt, welche alle Sprites außer dem Hintergrund aufnehmen wird. Danach werden die beiden Schläger erzeugt und per Übergabeparameter gleich der Spritegroup hinzugefügt.

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

```

66 class Game:
67     def __init__(self):
68         self.window = pygame.Window(size=Settings.WINDOW.size, title="My Kind of Pong",
69             position=pygame.WINDOWPOS_CENTERED)
70         self.screen = self.window.get_surface()
71         self.clock = pygame.time.Clock()
72         self.background = pygame.sprite.GroupSingle(Background())
73         self.all_sprites = pygame.sprite.Group()
74         self.paddle = {} # Schläger
75         self.paddle["left"] = Paddle("left", self.all_sprites)
76         self.paddle["right"] = Paddle("right", self.all_sprites)
77         self.running = True

```

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

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

```

90     def update(self):
91         self.all_sprites.update(action="move") # Bewegung
92
93     def draw(self):
94         self.background.draw(self.screen)
95         self.all_sprites.draw(self.screen) # Ausgabe
96         self.window.flip()

```

Und jetzt werden die Tastaturevents verarbeitet. Das Drücken einer Taste löst eine Bewegung aus (ab Zeile 105) und das Loslassen führt zu einem Anhalten des entsprechenden Schlägers (ab Zeile 113).

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

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

```

98     def watch_for_events(self):
99         for event in pygame.event.get():
100             if event.type == pygame.QUIT:
101                 self.running = False
102             elif event.type == pygame.KEYDOWN:
103                 if event.key == pygame.K_ESCAPE:
104                     self.running = False
105                 elif event.key == pygame.K_UP: # Schlägerbewegung
106                     self.paddle["right"].update(action="up")
107                 elif event.key == pygame.K_DOWN:
108                     self.paddle["right"].update(action="down")
109                 elif event.key == pygame.K_w:
110                     self.paddle["left"].update(action="up")

```

```

111         elif event.key == pygame.K_s:
112             self.paddle["left"].update(action="down")
113         elif event.type == pygame.KEYUP: # Schlägerstopp
114             if event.key == pygame.K_UP or event.key == pygame.K_DOWN:
115                 self.paddle["right"].update(action="halt")
116             elif event.key == pygame.K_w or event.key == pygame.K_s:
117                 self.paddle["left"].update(action="halt")

```

3.1.3 Requirement 3: Der Ball

Requirement 3 Ball

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

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

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

```

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

```

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

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

```

68  class Ball(pygame.sprite.Sprite):
69      def __init__(self, *groups: Tuple[pygame.sprite.Group]) -> None:
70          super().__init__(*groups)
71          self.rect = pygame.Rect(0, 0, 20, 20) # Größe
72          self.image = pygame.surface.Surface(self.rect.size).convert()
73          self.image.set_colorkey("black")
74          pygame.draw.circle(self.image, "green", self.rect.center, self.rect.width // 2)
75          self.speed = Settings.WINDOW.width // 3 # Geschwindigkeit
76          self.speedxy = pygame.Vector2()
77          self.service() # Aufschlag

```

In `update()` werden die Aufgaben verteilt.

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

```

79  def update(self, *args: Any, **kwargs: Any) -> None:
80      if "action" in kwargs.keys():
81          if kwargs["action"] == "move":
82              self.move()
83          elif kwargs["action"] == "service":
84              self.service()
85      return super().update(*args, **kwargs)

```

Schauen wir uns jetzt die Hilfsfunktionen im einzelnen an. Beginnen wir mit `move()`. Wie zu erwarten, werden die Positionsangabe mit Hilfe der Geschwindigkeiten aktualisiert. Danach wird ab Zeile 89 überprüft, ob der Ball eine der vier Ränder erreicht hat.

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

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

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

```

87  def move(self) -> None:
88      self.rect.move_ip(self.speedxy * Settings.DELTATIME)
89      if self.rect.top <= 0: # Rändercheck
90          self.vertical_flip()
91          self.rect.top = 0
92      elif self.rect.bottom >= Settings.WINDOW.bottom:
93          self.vertical_flip()
94          self.rect.bottom = Settings.WINDOW.bottom
95      elif self.rect.right < 0:
96          Settings.POINTS[1] += 1
97          self.service()
98      elif self.rect.left > Settings.WINDOW.right:
99          Settings.POINTS[0] += 1
100         self.service()

```

Beim Aufschlag wird das Zentrum des Balls auf das Zentrum des Bildschirms gesetzt (Requirement 3.3). Danach werden für die beiden Richtungsgeschwindigkeiten per Zufall die Vorzeichen und damit die Bewegungsrichtung (nach links oder rechts bzw. nach oben oder unten) bestimmt. Da wir noch keine Punkteausgabe haben, ist in Zeile 105 eine provisorische Ausgabe auf der Konsole programmiert.

Quelltext 3.15: Pong (Requirement 3) – `Ball.service()`

```

102 def service(self) -> None:
103     self.rect.center = Settings.WINDOW.center
104     self.speedxy = pygame.Vector2(choice([-1, 1]), choice([-1, 1])) * self.speed
105     print(Settings.POINTS) # Provisorium

```

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

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

```
107     def horizontal_flip(self) -> None:
108         self.speedxy.x *= -1
109
110     def vertical_flip(self) -> None:
111         self.speedxy.y *= -1
```

3.1.4 Requirement 4: Punkte

Requirement 4 Punkte

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

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

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

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

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

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

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

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

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

```

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

```

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

Quelltext 3.20: Pong (Requirement 4) – MyEvent

```

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

```

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

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

```

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

```

Jetzt muss nur noch in `watch_for_events()` das Userevent abgegriffen werden (ab Zeile 198).

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

```

178     def watch_for_events(self):
179         for event in pygame.event.get():
180             if event.type == pygame.QUIT:
181                 self.running = False
182             elif event.type == pygame.KEYDOWN:
183                 if event.key == pygame.K_ESCAPE:
184                     self.running = False
185                 elif event.key == pygame.K_UP:
186                     self.paddle["right"].update(action="up")

```

```

187         elif event.key == pygame.K_DOWN:
188             self.paddle["right"].update(action="down")
189         elif event.key == pygame.K_w:
190             self.paddle["left"].update(action="up")
191         elif event.key == pygame.K_s:
192             self.paddle["left"].update(action="down")
193     elif event.type == pygame.KEYUP:
194         if event.key == pygame.K_UP or event.key == pygame.K_DOWN:
195             self.paddle["right"].update(action="halt")
196         elif event.key == pygame.K_w or event.key == pygame.K_s:
197             self.paddle["left"].update(action="halt")
198     elif event.type == MyEvents.POINT_FOR: # Userevent
199         self.score.update(player=event.player)

```

3.1.5 Requirement 5: Tennisschlag

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

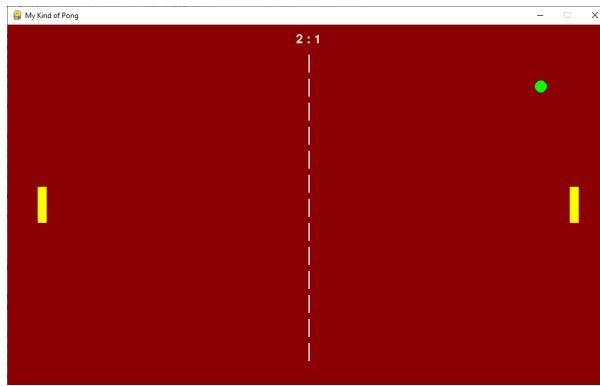


Abbildung 3.2: Pong: mit Schläger, Ball und Punktstand

Requirement 5 Punkte

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

In Game bauen wir dazu die Methode `check_collision()`, welche überprüft, ob der Ball einen Schläger getroffen hat. Es bietet sich an, dazu die Methode `pygame.sprite.collide_rect()` zu verwenden. Wenn eine Kollision vorliegt, wird die bisher noch nicht verwendete Methode `horizontal_flip()` (siehe Quelltext 3.16 auf Seite 141) über `update()` ausgeführt. Danach werden die Ränder wieder so verschoben, dass Ball und Schläger sich nicht überlappen. Ebenso wird über `update()` die Methode `respeed()` aufgerufen, so dass Requirement 5.2 erfüllt wird.

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

```

207     def check_collision(self):
208         if pygame.sprite.collide_rect(self.ball, self.paddle["left"]):
209             self.ball.update(action="hflip")
210             self.ball.rect.left = self.paddle["left"].rect.right + 1
211         elif pygame.sprite.collide_rect(self.ball, self.paddle["right"]):
212             self.ball.update(action="hflip")
213             self.ball.rect.right = self.paddle["right"].rect.left - 1

```

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

Quelltext 3.24: Pong (Requirement 5) – Ball.respeed()

```

123     def respeed(self) -> None:
124         self.speedxy.x += randrange(0, self.speed // 4)
125         self.speedxy.y += randrange(0, self.speed // 4)

```

3.1.6 Requirement 6: Computerspieler

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

Requirement 6 Punkte

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

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

Quelltext 3.25: Pong (Requirement 6) – Settings

```

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

```

In der Methode `update()` wird ab Zeile 196 mit Hilfe der Flags überprüft, ob der Schläger vom Computer gespielt wird und wenn *Ja*, wird eine Controler-Methode aufgerufen.

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

```

194     def update(self):
195         self.check_collision()
196         for i in Settings.KI.keys(): # Computerbefehl
197             if Settings.KI[i]:
198                 self.paddlecontroller(self.paddle[i])
199         self.all_sprites.update(action="move")

```

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

Quelltext 3.27: Pong (Requirement 6) – Game.paddlecontroller()

```

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

```

In `watch_for_events()` sind umfangreiche Umbauten notwendig. Zunächst muss die manuelle Steuerung für die Schläger unterbunden werden, wenn die auf Computerspieler stehen. Dazu wird vor Aufruf der entsprechenden `update()`-Methode zuerst gefragt, ob nicht der Computerspieler die Kontrolle hat. Ein Beispiel finden Sie in Zeile 214.

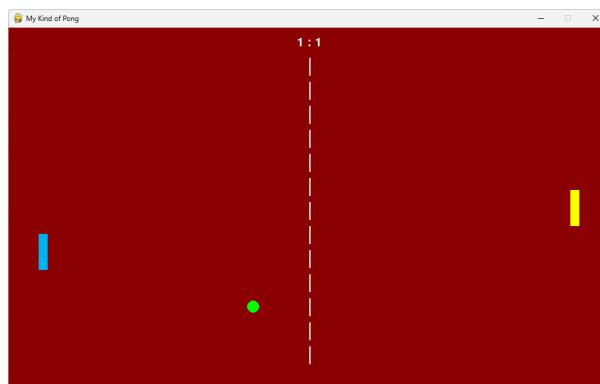


Abbildung 3.3: Pong: Schlägerfarbe markiert KI-Modus (links KI, rechts manuell)

Ein verbleibender Punkt ist noch Requirement 6.3. Dazu wird wie in Zeile 227 das entsprechende Flag abgefragt und dem Schläger das Halt-Signal gesendet.

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

```

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

```

3.1.7 Requirement 7: Sound

Ein wenig Sound könnte das Spiel noch aufpeppen.

Requirement 7 Sound

1. *Der Schlag mit dem Tennisschläger soll mit einem passendem Sound untermauert werden.*
2. *Das Abprallen vom oberen und unterem Rand soll mit einem passendem Sound untermauert werden.*
3. *Der Sound soll über $F2$ ein- bzw. ausgeschaltet werden können.*

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

Quelltext 3.29: Pong (Requirement 7) – Settings

```

9  class Settings:
10     WINDOW = pygame.rect.Rect(0, 0, 1000, 600)
11     FPS = 60
12     DELTATIME = 1.0 / FPS
13     KI = {"left": False, "right": False}
14     SOUNDFLAG = True  # Soundflags
15     PATH = {}
16     PATH["file"] = os.path.dirname(os.path.abspath(__file__))
17     PATH["sound"] = os.path.join(PATH["file"], "sounds")
18
19     @staticmethod
20     def get_sound(filename: str) -> str:
21         return os.path.join(Settings.PATH["sound"], filename)

```

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

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

```

100    class Ball(pygame.sprite.Sprite):
101        def __init__(self, *groups: Tuple[pygame.sprite.Group]) -> None:
102            super().__init__(*groups)
103            self.sounds: dict[str, pygame.mixer.Sound] = {}  # Geräusche speichern
104            self.sounds["left"] = pygame.mixer.Sound(Settings.get_sound("playerl.mp3"))
105            self.sounds["right"] = pygame.mixer.Sound(Settings.get_sound("playerr.mp3"))
106            self.sounds["bounce"] = pygame.mixer.Sound(Settings.get_sound("bounce.mp3"))
107            self.channel = pygame.mixer.find_channel()
108            self.rect = pygame.rect.FRect(0, 0, 20, 20)
109            self.image = pygame.surface.Surface(self.rect.size).convert()
110            self.image.set_colorkey("black")
111            pygame.draw.circle(self.image, "green", self.rect.center, self.rect.width // 2)
112            self.speed = Settings.WINDOW.width // 3
113            self.speedxy = pygame.Vector2()
114            self.service()

```

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

Quelltext 3.31: Pong (Requirement 7) – `Ball.horizontal_flip()`

```

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

```

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

Quelltext 3.32: Pong (Requirement 7) – `Ball.vertical_flip()`

```

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

```

Verbleibt noch das Ein- bzw. Ausschalten der Soundausgabe in `watch_for_events()` in Zeile 245 mit Hilfe der Funktionstaste `F2`.

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

```

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

```

3.1.8 Requirement 8: Pause und Hilfebildschirm

Requirement 8 Punkte

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

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

Quelltext 3.34: Pong (Requirement 8) – Pause

```

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

```

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

Quelltext 3.35: Pong (Requirement 8) – Help

```

55 class Help(pygame.sprite.Sprite):
56     def __init__(self, *groups: Tuple[pygame.sprite.Group]) -> None:
57         super().__init__(*groups)
58         self.rect = pygame.Rect(Settings.WINDOW.topleft, Settings.WINDOW.size)
59         self.image = pygame.surface.Surface(self.rect.size).convert_alpha()
60         self.image.fill([20, 20, 20, 200]) # transparentes Grau
61         font = pygame.font.Font(pygame.font.get_default_font(), 20)
62         text_l = "h\nESC\nF2\nnk\nnr\nUP\nDOWN\nnw\nns"
63         text_r = "-toggle help modus\n-toggle pause modus\n-quit\n\n-toggle sound modus\n"
64         text_r += "-toggle both paddles KI modus\n-toggle left paddle KI modus\n-toggle right paddle KI modus\n"
65         text_r += "-left paddle move up\n-left paddle move down\n-right paddle move up\n-right paddle move down"
66         lines = font.render(text_l, True, "white")
67         self.image.blit(lines, (10, 10))
68         lines = font.render(text_r, True, "white")
69         self.image.blit(lines, (10 + 70, 10))

```

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

Quelltext 3.36: Pong (Requirement 8) – Help

```

218 class Game:
219     def __init__(self):
220         pygame.init()
221         self.window = pygame.Window(size=Settings.WINDOW.size, title="My Kind of Pong",
222             position=pygame.WINDOWPOS_CENTERED)
223         self.screen = self.window.get_surface()
224         self.clock = pygame.time.Clock()
225         self.background = pygame.sprite.GroupSingle(Background())
226         self.all_sprites = pygame.sprite.Group()
227         self.paddle = {}
228         self.paddle["left"] = Paddle("left", self.all_sprites)
229         self.paddle["right"] = Paddle("right", self.all_sprites)
230         self.ball = Ball(self.all_sprites)
231         self.score = Score(self.all_sprites)
232         self.running = True
233         self.pausing = False # Pause Flag
234         self.helping = False # Hilfe Flag
235         self.pause = pygame.sprite.GroupSingle(Pause())
236         self.help = pygame.sprite.GroupSingle(Help())

```

Nachdem nun alles vorbereitet ist, wird die Methode `update()` so gestaltet, dass nur dann neue Zustände berechnet werden, wenn keine der beiden Modi aktiv ist (Zeile 250).

Quelltext 3.37: Pong (Requirement 8) – `Game.update()`

```

249     def update(self):
250         if not (self.pausing or self.helping): # Nur bei Normalbetrieb
251             self.check_collision()
252             for i in Settings.KI.keys():
253                 if Settings.KI[i]:
254                     self.paddlecontroller(self.paddle[i])
255             self.all_sprites.update(action="move")

```

In `draw()` wird ebenfalls auf die jeweiligen Modi abgefragt und ggf. das entsprechende Sprite ausgegeben.

Quelltext 3.38: Pong (Requirement 8) – `Game.draw()`

```

257     def draw(self):
258         self.background.draw(self.screen)
259         self.all_sprites.draw(self.screen)
260         if self.pausing: # Pausedarstellung
261             self.pause.draw(self.screen)
262         elif self.helping: # Hilfedarstellung
263             self.help.draw(self.screen)
264         self.window.flip()

```

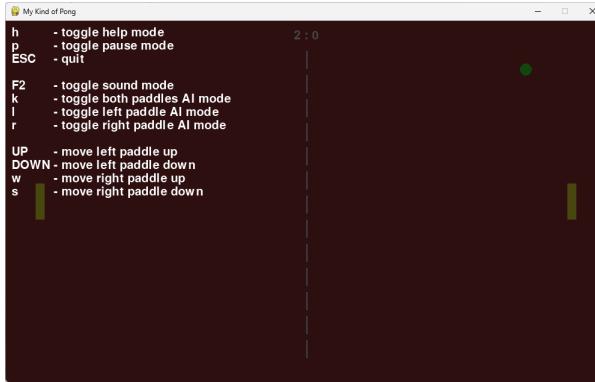


Abbildung 3.4: Pong: Hilfe-Bildschirm

3.2 Bubbles

In diesem Kapitel wird das Spiel *Bubbles* beispielhaft besprochen. Ich möchte gleich darauf hinweisen, dass die Spielidee nicht von mir stammt. Ein Schüler hat es mal als Handy-Version auf einer ITA-Messe vorgestellt. Leider kann ich mich nicht mehr an den Namen erinnern, aber auf diesem Wege ein herzliches *Dankeschön*.

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

Das Spiel lässt sich beliebig erweitern: Animation des Zerplatzens, Highscoreslisten usw., aber wie so oft ist das Bessere der Feind des Guten. Ich wünsche viel Spaß beim Studium.

3.2.1 Requirement 1: Standards

Requirement 1 Standardfunktionalität

1. Fenster hat eine angemessene Größe.
 2. Hintergrund ist eine passende Bitmap oder einfarbig.
 3. Beendet wird mit der Taste `Esc` oder per Mausklick auf rote „X“.
 4. Alle Bitmaps werden nach dem Laden konvertiert und passend skaliert.
 5. Alle Bitmaps – außer dem Hintergrund – sind transparent.
 6. Alle Bitmaps werden in `pygame.sprite.Group`- oder `pygame.sprite.GroupSingle`-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. Die statischen Angaben zum Spiel werden hier wie gewohnt in einer separaten Klasse `Settings` abgelegt.

Es wird gefordert, dass das Fenster eine angemessene Größe hat. Mit $1220\text{ px} \times 1002\text{ px}$ bin ich groß genug, um die Blasen zu verteilen und klein genug, um mit der Maus noch schnell wandern zu können. Der Rest ist in den vorherigen Kapiteln schon ausführlich behandelt worden (z.B. FPS, DELTATIME oder PATH) und wird deshalb hier nicht weiter erläutert.



Abbildung 3.5: Bubbles: Hintergrundbild (aquarium.png)

Quelltext 3.39: Bubbles (Requirement 1.1) – Präambel

```

1 import os
2 from time import time
3 from typing import Dict
4
5 import pygame
6
7
8 class Settings:
9     WINDOW = pygame.rect.Rect(0, 0, 1220, 1002)
10    FPS = 60
11    DELTATIME = 1.0 / FPS
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    CAPTION = 'Fingerübung „Bubbles“'
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)

```

Die Klasse `Background` ist eine Kindklasse von `Sprite`, 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 direkt als `Sprite`-Objekt implementieren können. Ich habe das hier nur der Übersichtlichkeit wegen gemacht. Das Hintergrundbild ist in Abbildung 3.5 zu sehen.

Quelltext 3.40: Bubbles (Requirement 1.2) – Background

```

31  class Background(pygame.sprite.Sprite):
32      def __init__(self) -> None:
33          super().__init__()
34          imagename = Settings.get_image("aquarium.png")
35          self.image: pygame.surface.Surface = pygame.image.load(imagename).convert()
36          self.image = pygame.transform.scale(self.image, Settings.WINDOW.size)
37          self.rect: pygame.rect.Rect = self.image.get_rect()

```

In der Klasse `Game` werden in `__init__()` die Pygame üblichen Verdächtigen `init()`, `Window()` und `clock()` und aufgerufen bzw. erzeugt. Auch wird das Flag der Hauptprogrammschleife `running` initialisiert. Die Methoden `run()`, `watch_for_events()`, `update()` und `draw()` enthalten nur Basisfunktionalitäten, die hier nicht weiter erläutert werden müssen.

Quelltext 3.41: Bubbles (Requirement 1) – Methoden von `Game`

```

40  class Game:
41
42      def __init__(self) -> None:
43          pygame.init()
44          self.window = pygame.Window(size=Settings.WINDOW.size, title=Settings.CAPTION,
45                                      position=pygame.WINDOWPOS_CENTERED)
46          self.screen = self.window.get_surface()
47          self.clock = pygame.time.Clock()
48          self.background = pygame.sprite.GroupSingle(Background())
49          self.running = True
50
51      def watch_for_events(self) -> None:
52          for event in pygame.event.get():
53              if event.type == pygame.QUIT:
54                  self.running = False
55              elif event.type == pygame.KEYDOWN:
56                  if event.key == pygame.K_ESCAPE:
57                      self.running = False
58
59      def draw(self) -> None:
60          self.background.draw(self.screen)
61          self.window.flip()
62
63      def update(self) -> None:
64          pass
65
66      def run(self) -> None:
67          time_previous = time()
68          self.running = True
69          while self.running:
70              self.watch_for_events()
71              self.update()
72              self.draw()
73              self.clock.tick(Settings.FPS)
74              time_current = time()
75              Settings.DELTATIME = time_current - time_previous
76              time_previous = time_current
77
78          pygame.quit()

```

Durch diese Methoden ist aber schon der generelle Ablauf des Spiels vorgegeben. Alle weiteren Eigenschaften des Spiels, sind nur noch Erweiterungen dieses Ablaufs, keine Veränderungen mehr.

Zum Schluss erfolgt der Aufruf (siehe Quelltext 3.42). Damit sind alle Unterpunkte von Requirement 1 auf Seite 151, die hier Anwendung finden, erfüllt.

Quelltext 3.42: Bubbles (Requirement 1) – Aufruf

```

79 def main():
80     Game().run()
81
82
83 if __name__ == "__main__":
84     main()

```

3.2.2 Requirement 2: Blasen erscheinen

Requirement 2 Blasen erscheinen

1. An zufälliger Position erscheint ein Blase.
2. Zu Beginn erscheint diese jede halbe Sekunde.
3. Sie hat einen Startradius von 15 px.
4. Sie hat zu den Rändern einen Abstand von mindestens 10 px.
5. Sie hat zu allen anderen Blasen einen Mindestabstand von 10 px.

Für die Blase wird die schon transparente Grafik aus Abbildung 3.6 verwendet. Die zufällige Position muss noch eingeschränkt werden. Das Aquarium füllt ja nicht den ganzen Bildschirm aus (siehe Abbildung 3.5 auf Seite 152), sondern steht innerhalb einer Art Fernseher. Wir müssen also eine Spielfläche (*playground*) definieren. Nur innerhalb dieser Spielfläche sollen die Blasen erscheinen.



Abb. 3.6: 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 20 werden die entsprechenden Werte festgehalten. Der Abstand von Spielfeldrand und der Blasen untereinander wird in Zeile 19 entsprechend Requirement 2.4 mit 10 px definiert. Der Startradius – und damit der minimale Radius – wird wegen Requirement 2.3 in Zeile 18 mit 15 px festgelegt. Während des Spielens ist mir aufgefallen, dass kleinere Startradien einfach zu schlecht gesehen werden.

Quelltext 3.43: Bubbles (Requirement 2) – Ergänzungen in `Settings`

```

18 RADIUS = {"min": 15} # Radius Startwert
19 DISTANCE = 50 # Rand-/Blasenabstand
20 PLAYGROUND = pygame.rect.Rect(90, 90, 1055, 615) # Rechteck im Aquarium

```

Die Klasse `Timer` ist exakt die oben in Kapitel 2.10 auf Seite 82 beschriebene; dort wird alles erklärt.

Quelltext 3.44: Bubbles (Requirement 2) – Timer

```

35 class Timer:
36     def __init__(self, duration: int, with_start: bool = True) -> None:
37         self.duration = duration
38         if with_start:
39             self._next = pygame.time.get_ticks()
40         else:
41             self._next = pygame.time.get_ticks() + self.duration
42
43     def is_next_stop_reached(self) -> bool:
44         if pygame.time.get_ticks() > self._next:
45             self._next = pygame.time.get_ticks() + self.duration
46             return True
47         return False

```

Schauen wir uns jetzt die Klasse `Bubble` an. Der Konstruktor ist selbsterklärend, hier werden nur die üblichen Verdächtigen bearbeitet: `image`, `rect` und `radius`. Die Methode `update()` ist derzeit leer, da noch keine Veränderung verlangt wurde. Die Methode `randompos()` wird allerdings wegen Requirement 2.1 benötigt. Sie berechnet eine neues Blasenzentrum und weist dieses `rect` zu. Ggf. muss diese Methode solange wiederholt werden, bis eine freie Fläche gefunden wird (siehe Requirement 2.4 und Requirement 2.5).

Quelltext 3.45: Bubbles (Requirement 2) – Bubble

```

59 class Bubble(pygame.sprite.Sprite):
60     def __init__(self) -> None:
61         super().__init__()
62         self.radius = Settings.RADIUS["min"]
63         imagename = Settings.get_image("blase1.png")
64         self.image: pygame.surface.Surface = pygame.image.load(imagename).convert_alpha()
65         self.image = pygame.transform.scale(self.image, (Settings.RADIUS["min"],
66             Settings.RADIUS["min"]))
66         self.rect: pygame.rect.Rect = self.image.get_rect()
67
68     def update(self, *args: Any, **kwargs: Any) -> None:
69         pass
70
71     def randompos(self) -> None:
72         bubbledistance = Settings.DISTANCE + Settings.RADIUS["min"]
73         centerx = randint(Settings.PLAYGROUND.left + bubbledistance,
74             Settings.PLAYGROUND.right - bubbledistance)
74         centery = randint(Settings.PLAYGROUND.top + bubbledistance,
75             Settings.PLAYGROUND.bottom - bubbledistance)
75         self.rect.center = (centerx, centery)

```

Die Klasse `Game` muss nun entsprechend erweitert werden. In der Zeile 84 wird das `Background`-Objekt angelegt. Zeile 85 erzeugt ein `Timer`-Objekt mit einer Intervalllänge von 500 ms, wobei im ersten Intervall noch keine Blasen erzeugt werden sollen (siehe Requirement 2.2).

Quelltext 3.46: Bubbles (Requirement 2) – Konstruktor von Game

```

78 class Game:
79     def __init__(self) -> None:
80         pygame.init()
81         self.window = pygame.Window(size=Settings.WINDOW.size, title=Settings.CAPTION,
82             position=pygame.WINDOWPOS_CENTERED)

```

```

82     self.screen = self.window.get_surface()
83     self.clock = pygame.time.Clock()
84     self.background = pygame.sprite.GroupSingle(Background()) #
85     self.timer_bubble = Timer(500, False) # Timer 500ms
86     self.all_sprites = pygame.sprite.Group() # Alle Blasen
87     self.running = True

```

In der Methode `draw()` werden lediglich die `draw()`-Methoden der Spritegruppen aufgerufen. Auch `update()` wurde angepasst, es ruft jetzt die Methode `spawn_bubble()` auf und delegiert damit die Aufgabe, neue Blasen zu erzeugen.

Quelltext 3.47: Bubbles (Requirement 2) – `draw()` und `update()` von Game

```

97     def draw(self) -> None:
98         self.background.draw(self.screen)
99         self.all_sprites.draw(self.screen)
100        self.window.flip()
101
102    def update(self) -> None:
103        self.spawn_bubble()

```

Die Grundidee hinter `spawn_bubble()` ist, solange eine Position für eine neue Blase zu raten, bis man eine freie Fläche gefunden hat. Damit man damit nicht in einer [Endlosschleife](#) landet, wird die Anzahl der Versuche auf 100 begrenzt. Wird keine Freifläche gefunden, wird die Blase nicht der Spritegruppe hinzugefügt – sie verfällt also.

Der Radius wird dazu kurzfristig erweitert (Zeile 110) und nach der Kollisionsprüfung wieder auf seinen Ursprungswert reduziert (Zeile 112).

sprite-
collide()
collide_-
circle() Sie sehen hier ein Beispiel dafür, dass der Methode `pygame.sprite.spritecollide()` eine Methodenreferenz mitgegeben wird – hier `pygame.sprite.collide_circle()` – und somit nicht die übliche Rechtecksprüfung vorgenommen wird.

Quelltext 3.48: Bubbles (Requirement 2) – `spawn_bubble()` von Game

```

105   def spawn_bubble(self) -> None:
106       if self.timer_bubble.is_next_stop_reached():
107           b = Bubble()
108           for _ in range(100):
109               b.randompos()
110               b.radius += Settings.DISTANCE # Abstand zu Blasen
111               collided = pygame.sprite.spritecollide(b, self.all_sprites, False,
112                                           pygame.sprite.collide_circle)
113               b.radius -= Settings.DISTANCE # Alter Radius!
114               if not collided:
115                   self.all_sprites.add(b)
116                   break

```

Das Ergebnis können Sie in Abbildung 3.7 auf der nächsten Seite sehen. Gleichmäßig sind die Bubbles auf der Spielfläche verteilt und der geforderte Abstand zum Rand und zwischen den Blasen ist dabei eingehalten.



Abbildung 3.7: Bubbles: Die Blasen haben beim Start einen Mindestabstand

3.2.3 Requirement 3: Blasenanzahl

Requirement 3 Blasenanzahl

Die maximale Anzahl der Blasen soll von der Spielfeldgröße abhängen.

Die maximale Anzahl will ich in `Game` festlegen. Ausgehend von der Fläche wird eine Obergrenze festgelegt:

Quelltext 3.49: Bubbles (Requirement 3) – Ergänzung von `Settings`

```
20 PLAYGROUND = pygame.rect.Rect(90, 90, 1055, 615)
21 MAX_BUBBLES = PLAYGROUND.height * PLAYGROUND.width // 10000 # Erfahrungswert
```

Diese Obergrenze aus Zeile 21 wird in Zeile 108 abgefragt. Nur wenn die maximale Anzahl noch nicht erreicht wurde, wird eine neue Blase erzeugt.

Quelltext 3.50: Bubbles (Requirement 3) – Ergänzung von `Game` in `spawn_bubbles()`

```
106 def spawn_bubble(self) -> None:
107     if self.timer_bubble.is_next_stop_reached():
108         if len(self.all_sprites) <= Settings.MAX_BUBBLES: # Platz?
109             b = Bubble()
110             for _ in range(100):
111                 b.randompos()
112                 b.radius += Settings.DISTANCE
113                 collided = pygame.sprite.spritecollide(b, self.all_sprites, False,
114                                             pygame.sprite.collide_circle)
115                 b.radius -= Settings.DISTANCE
116                 if not collided:
117                     self.all_sprites.add(b)
                     break
```

Der Rest des Programmes bleibt unverändert.

3.2.4 Requirement 4: Blasenwachstum

Requirement 4 Blasenwachstum

1. Die verschiedenen großen Blasen werden in einem Container verwaltet.
2. Der maximale Radius einer Blase ist 240 px.

Der Sinn von Requirement 4.1 ist das Einsparen von Rechenzeit. Im Spiel werden immer wieder Blasen mit einem bestimmten Radius starten und dann wachsen. Jedes mal das Bitmap auf die passende Größe zu skalieren, würde Rechenzeit verschwenden – schließlich wird die gleiche Blase ja mit den Radien mehrfach vorkommen. Aus diesem Grund ist es sinnvoll, einmal die Blase in alle möglichen Radien zu skalieren und das Ergebnis in einem Dictionary abzulegen. Der Key ist dabei der jeweilige Radius (siehe Zeile 64).

Die Methode `get()` liefert mir dann zu einem Radius das passend skalierte und schon fertige Image. Vorab wird in den Zeilen 67 und 68 überprüft, ob der Radius innerhalb des Gültigkeitsbereich liegt. Falls der Radius dabei zu groß ist, wird der maximale genommen und falls er zu klein ist, der minimale.

Quelltext 3.51: Bubbles (Requirement 4.1) – BubbleContainer

```

60  class BubbleContainer:
61      def __init__(self) -> None:
62          imagename = Settings.get_image("blase1.png")
63          image: pygame.surface.Surface = pygame.image.load(imagename).convert_alpha()
64          self.images = {i: pygame.transform.scale(image, (i * 2, i * 2)) for i in
65              range(Settings.RADIUS["min"], Settings.RADIUS["max"] + 1)} #
66
67      def get(self, radius: int) -> pygame.surface.Surface:
68          radius = max(Settings.RADIUS["min"], radius) # Untere Grenze
69          radius = min(Settings.RADIUS["max"], radius) # Obere Grenze
70          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 18 passend zu Requirement 4.2 um die Angabe eines maximalen Radius erweitert.

Quelltext 3.52: Bubbles (Requirement 4.2) – Erweiterung von Settings

```
18  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 77 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 89). Der neue Radius wird in Zeile 86 bestimmt. In der gleichen Zeile wird dieser

Wert mit dem maximalen Radius aus `Settings` verglichen und das Minimum der beiden ausgewählt. Diese Logik verhindert, dass der Radius zu groß wird.

Was hat es aber mit den Zeilen 88 und 91 auf sich? Der Referenzpunkt eines Image in einem Sprite ist die linke, obere Ecke. Wächst jetzt die Blase, würde sie sich nach rechts und unten vergrößern; der linke und obere Rand blieben gleich, was hässlich aussieht. Daher merken wir uns den alten Mittelpunkt, laden das neue Image, erzeugen das passende `Rect`-Objekt und verschieben es dann wieder auf den alten Mittelpunkt, so dass optisch die Blase vom Mittelpunkt aus in alle Richtungen wächst.

Quelltext 3.53: Bubbles (Requirement 4) – Ergänzung von `Bubble`

```

72 class Bubble(pygame.sprite.Sprite):
73     def __init__(self, bubble_container: BubbleContainer) -> None:
74         super().__init__()
75         self.bubble_container = bubble_container # Verweis auf Container
76         self.radius = Settings.RADIUS["min"]
77         self.image = self.bubble_container.get(self.radius) # Zugriff auf Bubbles
78         self.rect: pygame.rect.Rect = self.image.get_rect()
79         self.fradius = float(self.radius)
80         self.speed = 100
81
82     def update(self, *args: Any, **kwargs: Any) -> None:
83         if "action" in kwargs.keys():
84             if kwargs["action"] == "grow":
85                 self.fradius += self.speed * Settings.DELTATIME
86                 self.fradius = min(self.fradius, Settings.RADIUS["max"]) # Neuer Radius
87                 self.radius = round(self.fradius)
88                 center = self.rect.center # Alter Mittelpunkt
89                 self.image = self.bubble_container.get(self.radius) # Neues Image
90                 self.rect = self.image.get_rect()
91                 self.rect.center = center # Neuer MP = Alter MP

```

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 126).

Quelltext 3.54: Bubbles (Requirement 4) – Ergänzung von `update()` in `Game`

```

125     def update(self) -> None:
126         self.all_sprites.update(action="grow") # Bubbles aktualisieren
127         self.spawn_bubble()

```

Der `BubbleContainer` wird angelegt

Quelltext 3.55: Bubbles (Requirement 4) – Ergänzung im Konstruktor von `Game`

```

106         self.background = pygame.sprite.GroupSingle(Background())
107         self.bubble_container = BubbleContainer()
108         self.timer_bubble = Timer(500, False)

```

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

Quelltext 3.56: Bubbles (Requirement 4) – Ergänzung von `spawn_bubble()` in Game

```

129     def spawn_bubble(self) -> None:
130         if self.timer_bubble.is_next_stop_reached():
131             if len(self.all_sprites) <= Settings.MAX_BUBBLES:
132                 b = Bubble(self.bubble_container) # Verweis auf Bubbles
133                 for _ in range(100):

```

Die Blasen wachsen nun um ihren Mittelpunkt herum nach außen. Das Ergebnis könnte dann wie in Abbildung 3.8 aussehen.



Abbildung 3.8: Bubbles: Die Blasen sind gewachsen/verwachsen

3.2.5 Requirement 5: Mauscursor

Requirement 5 Mauscursor

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

Durch diese Anforderung soll der Spieler optisch unterstützt werden. Er kann schneller erkennen, ob er die Blase schon erreicht hat. Pygame selbst kennt keine Methode/Funktion um zu testen, ob ein Punkt innerhalb eines Kreises liegt. Die Abbildung 3.9 auf der nächsten Seite liefert mir aber einen einfachen Ansatz, das Problem zu lösen.

Der Wert d ist der Abstand in Pixel zwischen dem Mittelpunkt des Kreises (x_1, y_1) und dem Punkt (x_2, y_2) . Ist $d \leq r$, so liegt der Punkt innerhalb des Kreises bzw. berührt ihn. Erweitern wir also `Bubble` um eine entsprechende Methode.

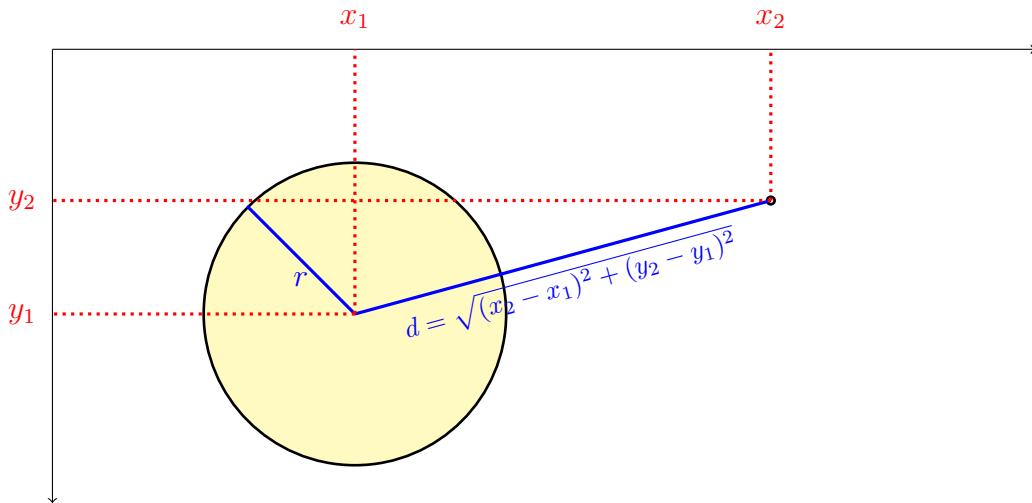


Abbildung 3.9: Kollisionserkennung: Punkt innerhalb des Kreises (Satz des Pythagoras)?

Quelltext 3.57: Bubbles (Requirement 5) – collidepoint() in Game

```

144     def collidepoint(self, point: Tuple[int, int], sprite: pygame.sprite.Sprite) -> bool:
145         if hasattr(sprite, "radius"):
146             deltax = point[0] - sprite.rect.centerx
147             deltay = point[1] - sprite.rect.centery
148             return sqrt(deltax * deltax + deltay * deltay) <= sprite.radius
149         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 155 in die Methode `Bubble.collidepoint()` gestopft. Falls eine Blase gefunden wurde, die mit der Maus kollidiert, wird das Flag auf `True` gesetzt und die Schleife mit `break` beendet, was uns ein wenig Rechenzeit einspart, da so nicht mehr alle anderen Blasen untersucht werden. Abhängig vom Flag wird dann der Mauscursor gesetzt.

Quelltext 3.58: Bubbles (Requirement 5) – set_mousecursor() in Game

```

151     def set_mousecursor(self) -> None:
152         is_over = False
153         pos = pygame.mouse.get_pos()
154         for b in self.all_sprites:
155             if self.collidepoint(pos, b): # Innerhalb?
156                 is_over = True
157                 break
158         if is_over:
159             pygame.mouse.set_cursor(pygame.SYSTEM_CURSOR_HAND)
160         else:

```

161

```
pygame.mouse.set_cursor(pygame.SYSTEM_CURSOR_CROSSHAIR)
```

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

Quelltext 3.59: Bubbles (Requirement 5) – `update()` in `Game`

```
126     def update(self) -> None:
127         self.all_sprites.update(action="grow")
128         self.spawn_bubble()
129         self.set_mousecursor()  # Mauscursor
```

Testen Sie das Programm mal aus. Positionieren Sie die Maus in eine linke untere Ecke außerhalb einer Blase und warten Sie, bis durch das Wachsen die Blase die Maus berührt.

3.2.6 Requirement 6: Blasen zerplatzen

Requirement 6 Blasen zerplatzen

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

MOUSE-
BUTTON-
DOWN
get_pos()

Für die Umsetzung dieser Anforderung ist schon mit der Implementierung der Methode `Bubble.collidepoint()` fast alles erledigt. Wir müssen diese Methode nur geschickt einsetzen – es sind in der Tat nur wenige Restarbeiten nötig. In `watch_for_events()` wird zunächst der linke Mausklick abgefangen (Zeile 120) und die aktuelle Mausposition an die – neu erstellte Methode – `sting()` übergeben (Zeile 122).

Hinweis: Implementieren Sie grundsätzlich so wenig Logik wie möglich in `watch_for_events()`. Diese Methode ist ein Verteiler; die Verarbeitung sollte immer in Methoden ausgelagert werden.

Quelltext 3.60: Bubbles (Requirement 6) – `watch_for_event()` in `Game`

```
113     def watch_for_events(self) -> None:
114         for event in pygame.event.get():
115             if event.type == pygame.QUIT:
116                 self.running = False
117             elif event.type == pygame.KEYDOWN:
118                 if event.key == pygame.K_ESCAPE:
119                     self.running = False
120             elif event.type == pygame.MOUSEBUTTONDOWN:  # Mausklick?
121                 if event.button == 1:  # left
122                     self.sting(pygame.mouse.get_pos())  # Aufruf
```

kill()

Die Methode `sting()` ist nun denkbar simpel. Es werden alle `Bubble`-Objekte durchwandert und dahingehend abgefragt, ob die Mausposition innerhalb des Radius liegt (Zeile 168). Wenn *Ja*, dann wird das entsprechende Objekt aus der Spritegroup mit `kill()` entfernt.

Quelltext 3.61: Bubbles (Requirement 6) – `sting()` in Game

```
166     def sting(self, mousepos: Tuple[int, int]) -> None:
167         for bubble in self.all_sprites:
168             if self.collidepoint(mousepos, bubble): # Innerhalb?
169                 bubble.kill()
```

3.2.7 Requirement 7: Punktestand**Requirement 7** Punktestand

1. Das Spiel startet mit 0 Punkten.
2. Zerplatzt eine Blase, wird der Punktestand proportional zum Radius erhöht.
3. Der Punktestand wird im unteren Teil angezeigt.

Das Anstechen der Blasen soll natürlich mit Punkten belohnt werden. Dazu müssen die Punkte ermittelt und ausgegeben werden. Die einfachste Art den Punktestand festzuhalten ist eine statische Variable in `Settings` oder eine globale Variable. Ich bevorzuge Variante 1 (Quelltext 3.62).

Quelltext 3.62: Bubbles (Requirement 7.1) – Erweiterung von Game

```
23     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 Methode in `Bubble` angelegt. In Zeile 106 wird einfach der Radius der Blase auf den Punktestand addiert.

Quelltext 3.63: Bubbles (Requirement 7.2) – `stung()` in Bubble

```
104     def stung(self):
105         self.kill()
106         Settings.POINTS += self.radius # Increment points
```

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

Quelltext 3.64: Bubbles (Requirement 7.2) – `update()` in Bubble

```
85     def update(self, *args: Any, **kwargs: Any) -> None:
86         if "action" in kwargs.keys():
87             if kwargs["action"] == "grow":
88                 self.fradius += self.speed * Settings.DELTATIME
89                 self.fradius = min(self.fradius, Settings.RADIUS["max"])
90                 self.radius = round(self.fradius)
91                 center = self.rect.center
92                 self.image = self.bubble_container.get(self.radius)
93                 self.rect = self.image.get_rect()
94                 self.rect.center = center
```

```
95         elif kwargs["action"] == "sting":
96             self.stung()
```

Die Methoden `sting()` und `update()` in `Game` müssen dazu passend verändert werden (Zeile 192 und Zeile 153).

Quelltext 3.65: Bubbles (Requirement 7.2) – `sting()` in `Game`

```
189     def sting(self, mousepos: Tuple[int, int]) -> None:
190         for bubble in self.all_sprites:
191             if self.collidepoint(mousepos, bubble):
192                 bubble.update(action="sting") # Nach Bubble verschoben
```

Quelltext 3.66: Bubbles (Requirement 7.2) – `update()` in `Game`

```
152     def update(self) -> None:
153         self.all_sprites.update(action="grow") #
154         self.spawn_bubble()
155         self.set_mousecursor()
```

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

Quelltext 3.67: Bubbles (Requirement 7.3) – Erweiterung von `Settings`

```
24     BOX = pygame.rect.Rect(90, 770, 1055, 130) # Ausgabebox
```

Für die Punktausgabe selbst bastle ich mir wieder eine kleine Klasse, die das Problem kapselt: `Points`. Im Konstruktor wird ein `Font`-Objekt erzeugt, welches mir in `update()` den Punktestand rendert. Die Position der Textausgabe wird aus den Angaben in `Settings` ermittelt. Den Rest erledigt die `Sprite`-Klasse für mich.

Quelltext 3.68: Bubbles (Requirement 7.3) – `Points`

```
109     class Points(pygame.sprite.Sprite):
110         def __init__(self) -> None:
111             super().__init__()
112             self.font = pygame.font.Font(pygame.font.get_default_font(), 18)
113             self.oldpoints = -1
114
115         def update(self, *args: Any, **kwargs: Any) -> None:
116             if self.oldpoints != Settings.POINTS:
117                 self.image = self.font.render(f"Points: {Settings.POINTS}", True, "red")
118                 self.rect = self.image.get_rect()
119                 self.rect.left = Settings.BOX.left
120                 self.rect.top = Settings.BOX.top
```

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

Quelltext 3.69: Bubbles (Requirement 7.3) – Erweiterung des Konstruktors von `Game`

```
132     self.all_sprites = pygame.sprite.Group()
133     self.all_sprites.add(Points())
134     self.running = True
```



Abbildung 3.10: Bubbles: Ausgabe Punktestand

In Abbildung 3.10 können Sie die Punkteausgabe in der unteren Hälfte sehen. Diese Fläche könnte man später auch noch für eine Liste der besten zehn Punktestände oder andere Ausgaben verwenden.

3.2.8 Requirement 8: Spielende

Requirement 8 Spielende

1. Berühren sich zwei Blasen, ist das Spiel verloren.
 2. Berührt eine Blase den Rand, ist das Spiel verloren.
-

Hinweis: Damit das Spiel spielbar wird, habe ich die Wachstumsgeschwindigkeit einer Bubble auf 10 gesetzt.

Quelltext 3.70: Bubbles (Requirement 8) – `Bubble.speed`

```
83     self.speed = 10
```

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

Quelltext 3.71: Bubbles (Requirement 8) – Erweiterung von `update()` in `Game`

```
155     def update(self) -> None:
156         if self.check_bubblecollision(): # Spielende?
157             self.running = False
158         else:
159             self.all_sprites.update(action="grow")
160             self.spawn_bubble()
161             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 156) 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 201 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.

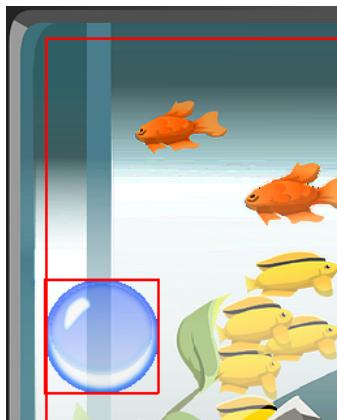


Abbildung 3.11: Bubbles – Kollision mit dem Rand

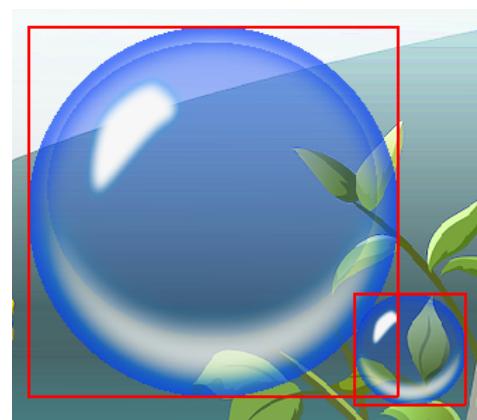


Abbildung 3.12: Bubbles – Kollision der Blasen

In Zeile 206 wird Requirement 8.1 überprüft. Dabei wird die auf der Kreisform basierende Kollisionsprüfung mit `collide_circle()` verwendet. In der Zeile 208 und Zeile 210 wird Requirement 8.2 umgesetzt. Dabei wird ausgenutzt, dass die Spielfläche ein Rechteck ist und das Sprite ebenfalls ein Rechteck besitzt. Die Methode `pygame.rect.Rect.contains()` überprüft dabei, ob ein Rechteck innerhalb eines anderen liegt. Ist dies nicht der Fall – also verlässt die Blase die Spielfläche –, liegt eine Kollision vor.

Quelltext 3.72: Bubbles (Requirement 8) – check_bubblecollision() in Game

```

200  def check_bubblecollision(self) -> bool:
201      for index1 in range(0, len(self.all_sprites) - 1): # Bubbles prüfen
202          for index2 in range(index1 + 1, len(self.all_sprites)):
203              bubble1 = self.all_sprites.sprites()[index1]
204              bubble2 = self.all_sprites.sprites()[index2]
205              if type(bubble1).__name__ == "Bubble" and type(bubble2).__name__ == "Bubble":
206                  if pygame.sprite.collide_circle(bubble1, bubble2): # Blasen kollidieren
207                      return True
208                  if not Settings.PLAYGROUND.contains(bubble1): # Blase1 berührt Rand
209                      return True
210                  if not Settings.PLAYGROUND.contains(bubble2): # Blase2 berührt Rand
211                      return True
212      return False

```

In Abbildung 3.11 auf der vorherigen Seite wird die Kollision der Blase mit dem Rand dargestellt. Um das besser erkennen zu können, habe ich Hilfslinien ausgegeben. Sie können gut sehen, dass das Rechteck der Blase nicht mehr im Rechteck der Spielfläche liegt. Abbildung 3.12 auf der vorherigen Seite zeigt die Kollision zweier Blasen. Auch hier sind Hilfslinien eingezeichnet. Die Hilfslinien werden Ihnen eingezeichnet, wenn Sie die drei Kommentarzeichen in `Game.draw()` entfernen.

Quelltext 3.73: Bubbles (Requirement 8) – Hilfslinien in Game

```

147  def draw(self) -> None:
148      self.background.draw(self.screen)
149      self.all_sprites.draw(self.screen)
150      # pygame.draw.rect(self._screen, "red", Settings.PLAYGROUND, 2)
151      # for b in self._all_sprites:
152      #     pygame.draw.rect(self._screen, "red", b.rect, 2)
153      self.window.flip()

```

3.2.9 Requirement 9: Zeitanpassungen

Requirement 9 Zeitanpassungen

Die Blasen sollen im Lauf der Zeit schneller wachsen.

Weil im Laufe der Zeit die Blasen schneller wachsen sollen, will ich ihnen die Wachstumsgeschwindigkeit als Übergabeparameter im Konstruktor mitgeben. In Zeile 83 wird dieser Parameter in ein Attribut geparkt.

Quelltext 3.74: Bubbles (Requirement 9) – Bubble

```

75  class Bubble(pygame.sprite.Sprite):
76      def __init__(self, bubble_container: BubbleContainer, speed: int) -> None:
77          super().__init__()
78          self.bubble_container = bubble_container
79          self.radius = Settings.RADIUS["min"]
80          self.image = self.bubble_container.get(self.radius)
81          self.rect: pygame.rect.Rect = self.image.get_rect()
82          self.radius = float(self.radius)
83          self.speed = speed # Wachstumsgeschwindigkeit

```

Timer

Das sind schon alle Anpassungen in `Bubble`, der Rest passiert in `Game`. In Zeile 132 wird ein Timer erstellt, der mir alle 1000 *ms* ein Signal geben wird. Darunter wird die anfängliche Wachstumsgeschwindigkeit der Blasen auf 10 *px/s* gestellt.

Quelltext 3.75: Bubbles (Requirement 9) – Konstruktor von `Game`

```

130     self.bubble_container = BubbleContainer()
131     self.timer_bubble = Timer(500, False)
132     self.timer_bubble_speed = Timer(1000, False)  #
133     self.bubble_speed = 10
134     self.all_sprites = pygame.sprite.Group()

```

In `spawn_bubble()` wird der Timer abgefragt und ggf. die Blasenwachstumsgeschwindigkeit¹ erhöht (Zeile 163). Die maximale Wachstumsgeschwindigkeit wird dabei auf 100 *px/s* begrenzt; schneller scheint mir nicht spielbar. Bei jedem Timer-Signal wird dabei die Geschwindigkeit um 5 *px/s* erhöht. Dies geschieht in dieser Methode, da dann die neue Geschwindigkeit für die zu erstellenden Blasen zur Verfügung steht.

Quelltext 3.76: Bubbles (Requirement 9) – `Game.spawn_bubble()`

```

162     def spawn_bubble(self) -> None:
163         if self.timer_bubble_speed.is_next_stop_reached():  #
164             if self.bubble_speed < 100:
165                 self.bubble_speed += 5
166         if self.timer_bubble.is_next_stop_reached():
167             if len(self.all_sprites) <= Settings.MAX_BUBBLES:
168                 b = Bubble(self.bubble_container, self.bubble_speed)
169                 for _ in range(100):
170                     b.randompos()
171                     b.radius += Settings.DISTANCE
172                     collided = pygame.sprite.spritecollide(b, self.all_sprites, False,
173                                                 pygame.sprite.collide_circle)
174                     b.radius -= Settings.DISTANCE
175                     if not collided:
176                         self.all_sprites.add(b)
177                         break

```

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

3.2.10 Requirement 10: Kollision anzeigen

Requirement 10 Kollision anzeigen

Wenn Blasen mit dem Rand oder miteinander kollidieren, sollen sie die Farbe wechseln und für 2 s sichtbar bleiben, bevor die Anwendung sich beendet.

¹ Deutsch ist schon eine coole Sprache ;-)

Bisher beendet sich das Spiel so schnell, dass ich nicht überprüfen kann, ob ich eigentlich zu Recht verloren habe, oder ob das Programm spinnt. Ich möchte durch diese Anforderung die beiden kollidierenden Blasen oder die Blase, die den Rand berührt, andersfarbig sehen können. Ich habe dazu die Blase rot eingefärbt (siehe Abbildung 3.13).



Abb. 3.13: Blase 2

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

Quelltext 3.77: Bubbles (Requirement 10) – Erweiterung von `Game`

```
130 class Game:
131     BUBBLE_CONTAINER: Dict[str, BubbleContainer] = {} # Mehrere
```

Der Konstruktor von `BubbleContainer` bekommt nun einen Dateinamen mitgegeben, so dass hier verschiedene Grafiken zu Grunde gelegt werden können.

Quelltext 3.78: Bubbles (Requirement 10) – Änderung Konstruktor von `BubbleContainer`

```
63 class BubbleContainer:
64     def __init__(self, filename: str) -> None: # Jetzt mit Dateinamen
65         imagename = Settings.get_image(filename)
66         image: pygame.surface.Surface = pygame.image.load(imagename).convert_alpha()
67         self.images = {i: pygame.transform.scale(image, (i * 2, i * 2)) for i in
range(Settings.RADIUS["min"], Settings.RADIUS["max"] + 1)}
```

Der Konstruktor von `Game` füllt nun das statische Dictionary `BUBBLE_CONTAINER` auf (Zeile 138 und Zeile 139).

Quelltext 3.79: Bubbles (Requirement 10) – Änderung vom Konstruktor von `Game`

```
137     self.clock = pygame.time.Clock()
138     Game.BUBBLE_CONTAINER["blue"] = BubbleContainer("blase1.png") # blau
139     Game.BUBBLE_CONTAINER["red"] = BubbleContainer("blase2.png") # rot
140     self.background = pygame.sprite.GroupSingle(Background())
```

In `Bubble` sind nun mehrere Änderungen nötig. Durch das neue Attribut `mode` (Zeile 78) 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 92 in `update()` erwähnt.

Quelltext 3.80: Bubbles (Requirement 10) – Konstruktor von `Bubble` und `update()`

```
75 class Bubble(pygame.sprite.Sprite):
76     def __init__(self, speed: int) -> None:
77         super().__init__()
```

```

78         self.mode = "blue" # Farbmodus
79         self.radius = Settings.RADIUS["min"]
80         self.image = Game.BUBBLE_CONTAINER[self.mode].get(self.radius)
81         self.rect: pygame.rect.Rect = self.image.get_rect()
82         self.fradius = float(self.radius)
83         self.speed = speed
84
85     def update(self, *args: Any, **kwargs: Any) -> None:
86         if "action" in kwargs.keys():
87             if kwargs["action"] == "grow":
88                 self.fradius += self.speed * Settings.DELTATIME
89                 self.fradius = min(self.fradius, Settings.RADIUS["max"])
90                 self.radius = round(self.fradius)
91                 center = self.rect.center
92                 self.image = Game.BUBBLE_CONTAINER[self.mode].get(self.radius) #
93                 self.rect = self.image.get_rect()
94                 self.rect.center = center
95             elif kwargs["action"] == "sting":
96                 self.stung()
97             elif "mode" in kwargs.keys():
98                 self.set_mode(kwargs["mode"])

```

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

Quelltext 3.81: Bubbles (Requirement 10) – `set_mode()` in `Bubble`

```

100    def set_mode(self, mode: str) -> None:
101        if mode != self.mode:
102            self.mode = mode
103            self.image = Game.BUBBLE_CONTAINER[self.mode].get(self.radius)

```

Jetzt muss nur noch im Falle einer Kollision – also eines Spielendes – der Modus geändert werden. In Abbildung 3.14 auf der nächsten Seite können Sie sehen, wie die beiden kollidierenden Blasen rot erscheinen. Wie das geschieht, können Sie beispielhaft in Zeile 219 sehen.

Quelltext 3.82: Bubbles (Requirement 10) – `check_bubblecollision()` in `Game`

```

212     def check_bubblecollision(self) -> bool:
213         for index1 in range(0, len(self.all_sprites) - 1):
214             for index2 in range(index1 + 1, len(self.all_sprites)):
215                 bubble1 = self.all_sprites.sprites()[index1]
216                 bubble2 = self.all_sprites.sprites()[index2]
217                 if type(bubble1).__name__ == "Bubble" and type(bubble2).__name__ == "Bubble":
218                     if pygame.sprite.collide_circle(bubble1, bubble2):
219                         bubble1.update(mode="red") # rot
220                         bubble2.update(mode="red")
221                         return True
222                     if not Settings.PLAYGROUND.contains(bubble1):
223                         bubble1.update(mode="red")
224                         return True
225                     if not Settings.PLAYGROUND.contains(bubble2):
226                         bubble2.update(mode="red")
227                         return True
228         return False

```

Damit mir Zeit bleibt, die Kollision zu sehen, will ich am Ende 2 s warten. Die Methode `pygame.time.wait()` hält die Anwendung entsprechend lang an (Zeile 241).

`wait()`

Quelltext 3.83: Bubbles (Requirement 10) – Wartezeit in `run()`

```

230  def run(self) -> None:
231      time_previous = time()
232      self.running = True
233      while self.running:
234          self.watch_for_events()
235          self.update()
236          self.draw()
237          self.clock.tick(Settings.FPS)
238          time_current = time()
239          Settings.DELTATIME = time_current - time_previous
240          time_previous = time_current
241          pygame.time.wait(2000)  # Kurz warten
242          pygame.quit()

```



Abbildung 3.14: Bubbles: Kollision anzeigen

3.2.11 Requirement 11: Pause

Requirement 11 Pause

*Mit der rechten Maustaste oder der Taste **P** springt das Spiel in den Pausenmodus oder beendet diesen. Der aktuelle Spielstand friert ein und wird „eingegraut“.*

Die Idee hinter dieser Anforderung ist, dass eine notwendige Unterbrechung nicht zwangsläufig bedeutet, dass man verliert. In Abbildung 3.15 auf der nächsten Seite können Sie sehen, wie der Pausenbildschirm aussehen sollte.

Im Konstruktor von `Game` wird das Flag `pausing` definiert. Dieser steuert später, ob sich das Spiel im Pausenmodus befindet oder nicht.

Quelltext 3.84: Bubbles (Requirement 11) – Konstruktor in `Game`

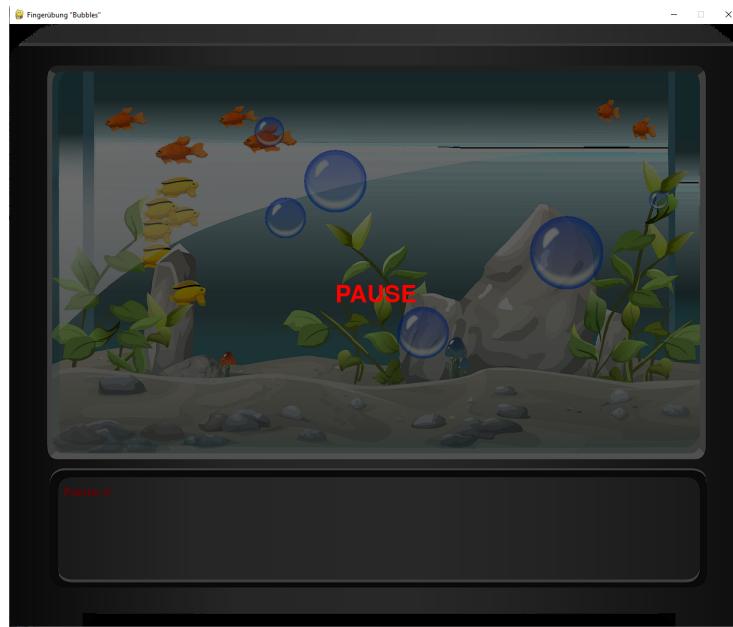


Abbildung 3.15: Bubbles: Pausenbildschirm

```

154         self.running = True
155         self.pausing = False #*
156         self.msgpause = Pause()

```

In `watch_for_events()` wird nun abgefragt, ob die P-Taste (Zeile 166) oder die rechte Maustaste (Zeile 169) gedrückt wurde. In beiden Fällen wird die neue Methode `setpause()` aufgerufen.

Quelltext 3.85: Bubbles (Requirement 11) – `watch_for_events()` in Game

```

158     def watch_for_events(self) -> None:
159         for event in pygame.event.get():
160             if event.type == pygame.QUIT:
161                 self.running = False
162             elif event.type == pygame.KEYDOWN:
163                 if event.key == pygame.K_ESCAPE:
164                     self.running = False
165                 elif event.type == pygame.KEYUP:
166                     if event.key == pygame.K_p: #*
167                         self.setpause()
168                     elif event.type == pygame.MOUSEBUTTONUP:
169                         if event.button == 3: # right
170                             self.setpause()
171                     elif event.type == pygame.MOUSEBUTTONDOWN:
172                         if event.button == 1: # left
173                             self.sting(pygame.mouse.get_pos())

```

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

Quelltext 3.86: Bubbles (Requirement 11) – Pause

```

75 class Pause(pygame.sprite.Sprite):
76     def __init__(self) -> None:
77         super().__init__()
78         imagename = Settings.get_image("pause.png")
79         self.image: pygame.surface.Surface = pygame.image.load(imagename).convert_alpha()
80         self.rect = self.image.get_rect()

```

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

Quelltext 3.87: Bubbles (Requirement 11) – Konstruktor in `Game`

```
156     self.msgpause = 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 3.88: Bubbles (Requirement 11) – `setpause()` in `Game`

```

189     def setpause(self):
190         if not self.pausing:
191             self.all_sprites.add(self.msgpause)
192         else:
193             self.msgpause.kill()
194             self.pausing = not self.pausing

```

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

3.2.12 Requirement 12: Neustart

Requirement 12 Neustart

Am Ende des Spiels soll erfragt werden, ob der Spieler das Spiel neu starten möchte oder nicht.

Die Grundidee der Implementierung ist dabei, dass mit Hilfe von zwei Flags der Status des Spiels festgelegt wird. Wie bei der Pause brauchen wir ein Flag, welches steuert, ob der halbtransparente Vordergrund über das Spiel gelegt wird ([restarting](#)). Dies ist immer dann der Fall, wenn die Kollisionsprüfung der Blasen eine Kollision feststellt.

Flag

Das andere Flag – `do_start` – markiert, ob der Spieler einen Neustart möchte. An den entscheidenden Stellen in `update()` und `draw()` werden dann diese Flags abgefragt.

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 76).

Quelltext 3.89: Bubbles (Requirement 12) – von Pause zu Message

```

75  class Message(pygame.sprite.Sprite):
76      def __init__(self, filename: str) -> None: #
77          super().__init__()
78          imagename = Settings.get_image(filename)
79          self.image: pygame.surface.Surface = pygame.image.load(imagename).convert_alpha()
80          self.rect = self.image.get_rect()

```

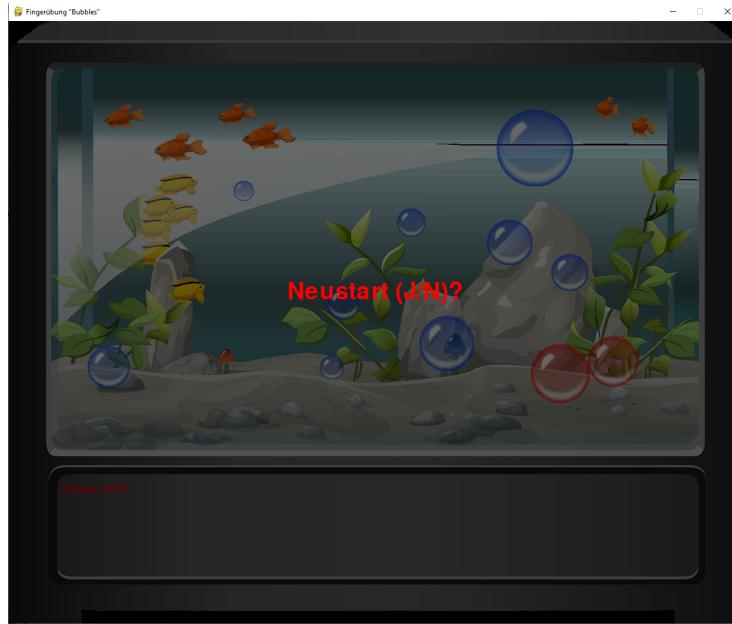


Abbildung 3.16: Bubbles: Neustartbildschirm

In Game werden im Konstruktor ab Zeile 152 die Anpassungen für den Neustart programmiert. Im wesentlichen werden für die Pause und den Neustart die beiden Message-Objekte erzeugt und alle Attribute, die bei einem Start/Neustart zurückgesetzt werden müssen, werden in der neuen Methode `restart()` bearbeitet.

Quelltext 3.90: Bubbles (Requirement 12) – Umbau Konstruktor von Game

```

151      self.pausing = False
152      self.msg_pause = Message("pause.png") #
153      self.msg_restart = Message("neustart.png")
154      self.restart()

```

Der Punktestand wird zurückgesetzt, die Spritegroup von den Blasen geleert, die Timer neu aufgesetzt, die Blasenwachstumsgeschwindigkeit auf Anfang gesetzt und die beiden oben beschriebenen Flags auf `False` gesetzt.

Quelltext 3.91: Bubbles (Requirement 12) – `restart()` in Game

```

195  def restart(self):
196      Settings.POINTS = 0

```

```

197     self.all_sprites.empty()
198     self.all_sprites.add(Points())
199     self.bubble_speed = 10
200     self.timer_bubble = Timer(500, False)
201     self.timer_bubble_speed = Timer(10000, False)
202     self.do_start = False
203     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 3.92: Bubbles (Requirement 12) – `update()` in `Game`

```

182     def update(self) -> None:
183         if self.do_start: # Neustart?
184             self.restart()
185         if not self.pausing and self.running:
186             if self.check_bubblecollision():
187                 if not self.restarting:
188                     self.all_sprites.add(self.msg_restart)
189                     self.restarting = True
190             else:
191                 self.all_sprites.update(action="grow")
192                 self.spawn_bubble()
193                 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 166), 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 168).

Quelltext 3.93: Bubbles (Requirement 12) – Erweiterung von `watch_for_events()`

```

163     elif event.type == pygame.KEYUP:
164         if event.key == pygame.K_p:
165             self.setpause()
166         elif event.key == pygame.K_j: #
167             self.do_start = True
168         elif event.key == pygame.K_n: #
169             self.running = False

```

Da wir nun am Spielende eine halbtransparente Vordergrundausgabe haben, brauchen wir keine zweisekündige Pause, um die kollidierenden Blasen anzusehen (siehe Abbildung 3.16 auf der vorherigen Seite).

Quelltext 3.94: Bubbles (Requirement 12) – `run()` in `Game`

```

281     pygame.quit()

```

3.2.13 Requirement 13: Sound

Requirement 13 Sound

1. Das Erscheinen der Blasen wird mit einem Sound unterlegt.
2. Das Zerstechen wird mit einem Sound unterlegt.
3. Das Berühren wird mit einem Sound unterlegt.

Zum Schluss noch eine kleine Sound-Untermalung. Ähnlich wie bei den Blasen-Sprites, möchte ich keine Performance durch permanentes Laden der Sound-Dateien verlieren. Daher werden die Sounds in einem statischen Dictionary abgelegt (Zeile 140).

Quelltext 3.95: Bubbles (Requirement 13) – SOUND_CONTAINER

```
138 class Game:
139     BUBBLE_CONTAINER: Dict[str, BubbleContainer] = {}
140     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 149ff.).

Sound

Quelltext 3.96: Bubbles (Requirement 13) – SOUND_CONTAINER auffüllen

```
149     Game.SOUND_CONTAINER["bubble"] =
150         pygame.mixer.Sound(Settings.get_sound("plopp1.mp3")) #
151     Game.SOUND_CONTAINER["burst"] = pygame.mixer.Sound(Settings.get_sound("burst.mp3"))
151     Game.SOUND_CONTAINER["clash"] = pygame.mixer.Sound(Settings.get_sound("glas.wav"))
```

play()

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 232.

Quelltext 3.97: Bubbles (Requirement 13.1) – spawn_bubble()

```
230         if not collided:
231             self.all_sprites.add(b)
232             Game.SOUND_CONTAINER["bubble"].play() #
233             break
```

Dann, wenn in `sting()` eine Blase zerplatzt (Zeile 257):

Quelltext 3.98: Bubbles (Requirement 13.2) – sting()

```
256     if self.collidepoint(mousepos, bubble):
257         Game.SOUND_CONTAINER["burst"].play() #
258         bubble.update(action="sting")
```

Und zum Schluss bei der Kollision mit anderen Blasen oder dem Rand in `update()`. Dabei muss noch berücksichtigt werden, ob das Spiel gerade den Abfrage für den Neustart anzeigt. Wenn *Ja*, darf der Sound nicht noch einmal abgespielt werden; ansonsten würde permanent der Berühren-Sound abgespielt.

Quelltext 3.99: Bubbles (Requirement [13.3](#)) – `update()`

```
190     if not self.pausing and self.running:
191         if self.check_bubblecollision():
192             if not self.restarting:
193                 Game.SOUND_CONTAINER["clash"].play()  #
194                 self.all_sprites.add(self.msgrestart)
195                 self.restarting = True
```

Und Schluss :-)

3.3 Moonlander

In diesem Kapitel wird ein Moonlander erstellt. Dabei will ich auf fertige Sprites verzichten und die Grafik komplett mit Grafikprimitiven erstellen (siehe Abschnitt 2.2 auf Seite 11).

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

3.3.1 Requirement 1: Standards

Requirement 1 Standardfunktionalität

1. Fenster hat eine Größe von $600 \times 800 \text{ px}$.
2. Der Hintergrund teilt sich in einen schwarzen Himmel, einer blauen Erde am rechten, oberen Rand und der Mondoberfläche.
3. Beendet wird mit der Taste `Esc` oder per Mausklick auf rote „X“.
4. Mit der Taste `R` wird ein Neustart ausgelöst.
5. Das Spiel hat eine von der fps unabhängige Ablaufgeschwindigkeit.

Requirement 1 wird in der Präambel schon festgelegt. Auch wird in `Settings` die FPS und die damit verbundene `DELTATIME` definiert. In `HORIZONT` lege ich fest, wo die Mondoberfläche aufhört und der schwarze Nachthimmel beginnt.

Quelltext 3.100: Moonlander (Requirement 1.1) – Präambel

```

1 from time import time
2
3 import pygame
4
5
6 class Settings:
7     WINDOW = pygame.Rect(0, 0, 600, 800)
8     FPS = 60
9     DELTATIME = 1.0 / FPS
10    HORIZONT = 50

```

Requirement 1.2 löse ich durch drei Klassen: `Sky`, `Moon` und `Earth`. Zunächst die Klasse `Sky`. Sie besteht aus einem recht einfachen Grundaufbau. Im Konstruktor wird ein Verweis auf das Fenster übergeben und die Größe des Himmels in Form eines `Rect`-Objektes abgespeichert. Dabei wird unten Platz für die Mondoberfläche gelassen. Die Methode `draw()` zeichnet mir ein schwarzes Rechteck an die richtige Stelle.

Quelltext 3.101: Moonlander (Requirement 1.2) – Sky

```

12 class Sky:
13     def __init__(self, screen:pygame.surface.Surface) -> None:
14         top = 0
15         left = 0
16         width = Settings.WINDOW.width
17         height = Settings.WINDOW.height - Settings.HORIZONTAL
18         self.rect = pygame.Rect(top, left, width, height)
19         self.screen = screen
20
21     def draw(self) -> None:
22         pygame.draw.rect(self.screen, "black", self.rect)

```

Völlig analog arbeitet die Klasse Moon (siehe Quelltext 3.102). Die einzigen Unterschiede sind die unterschiedlichen Positionen und die andere Farbe – hier Grau.

Quelltext 3.102: Moonlander (Requirement 1.2) – Moon

```

24 class Moon:
25     def __init__(self, screen:pygame.surface.Surface) -> None:
26         top = 0
27         left = Settings.WINDOW.height - Settings.HORIZONTAL
28         width = Settings.WINDOW.width
29         height = Settings.HORIZONTAL
30         self.rect = pygame.Rect(top, left, width, height)
31         self.screen = screen
32
33     def draw(self) -> None:
34         pygame.draw.rect(self.screen, "gray", self.rect)

```

Die Klasse Earth zeichnet mir eine blaue Kugel in das rechte obere Eck (siehe Quelltext 3.103).

Quelltext 3.103: Moonlander (Requirement 1.2) – Earth

```

37 class Earth:
38     def __init__(self, screen:pygame.surface.Surface) -> None:
39         self.radius = 80
40         left = Settings.WINDOW.right - 2*self.radius - 30
41         top = Settings.WINDOW.top + 15
42         width = 2*self.radius
43         height = 2*self.radius
44         self.rect = pygame.Rect(left, top, width, height)
45         self.screen = screen
46
47     def draw(self) -> None:
48         pygame.draw.circle(self.screen, "blue", self.rect.center, self.radius)

```

Wie üblich wird das Spiel in einer eigenen Klasse gekapselt: Game. Die drei Objekte müssen nur noch in den üblichen Aufbau von Game eingefügt werden. Dabei wird auch das Beenden und der Neustart des Spiels implementiert.

In Quelltext 3.103 wird im Konstruktor von Game Pygame initialisiert, ein Fenster erstellt, der Bildschirm des Fensters ermittelt und ein Clock-Objekt für die Deltatime-Logik (siehe Abschnitt 2.4.2 auf Seite 32) erzeugt.

Quelltext 3.104: Moonlander (Requirement 1) – Konstruktor von Game

```

51  class Game:
52      def __init__(self) -> None:
53          pygame.init()
54          self.window = pygame.Window(size=Settings.WINDOW.size, title="MyMoonlander",
55                                      position=pygame.WINDOWPOS_CENTERED)
56          self.screen = self.window.get_surface()
57          self.clock = pygame.time.Clock()

```

Der Aufbau der Methode `run()` entspricht den obigen Beispielen. Der Kern ist der Aufruf des Ereignisverwalters, das Aktualisieren der Spielobjekte und das Zeichnen der Spielobjekte; hinzu kommt die Deltatime-Logik.

Quelltext 3.105: Moonlander (Requirement 1) – Game.run()

```

59  def run(self) -> None:
60      self.restart()
61      time_previous = time()
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()

```

Auf der Ereignisverwalter sollte keine Überraschung mehr darstellen. Mit `QUIT` und `Esc` wird das Spiel beendet und mit `r` erfolgt ein Neustart.

Quelltext 3.106: Moonlander (Requirement 1) – Game.watch_for_events()

```

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          elif event.type == pygame.KEYDOWN:
77              if event.key == pygame.K_ESCAPE:
78                  self.running = False
79              elif event.key == pygame.K_r:
80                  self.restart()

```

Die Methode `update()` ist derzeit nur ein Platzhalter für zukünftige Aufgaben.

Quelltext 3.107: Moonlander (Requirement 1) – Game.update()

```

82  def update(self) -> None:
83      pass

```

In `draw()` werden die entsprechenden Methoden der Spielobjekte aufgerufen und das Fenster geflipt.

Quelltext 3.108: Moonlander (Requirement 1) – `Game.draw()`

```

85  def draw(self) -> None:
86      self.background.draw()
87      self.moon.draw()
88      self.earth.draw()
89      self.window.flip()

```

Der Neustart setzt nicht die einzelnen Spielobjekte zurück, sondern erzeugt einfach die Spielobjekte neu. Dies ist der einfachste, aber nicht immer mögliche Weg eines Neustarts.

Quelltext 3.109: Moonlander (Requirement 1) – `Game.restart()`

```

91  def restart(self) -> None:
92      self.background = Sky(self.screen)
93      self.moon = Moon(self.screen)
94      self.earth = Earth(self.screen)
95      self.running = True

```

Verbleibt der Spielaufruf selbst.

Quelltext 3.110: Moonlander (Requirement 1) – Start des Spiels

```

97  def main():
98      Game().run()
99
100 if __name__ == "__main__":
101     main()

```

Nach einem Start des Programms erscheint ein Szenario wie in Abbildung 3.17.

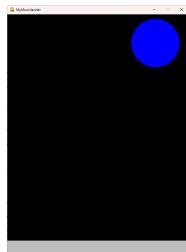


Abbildung 3.17: Moonlander – Der Hintergrund

3.3.2 Requirement 2: Mondoberfläche

Bisher ist die Mondoberfläche ein graues Rechteck. Ich möchte aber, dass eine graue Gebirgslandschaft den optischen Reiz erhöht.

Requirement 2 Mondoberfläche

Die Mondoberfläche besteht aus hintereinander angeordneten Gebirgszügen.

Im ersten Ansatz erweitere ich den Konstruktor von `Moon` um die Anzahl der Gebirgszüge. Jeder Gebirgszug wird erstmal durch ein grauen Rechteck dargestellt. Die Höhenunterschiede kommen später.

Die eigentliche Mondoberfläche (Landeplatz in Zeile 30) bleibt ein Rechteck mit der Höhe von `Settings.HORIZONT`. In `self.layers` werden die Informationen über jeden Gebirgszug als Liste abgelegt.

Ab Zeile 33 werden die Gebirgszüge (default ist 5) erstellt. Dabei wird zuerst die Farbe des Gebirges festgelegt (Zeile 34). Dabei wird vom Farbwert 180 ein Anteil abgezogen, der vom Layerindex abhängig ist. Je größer der Layerindex ist, desto mehr wird von der 180 abgezogen. Farblich bedeutet dies, dass sie dunkler wird. Je weiter also das Gebirge (der Layer) entfernt ist, desto dunkler erscheint er.

Die Höhe des Gebirges (`y`) wird errechnet, in dem von der Oberkante der Landefläche mindestens 10 px nach oben gegangen wird; dieser Wert wird noch um eine Zufallszahl zwischen 5 und 30 ergänzt, so dass die Höhen der Gebirgszüge nicht immer gleich sind. Damit die hinteren Gebirgszüge dabei immer schön herausragen, wird der Wert noch mit dem Layerindex multipliziert. Nun wird noch `draw()` angepasst (Quelltext 3.112), indem die Gebirgszüge als Rechtecke gezeichnet werden. Die Mondoberfläche sollte ungefähr wie in Abbildung 3.18 aussehen.



Abb. 3.18: Gebirge (1)

Quelltext 3.111: Moonlander (Requirement 2) – Konstruktor von `Moon`

```

25  class Moon:
26      def __init__(self, screen: pygame.surface.Surface, layer_count: int=5):
27          self.screen = screen
28          top = Settings.WINDOW.height - Settings.HORIZONT
29          self.rect = pygame.Rect(0, top,
30                                 Settings.WINDOW.width, Settings.HORIZONT) # Landeplatz
31
32          self.layers = []
33          for layer_index in range(layer_count): # Aufbau Gebirge
34              mycolor = 180 - layer_index * 20 # Vordergrund dunkler, Hintergrund heller
35              self.layers.append({"y":self.rect.top - 10 - randint(5, 30)*layer_index,
36                  "color":(mycolor, mycolor, mycolor)})

```

Quelltext 3.112: Moonlander (Requirement 2) – `Moon.draw()`

```

38  def draw(self):
39      # Landefläche
40      pygame.draw.rect(self.screen, (230, 230, 230), self.rect)
41
42      # Für jeden Gebirgszug von hinten nach vorne zeichnen + Tiefeneffekt
43      for layer in reversed(self.layers):
44          r = pygame.Rect()
45          r.top = layer["y"]
46          r.left = self.rect.left
47          r.width = self.rect.width
48          r.height = self.rect.top - r.top
49          pygame.draw.rect(self.screen, layer["color"], r)

```

Jetzt wird es Zeit für die Berggipfel. Die Grundidee ist, dass um die Oberkante des Gebirgszuges zufällige Höhenunterschiede generiert werden, die von der Höhe der Oberkante abgezogen werden.

Dazu wird im ersten Schritt der Konstruktor von `Moon` um den Übergabeparameter `peaks` erweitert. In Zeile 33 wird der Abstand zwischen zwei Höhenunterschieden berechnet und in `dist` abgelegt; auch den könnte man zufälliger streuen, aber irgendwie hatte ich dazu keine Lust.

Jetzt wird innerhalb der Schleife für jeden Gebirgszug eine Spitze oder ein Tal generiert. Die Ermittlung der Farbe (des Grautons) bleibt unverändert. In `lofPeaks` werden die Peaks als Liste von Punkten gespeichert. Der erste Punkt ist immer ganz links auf der Oberkante der Landefläche (Zeile 38). Dies ist dann der Startpunkt unseres geschlossenen Polygons. Anschließend wird mit Hilfe einer Schleife die Liste der Peaks um weitere zufällige Punkte erweitert. In Zeile 40 wird dazu ein Höhenunterschied zwischen -5 px und 10 px geraten und von der Oberkante subtrahiert. In der Zeile danach wird durch die Addition von `dist` der nächste Peak nach rechts verschoben. Ist diese innere for-Schleife abgearbeitet, steht die Liste der Höhenpunkte und kann dem entsprechendem Layer in Zeile 43 hinzugefügt werden. Vorher muss aber noch der letzte Punkt des Polygonzuges erzeugt und hinzugefügt werden.

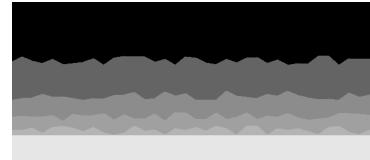


Abb. 3.19: Gebirge (2)

Quelltext 3.113: Moonlander (Requirement 2) – Konstruktor von Moon mit Peaks

```

25  class Moon:
26      def __init__(self, screen: pygame.surface.Surface, layer_count:int=5, peaks: int=35):
27          self.screen = screen
28          top = Settings.WINDOW.height - Settings.HORIZONTAL
29          self.rect = pygame.rect.Rect(0, top,
30                                      Settings.WINDOW.width, Settings.HORIZONTAL) # Landeplatz
31
32          self._layers = []
33          dist = self.rect.width // peaks          # Abstand zwischen Höhenunterschieden
34          for layer_index in range(layer_count): # Aufbau Gebirge
35              mycolor = 180 - layer_index * 20      # Vordergrund dunkler, Hintergrund heller
36              y = self.rect.top - 10 - randint(5, 30)*layer_index # Zufällige Starthöhe
37              x = self.rect.left                     # Erster Peak startet ganz links
38              lofPeaks = [(x, top)]                 # Der erste Peak als Punkt
39              for i in range(peaks):               # Die anderen Peaks des layers werden erzeugt.
40                  lofPeaks.append((x, y + randint(-5, 10))) # Zufälliger Höheunterschied
41                  x += dist                      # Der nächste Peak ist weiter rechts
42              lofPeaks.append((self.rect.right, y))   # Letzter Peak ist ganz rechts
43              lofPeaks.append((self.rect.right, top)) # Basis des Gebirgszuges
44              self._layers.append({"color":(mycolor, mycolor, mycolor),
45                      "peaks":lofPeaks})

```

Die Methode `draw()` ist nun schön einfach geworden. Für jede Gebirgskette wird `draw.-polygon()` aufgerufen, die eigentliche Arbeit ist im Konstruktor erfolgt. Das Ergebnis kann man in Abbildung 3.19 bewundern.

Quelltext 3.114: Moonlander (Requirement 2) – Moon.draw() mit Bergspitzen

```

47  def draw(self):
48      # Landefläche
49      pygame.draw.rect(self.screen, (230, 230, 230), self.rect)
50
51      # Für jeden Gebirgszug von hinten nach vorne zeichnen -> Tiefeneffekt
52      for layer in reversed(self._layers):
53          pygame.draw.polygon(
54              self.screen,
55              layer["color"],
56              layer["peaks"]
57      )

```

Zum Abschluss möchte ich noch den Bergen eine gewisse Kontur geben. Dazu wird das eine Polygon in viele aufgeteilt, wobei ein Polygon immer von einem Peak zum nächsten reicht. Zunächst fällt auf, dass die Anzahl der Peaks nun pro Gebirgszug variiert (Zeile 34); dies lässt die Züge nicht so schachbrettartig erscheinen. Natürlich muss dann auch der Abstand der Peaks neu berechnet werden (Zeile 35).



Abb. 3.20: Gebirge (3)

Um den Quelltext besser zu verstehen, habe ich das Erzeugen der Peaks und das Berechnen der entsprechenden Polygone getrennt programmiert. Etwas Neues passiert erst ab Zeile 46. Für jeden Peak werden nun vier Punkt ermittelt. Der Startpeak, der rechts davon liegende Peak, der Punkt darunter bis zur Oberfläche und auf der Oberfläche wieder nach links unter dem Startpeak. Auch wird eine dezente Grauschattierung erraten. Die vier Punkte werden als Polygonzug gemeinsam mit der passenden Farbe in die Liste `layers` abgelegt.

Quelltext 3.115: Moonlander (Requirement 2) – Konstruktor von Moon mit Kontur

```

25  class Moon:
26      def __init__(self, screen: pygame.surface.Surface, layer_count:int=5, peaks: int=35):
27          self.screen = screen
28          top = Settings.WINDOW.height - Settings.HORIZONTAL
29          self.rect = pygame.Rect(0, top,
30                                 Settings.WINDOW.width, Settings.HORIZONTAL) # Landeplatz
31
32          self._layers = []
33          for layer_index in range(layer_count): # Aufbau Gebirge
34              mypeaks = randint(peaks//2, peaks) # Anzahl variiert
35              dist = self.rect.width // mypeaks # Abstand zwischen Höhenunterschieden
36              mycolor = 180 - layer_index * 20 # Vordergrund dunkler, Hintergrund heller
37              y = self.rect.top - 10 - randint(5, 30)*layer_index # Zufällige Starthöhe
38              x = self.rect.left # Erster Peak startet ganz links
39              lofPeaks = [(x, top)] # Der erste Peak als Punkt
40              for i in range(mypeaks): # Die anderen Peaks werden erzeugt.
41                  lofPeaks.append((x, y + randint(-5, 20))) # Zufälliger Höheunterschied
42                  x += dist # Der nächste Peak ist weiter rechts
43              lofPeaks.append((self.rect.right, y)) # Letzter Peak ist ganz rechts
44              lofPeaks.append((self.rect.right, top)) # Basis des Gebirgszuges
45
46              poly = [] # Ein Polygonzug
47              for index in range(len(lofPeaks)-1):
48                  p1 = lofPeaks[index]
49                  p2 = lofPeaks[index+1]
50                  p3 = (lofPeaks[index+1][0], self.rect.top)

```

```

51         p4 = (lofPeaks[index][0], self.rect.top)
52         r = randint(-5,5)
53         c = [mc + r for mc in (mycolor, mycolor, mycolor)]
54         poly.append({"points":(p1, p2, p3, p4), "color":c})
55         self._layers.append(poly)

```

Noch ein paar leicht nachzuvollziehende Änderungen in `draw()` und der Mond ist fertig (Abbildung 3.20 auf der vorherigen Seite).

Quelltext 3.116: Moonlander (Requirement 2) – `Moon.draw()` mit Kontur

```

57     def draw(self):
58         # Landefläche
59         pygame.draw.rect(self.screen, (230, 230, 230), self.rect)
60
61         # Für jeden Gebirgszug von hinten nach vorne zeichnen -> Tiefeneffekt
62         for layer in reversed(self._layers):
63             for poly in layer:
64                 pygame.draw.polygon(
65                     self.screen,
66                     poly["color"],
67                     poly["points"])

```

Den Gebirgszug jedes Mal neu zeichnen zu lassen, ist sicherlich eine enorme Rechenzeitverschwendungen. Eine gängige Technik ist nun, das Bild einmal in ein Bitmap (`pygame.surface.Surface`) zeichnen zu lassen und dann immer das Bitmap zu blitzen. Hinweis: `rect` wird nun für das gesamte Bitmap gebraucht und wurde deshalb zuerst in `landeplatz` umbenannt. Auch kann darauf verzichtet werden `layer` und `landeplatz` als Attribute der Klasse zu definieren, da diese Informationen nach der Erstellung des Bitmaps nicht mehr gebraucht werden.

Quelltext 3.117: Moonlander (Requirement 2) – Moon als Bitmap

```

25 class Moon:
26     def __init__(self, screen: pygame.surface.Surface, layer_count:int=5, peaks: int=35):
27         self.screen = screen
28         self.surface = pygame.surface.Surface((Settings.WINDOW.width,
29                                             Settings.HORIZONTAL + layer_count*30),
30                                             pygame.SRCALPHA)
31         self.rect = self.surface.get_rect()
32         self.rect.left = Settings.WINDOW.left
33         self.rect.bottom = Settings.WINDOW.bottom
34         landingarea = pygame.rect.Rect(0, self.rect.height - Settings.HORIZONTAL,
35                                         Settings.WINDOW.width, Settings.HORIZONTAL) # Landeplatz
36
37         layers = []
38         for layer_index in range(layer_count): # Aufbau Gebirge
39             mypeaks = randint(peaks//2, peaks) # Anzahl variiert
40             dist = landingarea.width // mypeaks # Abstand zwischen Höhenunterschieden
41             mycolor = 180 - layer_index * 20 # Vordergrund dunkler, Hintergrund heller
42             y = landingarea.top - 10 - randint(5, 30)*layer_index # Zufällige Starthöhe
43             x = landingarea.left # Erster Peak startet ganz links
44             lofPeaks = [(x, landingarea.top)] # Der erste Peak als Punkt
45             for i in range(mypeaks): # Die anderen Peaks werden erzeugt.
46                 lofPeaks.append((x, y + randint(-5, 20))) # Zufälliger Höheunterschied
47                 x += dist # Der nächste Peak ist weiter rechts
48             lofPeaks.append((landingarea.right, y)) # Letzter Peak ist ganz rechts
49             lofPeaks.append((landingarea.right, landingarea.top)) # Basis des Gebirgszuges

```

```

50
51     poly = []                                # Ein Polygonzug
52     for index in range(len(lofPeaks)-1):
53         p1 = lofPeaks[index]
54         p2 = lofPeaks[index+1]
55         p3 = (lofPeaks[index+1][0], landingarea.top)
56         p4 = (lofPeaks[index][0], landingarea.top)
57         r = randint(-5,5)
58         c = [mc + r for mc in (mycolor, mycolor, mycolor)]
59         poly.append({"points":(p1, p2, p3, p4), "color":c})
60     layers.append(poly)
61
62     # Landefläche
63     pygame.draw.rect(self.surface, (230, 230, 230), landingarea)
64
65     # Für jeden Gebirgszug von hinten nach vorne zeichnen -> Tiefeneffekt
66     for layer in reversed(layers):
67         for poly in layer:
68             pygame.draw.polygon(
69                 self.surface,
70                 poly["color"],
71                 poly["points"])
72
73     def draw(self):
74         self.screen.blit(self.surface, self.rect.topleft)

```

3.3.3 Requirement 3: Erde

Die Erde als einfachen blauen Fleck? Das ist wohl zu unschön!

Requirement 3 Erde

1. *Die Erde soll einen Atmosphärenkranz bekommen.*
2. *Auf der Erde sollen Landmassen erkennbar sein.*

Zuerst wird auch die Erde als Bitmap umgebaut, so dass sich die Performance verbessert. Das Vorgehen ist analog zum Quelltext [3.117](#) auf der vorherigen Seite.

Quelltext 3.118: Moonlander (Requirement 3) – Earth als Bitmap

```

77 class Earth:
78     def __init__(self, screen:pygame.surface.Surface) -> None:
79         self.radius = 80
80         self.surface = pygame.surface.Surface(
81             (2*self.radius, 2*self.radius),
82             pygame.SRCALPHA)
83         self.rect = self.surface.get_rect()
84         self.rect.left = Settings.WINDOW.right - 200
85         self.rect.top = Settings.WINDOW.top + 50
86         self.screen = screen
87
88         pygame.draw.circle(self.surface,
89                         (30, 144, 255),
90                         (self.radius, self.radius),
91                         self.radius)
92
93     def draw(self) -> None:
94         self.screen.blit(self.surface, self.rect.topleft)

```

Nun wird der Kranz erzeugt. Die Grundidee ist, dass von innen nach außen Kreise mit immer stärkerer Transparenz gezeichnet werden. Dazu wird in der Schleife von 20 nach 1 runtergezählt. Dieser Zähler wird in Zeile 90 mit 10 multipliziert und von 210 abgezogen. Es ergibt sich als ein Folge wie $(10, 20, \dots, 200)$. Passend wird dazu der Radius dieser Kreise in Zeile 92 immer größer. Zum Schluss wird die etwas verkleinerte Erde gezeichnet.

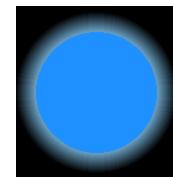


Abb. 3.21: Erde

Quelltext 3.119: Moonlander (Requirement 3.1) – Earth mit Atmosphärenkranz

```
88     for a in range(20, 1, -1):
89         pygame.draw.circle(self.surface,
90             (135, 206, 250, 210-a*10), # Steigende Transparanz
91             (self.radius, self.radius),
92             self.radius-20+a)        # Wachsender Radius
93     pygame.draw.circle(self.surface,
94         (30, 144, 255, 255),
95         (self.radius, self.radius),
96         self.radius-20)
```

Die Polygondaten für die Landmassen habe ich mir von ChatGPT erzeugen lassen (was für ein Segen!). Der Lesbarkeit wegen sind diese in einer externen Daten (`continent_polygons.py`) ausgelagert worden. Es handelt sich dabei um Liste von Listen von Punkten. Die inneren Listen repräsentieren dabei die Landmassen als geschlossene Polygonzüge. Zuerst werden die Polygondaten als Modul importiert:



Abb. 3.22: mit Kontinenten

Quelltext 3.120: Moonlander (Requirement 3.2) – Importieren der Polygondaten

```
6  from continent_polygons import continent_polygons
```

Das eigentliche Zeichnen ist dann wieder recht einfach. Dabei müssen lediglich die Koordinaten auf den Erdmittelpunkt ausgerichtet und auf die halbe Größe skaliert werden, damit sie in den Kreis reinpassen.

Quelltext 3.121: Moonlander (Requirement 3.2) – Earth mit Landmassen

```
99     for landmasse in continent_polygons:
100         poly = [(self.radius + (0.5*x), self.radius + (0.5*y)) for (x, y) in landmasse]
101         pygame.draw.polygon(self.surface, (181, 150, 116), poly)
```

Das soll mir für die Erde genug sein. Auf zum nächsten Effekt.

3.3.4 Requirement 4: Sterne

Der Weltraum ist nicht schwarz und leer.

Requirement 4 Sterne

1. *Es sollen im Hintergrund unterschiedlich große Sterne zu sehen sein.*
 2. *Die Sterne sollen sich in der Lichtintensität und Größe ändern. Dadurch soll eine Art Funkeln entstehen.*
-

Als erstes wird der Konstruktor von `Sky` um die Angabe über der Anzahl der Sterne erweitert; die Vorbelegung sind 200 Sterne. In Zeile 24 wird eine Liste für die Sterne angelegt. Danach werden in der Schleife dort entsprechend viele Einträge gemacht. Position, Größe und Farbe werden per Zufall bestimmt.

Quelltext 3.122: Moonlander (Requirement 4.1) – Sternenhimmel

```

15  class Sky:
16      def __init__(self, screen:pygame.surface.Surface, star_count: int=200) -> None:
17          top = 0
18          left = 0
19          width = Settings.WINDOW.width
20          height = Settings.WINDOW.height - Settings.HORIZONTAL
21          self.rect = pygame.rect.Rect(top, left, width, height)
22          self.screen = screen
23
24          self.stars = []                      # Sternenliste
25          for _ in range(star_count):
26              self.stars.append({"pos":(randint(2, self.rect.right-1),
27                                randint(2, self.rect.right-1)),
28                                "size":randint(1, 3),
29                                "color":randint(10, 255)})
```

In `draw()` werden die Listeneinträge für das Zeichnen der Sterne verwendet.

Quelltext 3.123: Moonlander (Requirement 4.1) – Sternenhimmel

```

31  def draw(self) -> None:
32      pygame.draw.rect(self.screen, "black", self.rect)
33      for star in self.stars:
34          pygame.draw.circle(self.screen, (255,255,star["color"]), star["pos"],
35                             star["size"])
```

Etwas interessanter wird es, das Funkeln zu erzeugen. Vorbereitend wird den Sternendaten in Zeile 29 ein Zufallswert mitgegeben, der angibt, nach wie vielen Frames eine Helligkeitsänderung erfolgen soll. Bei 60 fps sind die 3.3 – 10 sek.

Quelltext 3.124: Moonlander (Requirement 4.2) – Funkelsterne (1)

```

24      self.stars = []                      # Sternenliste
25      for _ in range(star_count):
26          self.stars.append({"pos":(randint(2, self.rect.right-1),
27                                randint(2, self.rect.right-1)),
28                                "size":randint(1, 3),
29                                "duration": randint(200, 600), # Zeit in frames
30                                "counter":0,
31                                "color":randint(10, 255)})
```

Da nun der Zustand des Spielobjektes sich im Laufe der Zeit verändert, wird die Methode `update()` gebraucht, welche die Helligkeit und Größe der Sterne neu bestimmt. In Zeile 35 wird in jedem Frame (bei jedem Aufruf) ein Zähler um 1 erhöht. Dann wird der Wert mit dem Modulo verarbeitet. Somit bleibt der Zähler immer zwischen diesen beiden Werte, damit der Zähler keinen Überlauf hat, also den Definitionsbereich von einem Integer überschreitet.

In der Schleife werden nun alle Sterne betrachtet. Hat der Zähler `counter` modulo `duration` den Wert 0, so sind seit genau `duration` viele Frames vergangen und die Farbe und Größe muss angepasst werden. Die Methode `draw()` bleibt unverändert.



Abb. 3.23: Sternenhimmel

Quelltext 3.125: Moonlander (Requirement 4.2) – Funkelsterne (2)

```
33     def update(self) -> None:
34         for star in self.stars:
35             star["counter"] = (star["counter"] + 1) % (star["duration"] + 1) # Zähler
36             if star["counter"] == 0:
37                 star["color"] = (star["color"] + randint(0, 70)) % 256
38                 star["size"] = (star["size"] + 1) % 4
```

Jetzt wird nur noch die bisher nutzlose Methode `update()` in `Game` mit dem Aufruf versehen und alles sollte funktionieren (siehe Abbildung 3.23).

Quelltext 3.126: Moonlander (Requirement 4.2) – Funkelsterne `Game.update()`

```
156     def update(self) -> None:
157         self.background.update()
```

3.3.5 Requirement 5: Landefähre

Requirement 5 Lander

1. Die Landefähre besteht aus einer Antenne, einem Crewmodul, einer Basis mit Verbindern zum Crewmodul und Landebeinen mit Tellern.
2. Mit Drücken der Taste `Space` wird ein Antriebsausstoß angezeigt.
3. Der Lander startet in der Mitte ziemlich weit oben, aber nicht direkt am oberen Rand.

Als erstes fällt auf, dass ich nicht ein Surface-Objekt verwende, sondern zwei. Die Idee dahinter ist, dass ich einen Sprite für den Lander mit und einen ohne Antrieb erstelle.

In `draw()` wird dann anhand des Attributs `thrusting` (Zeile 135) gesteuert, welche der beiden Sprites auf den Bildschirm geblättert wird. Das Zeichnen der Lander wird in der Übersichtlichkeit wegen in die Methode `create_lander()` (Zeile 134) gekapselt.

Quelltext 3.127: Moonlander (Requirement 5.1) – Konstruktor von Lander

```

126 class Lander:
127     def __init__(self, screen: pygame.surface.Surface) -> None:
128         self.screen = screen
129         self.surface = pygame.surface.Surface((90,81), pygame.SRCALPHA)
130         self.surface_thrusting = pygame.surface.Surface((90,81), pygame.SRCALPHA)
131         self.rect = self.surface.get_rect()
132         self.rect.centerx = Settings.WINDOW.centerx # horizontale Startposition
133         self.rect.top = self.rect.height # vertikale Startposition
134         self.create_lander() # Zeichnen wird ausgelagert
135         self.thrusting = False # Flag, ob beschleunigt wird

```

Die einzelnen Zeichenschritte zu erläutern ist sicherlich etwas mühselig und hätte auch keinen größeren Lerneffekt. Am einfachsten kann man den Quelltext nachvollziehen, wenn man Einzelheiten ändert und den Effekt beobachtet. Trotzdem möchte ich auf eine Sache eingehen.

Im ersten Schritt werden alle Zeichnungen auf das Surface `surface` gemacht. Dadurch erhält man ein Fähre ohne Antriebsausstoß. Ab Zeile 135 wird das Surface mit Antrieb erstellt. Dazu wird der Lander ohne Antrieb erstmal auf `surface_thrusting` mit Hilfe von `blit()` kopiert. Danach wird auf `surface_thrusting` noch eine zusätzliche Antriebsflamme gezeichnet. Nun stehen zwei Surface-Objekte zur Darstellung des Landers zur Verfügung. Beide sind in Abbildung 3.24 bzw. Abbildung 3.25 auf der nächsten Seite zu sehen.

Quelltext 3.128: Moonlander (Requirement 5.1) – Lander.create_lander()

```

137     def create_lander(self) -> None:
138         # Ein paar Abkürzungen
139         cx = self.rect.width // 2
140         cy = self.rect.height // 2
141         s = self.rect.width // 2
142         sur = self.surface
143
144         # Antenne
145         pygame.draw.line(sur, (220, 220, 220), (cx, cy - s//2), (cx, cy - s//1.2), 2)
146         pygame.draw.circle(sur, (255, 255, 255), (cx, cy - s//1.2), 3)
147
148         # Oberes Crewmodul (schmäler)
149         pygame.draw.polygon(
150             sur, (160, 160, 160),
151             [(cx - s//4, cy - s//2),
152              (cx - s//6, cy - s//3),
153              (cx + s//6, cy - s//3),
154              (cx + s//4, cy - s//2)])
155
156         # Verbinde zwischen Basis und Crewmodul
157         conn_color = (160, 160, 160)
158         pygame.draw.line(sur, conn_color, (cx - s//3, cy), (cx - s//6, cy - s//3), 2)
159         pygame.draw.line(sur, conn_color, (cx, cy), (cx, cy - s//3), 2)
160         pygame.draw.line(sur, conn_color, (cx + s//3, cy), (cx + s//6, cy - s//3), 2)
161

```

```

162     # Modulbasis (zentrale Kapsel, hellgrau)
163     pygame.draw.polygon(
164         sur, (200, 200, 200),
165         [(cx - s//3, cy),
166          (cx - s//2, cy + s//2),
167          (cx + s//2, cy + s//2),
168          (cx + s//3, cy)])
169     )
170
171     # Fenster im Modul
172     r = 5
173     window_color = (50, 50, 50)
174     pygame.draw.circle(sur, window_color, (cx, cy+(s//4)), r)
175     pygame.draw.circle(sur, window_color, (cx-(s//4), cy+(s//4)), r)
176     pygame.draw.circle(sur, window_color, (cx+(s//4), cy+(s//4)), r)
177
178     # Landbeine
179     leg_color = (100, 100, 100)
180     pygame.draw.line(sur, leg_color, (cx - s//2, cy + s//2), (cx - s, cy + s), 3)
181     pygame.draw.line(sur, leg_color, (cx + s//2, cy + s//2), (cx + s, cy + s), 3)
182     pygame.draw.line(sur, leg_color, (cx - s//4, cy + s//2), (cx - s//3, cy + s), 3)
183     pygame.draw.line(sur, leg_color, (cx + s//4, cy + s//2), (cx + s//3, cy + s), 3)
184
185     # Füße
186     feet_color = (150, 150, 150)
187     pygame.draw.circle(sur, feet_color, (cx - s + (r+2), cy + s - (r+2)), r-1)
188     pygame.draw.circle(sur, feet_color, (cx + s - (r+2), cy + s - (r+2)), r-1)
189     pygame.draw.circle(sur, feet_color, (cx - s//3 + (r-4), cy + s - (r+2)), r-1)
190     pygame.draw.circle(sur, feet_color, (cx + s//3 - (r-4), cy + s - (r+2)), r-1)
191
192     # Ausstoß
193     self.surface_thrusting.blit(sur, (0,0))
194     pygame.draw.polygon(self.surface_thrusting, (255, 140, 0), [
195         (cx - 5, cy + s//2),
196         (cx + 5, cy + s//2),
197         (cx, cy + s//2 + 20)
198     ])

```



Abbildung 3.24: Lander ohne Antrieb



Abbildung 3.25: Lander mit Antrieb

In `update()` wird das Flag `thrusting` von außen gesteuert gesetzt.

Quelltext 3.129: Moonlander (Requirement 5.1) – `Lander.update()`

```

201     def update(self, *args: Any, **kwargs: Any) -> None:
202         if "action" in kwargs.keys():
203             if kwargs["action"] == "thrust":
204                 self.thrusting = True
205             elif kwargs["action"] == "unthrust":
206                 self.thrusting = False

```

In Game muss dazu `watch_for_events()` entsprechend erweitert werden. Wird die Taste

Space gedrückt, wird der Lander in den *thrust*-Modus gesetzt. Wird sie wieder losgelassen, wird der Modus wieder ausgeschaltet.

Quelltext 3.130: Moonlander (Requirement 5.2) – Game.watch_for_events()

```

234     def watch_for_events(self) -> None:
235         for event in pygame.event.get():
236             if event.type == pygame.QUIT:
237                 self.running = False
238             elif event.type == pygame.KEYDOWN:
239                 if event.key == pygame.K_ESCAPE:
240                     self.running = False
241                 elif event.key == pygame.K_r:
242                     self.restart()
243                 elif event.key == pygame.K_SPACE:
244                     self.lander.update(action="thrust")
245             elif event.type == pygame.KEYUP:
246                 if event.key == pygame.K_SPACE:
247                     self.lander.update(action="unthrust")

```

In Lander.draw() wird nun abhängig vom *thrust*-Modus mal das eine mal das andere Surface ausgegeben.

Quelltext 3.131: Moonlander (Requirement 5.2) – Lander.draw()

```

208     def draw(self) -> None:
209         if self.thrusting:
210             self.screen.blit(self.surface_thrusting, self.rect.topleft)
211         else:
212             self.screen.blit(self.surface, self.rect.topleft)

```

Verbleibt noch den Startpunkt festzulegen, was recht simpel ist. Im Konstruktor von Lander wird die entsprechende Position bestimmt (siehe Zeile 132 und Zeile 133). Das Ergebnis sollte wie in Abbildung 3.26 auf der nächsten Seite aussehen.

Quelltext 3.132: Moonlander (Requirement 5.3) – Startposition des Lander

```

132     self.rect.centerx = Settings.WINDOW.centerx # horizontale Startposition
133     self.rect.top = self.rect.height # vertikale Startposition

```

3.3.6 Requirement 6: Gravitation und Aufsetzen

Requirement 6 Gravitation und Aufsetzen

1. Der Lander wird durch die Gravitation des Mondes mit 1.62 m/s^2 beschleunigt.
2. Berühren die Teller der Landestützen die Mondoberfläche, bleibt der Lander stehen.

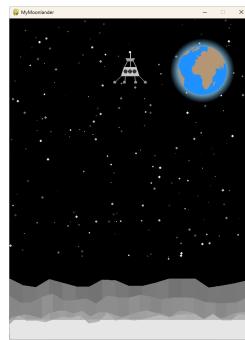


Abbildung 3.26: Moonlander – Der Lander

Dazu werden in **Settings** ab Zeile 15 einige Parameter festgelegt. Ich habe dabei sowohl die Mond- als auch die Erdbeschleunigung eingefügt. Sie sind natürlich völlig frei, den Lander auch auf der Venus oder dem Jupiter landen zu lassen.

Quelltext 3.133: Moonlander (Requirement 6.1) – Physikalische Konstanten

```

10 class Settings:
11     WINDOW = pygame.rect.Rect(0, 0, 600, 800)
12     FPS = 60
13     DELTATIME = 1.0 / FPS
14     HORIZONT = 50
15     # Physikalische Konstanten (Mondbedingungen)
16     MOON_GRAVITY = 1.62           # m/s2
17     EARTH_GRAVITY = 9.81          # m/s2
18     PIXELS_PER_METER = 10          # Skalierung 1m = 10px
19     GRAVITY = MOON_GRAVITY * PIXELS_PER_METER # = 16.2 px/s2

```

Im Konstruktor des Landers wird in Zeile 141 seine vertikale Geschwindigkeit definiert. Zu Beginn des Spieles soll sie immer 0 betragen, was unrealistisch ist, da der Lander ja mitten im Sinkflug ist, aber sei es drum.

Quelltext 3.134: Moonlander (Requirement 6.1) – Erweiterung des Konstruktors von Lander

```
141     self.velocity = 0           # Horizontale Geschwindigkeit
```

Die Methode `update()` vom `Lander` wird um die Aktion `move` erweitert. Die Berechnung der neuen Position selbst wird in der Methode `move()` gekapselt.

Quelltext 3.135: Moonlander (Requirement 6.1) – Erweiterung von `Lander.update()`

```
213     elif kwargs["action"] == "move":
214         self.move()
```

Zuerst wird die neue Geschwindigkeit abhängig von der Gravitation berechnet. Anschließend die Positionsänderung anhand der Geschwindigkeit. Wird dabei die untere Grenze unterschritten (Teller der Landestützen), wird der Lander auf der Mondoberfläche justiert und verbleibt dann auch dort.

Quelltext 3.136: Moonlander (Requirement 6.2) – `Lander.move()`

```

222     def move(self) -> None:
223         self.velocity += Settings.GRAVITY * Settings.DELTATIME
224         self.rect.top += self.velocity * Settings.DELTATIME
225         if self.rect.bottom >= Settings.WINDOW.bottom - Settings.HORIZONTAL:
226             self.rect.bottom = Settings.WINDOW.bottom - Settings.HORIZONTAL

```

Zum Schluss muss noch `Game.update()` angepasst werden.

Quelltext 3.137: Moonlander (Requirement 6) – `Game.update()`

```

263     def update(self) -> None:
264         self.background.update()
265         self.lander.update(action="move")      # Soll sich bewegen!

```

3.3.7 Requirement 7: Gegenschub

Requirement 7 Gegenschub

Wird mit Hilfe von `Space` Gegenschub ausgelöst, soll dieser Schub sich auf die Sinkgeschwindigkeit des Landers auswirken. Der Gegenschub soll -3 m/s^2 betragen.

Der Gegenschub ist recht willkürlich mit -3 m/s^2 festgelegt. Das negative Vorzeichen kommt daher, da dieser Schub ja der Mondgravitation genau entgegengesetzt wirken soll.

Quelltext 3.138: Moonlander (Requirement 7) – Größe des Gegenschubs

```

20     THRUST = -3 * PIXELS_PER_METER          # = 30.0 px/s2

```

In `move()` wird nun der Gegenschub in die Geschwindigkeitsberechnung einberechnet. Dazu wird zuerst abgefragt, ob der Gegenschub durch `Space` ausgelöst wird.

Quelltext 3.139: Moonlander (Requirement 7) – Anpassung von `Lander.move()`

```

223     def move(self) -> None:
224         if self.thrusting:
225             self.velocity += Settings.THRUST * Settings.DELTATIME
226             self.velocity += Settings.GRAVITY * Settings.DELTATIME
227             self.rect.top += self.velocity * Settings.DELTATIME
228             if self.rect.bottom >= Settings.WINDOW.bottom - Settings.HORIZONTAL:
229                 self.rect.bottom = Settings.WINDOW.bottom - Settings.HORIZONTAL

```

Und schon kann durch den Gegenschub die Sinkgeschwindigkeit vom Spieler manipuliert werden.

3.3.8 Requirement 8: Treibstoff

Requirement 8 Treibstoff

1. Der Lander hat einen begrenzten Treibstoffvorrat.
2. Je nach Schwierigkeitsgrad gibt es unterschiedliche Vorräte.
3. Der Verbrauch beträgt 20 Einheiten pro Sekunde.
4. Ist der Vorrat leer, kann kein Gegenschub mehr erzeugt werden.

Zunächst werden in `Settings` die Spielkonstanten definiert. `THRUST` ist der Gegenschub, aber nicht in der Maßeinheit m/s^2 , sondern px/s^2 . Die möglichen Treibstoffvorräte für Requirement 8.2 werden in Zeile 22 in dem Dictionary `LEVEL` abgelegt.

Quelltext 3.140: Moonlander (Requirement 8) – Vorbereitungen in `Settings`

```
21  THRUST = -3 * PIXELS_PER_METER           # = 30.0 px/s2
22  LEVEL = {"easy":sys.maxsize, "fair":500, "hard":450, "ai":380} #
```

Im Konstruktor von `Lander` wird der Starttreibstoffvorrat festgelegt und der aktuelle Treibstoffvorrat im Attribut `fuel` mit dem Starttreibstoffvorrat.

Quelltext 3.141: Moonlander (Requirement 8) – Anpassung im Konstruktor von `Lander`

```
145  self.fuel_initial = Settings.LEVEL["fair"] # Starttreibstoff
146  self.fuel = self.fuel_initial             # Aktueller Treibstoff
147  self.fuel_consumption = 20                # Treibstoffverbrauch pro Sekunde
```

In Zeile 236 wird nun vor der Berechnung des Gegenschub überprüft, ob überhaupt noch Treibstoff im Tank ist und in Zeile 238 wird der verbrauchte Treibstoff aus dem Tank entfernt. Falls nichts mehr im Tank ist, wird muss der Gegenschub-Modus ausgeschaltet werden und, um negativen Treibstoffvorrat zu verhindern, der Vorrat auf 0 gesetzt werden.

Quelltext 3.142: Moonlander (Requirement 8) – `Lander.move()`

```
235  def move(self) -> None:
236      if self.thrusting and self.fuel > 0:                                # Treibstoff übrig?
237          self.velocity += Settings.THRUST * Settings.DELTATIME
238          self.fuel -= self.fuel_consumption * Settings.DELTATIME        # Treibstoffverbrauch
239          if self.fuel < 0:
240              self.thrusting = False
241              self.fuel = 0
242          self.velocity += Settings.GRAVITY * Settings.DELTATIME
243          self.rect.top += self.velocity * Settings.DELTATIME
244          if self.rect.bottom >= Settings.WINDOW.bottom - Settings.HORIZONT:
245              self.rect.bottom = Settings.WINDOW.bottom - Settings.HORIZONT
```

Damit wir überprüfen können, ob der Treibstoffvorrat richtig startet, sich bei Gegenschub richtig reduziert und der Schub sich bei leerem Tank ausschaltet, habe ich in `Lander.draw()` einen `print()` in Zeile 233 eingefügt.

Quelltext 3.143: Moonlander (Requirement 8) – `Lander.draw()`

```

228     def draw(self) -> None:
229         if self.thrusting:
230             self.screen.blit(self.surface_thrusting, self.rect.topleft)
231         else:
232             self.screen.blit(self.surface, self.rect.topleft)
233         print(self.fuel)      # Nur vorübergehend

```

Probieren Sie es aus, es müsste klappen!

3.3.9 Requirement 9: Statusanzeige

Requirement 9 Statusanzeige

1. Für den Lander wird eine separate Statusanzeige benötigt.
2. Geschwindigkeit und Höhe werden mit Maßeinheiten in Textform aus-gegeben.
3. Ist der Gegenschub aktiv, wird ein farbiger Balken angezeigt.
4. Der Treibstoffvorrat wird als Verlaufsbalken angezeigt.

Alle wesentliche Änderungen dazu finden in der Klasse `Lander` statt. Da ich die Position der separaten Statusanzeige von der Position des Hauptfensters abhängig machen möchte, muss die Signatur des Konstruktor geändert werden. Anstelle eines `Surface`-Objektes wird nun in Zeile 135 ein `Window`-Objekt übergeben.

Quelltext 3.144: Moonlander (Requirement 9) – `Lander.draw()`

```

134 class Lander:
135     def __init__(self, window: pygame.window.Window) -> None: #
136         self.screen = window.get_surface()
137         self.surface = pygame.surface.Surface((90, 81), pygame.SRCALPHA)
138         self.surface_thrusting = pygame.surface.Surface((90, 81), pygame.SRCALPHA)
139         self.rect = self.surface.get_rect()
140         self.rect.centerx = Settings.WINDOW.centerx # horizontale Startposition
141         self.rect.top = self.rect.height # vertikale Startposition
142         self.create_lander() # Zeichnen wird ausgelagert
143         self.thrusting = False # Flag, ob beschleunigt wird
144         self.velocity = 0 # Horizontale Geschwindigkeit
145         self.fuel_initial = Settings.LEVEL["fair"] # Starttreibstoff
146         self.fuel = self.fuel_initial # Aktueller Treibstoff
147         self.fuel_consumption = 20 # Treibstoffverbrauch pro Sekunde
148         self.create_status_window(window)

```

Das separate Fenster wird in `create_status_window()` erzeugt. Dort wird zunächst ein passend großes Fenster erstellt und das `Surface`-Objekt dazu ermittelt. Ich möchte das Statusfenster rechts neben dem Hauptfenster an der Oberkante ausgerichtet positionieren. Dazu nehme ich die obere Kante des Hauptfensters und weise diesen Wert der oberen Kante des Statusfenster zu. Danach nehme ich die linke Kante des Hauptfensters, addiere die Breite des Hauptfenster dazu, um die rechte Kante zu haben und verschaffe mir noch 10 px Abstand.

Quelltext 3.145: Moonlander (Requirement 9) – Lander.create_status_window()

```

214     def create_status_window(self, window: pygame.window.Window) -> None:
215         self.status_window = pygame.Window(size=(300, 100), title="Status")
216         self.status_screen = self.status_window.get_surface()
217         top = window.position[1]
218         left = window.position[0] + Settings.WINDOW.width + 10
219         self.status_window.position = (left, top)

```

Die Methode `draw()` wird in der letzten Zeile um den Aufruf von `draw_status()` erweitert. Dadurch wird bei jedem Aufruf von `draw()` nicht nur der Lander im Hauptfenster, sondern auch das Statusfenster neu gezeichnet.

Quelltext 3.146: Moonlander (Requirement 9) – Lander.draw()

```

236     def draw(self) -> None:
237         if self.thrusting:
238             self.screen.blit(self.surface_thrusting, self.rect.topleft)
239         else:
240             self.screen.blit(self.surface, self.rect.topleft)
241         self.draw_status()

```

In `draw_status()` wird zunächst das Fenster mit schwarzer Farbe ausgefüllt. Dann folgt ab Zeile 246 die Statusanzeige von Höhe und Treibstoff als Textausgabe. Ab Zeile 257 werden zwei Balken angezeigt. Der erste Balken wird nur eingeblendet, wenn die Mondfähre gerade Gegenschub auslöst. Der zweite Balken besteht aus zwei Rechtecken. Der graue Balken wird über die Breite des Fenters gemalt und grüne nur anteilig von links, abhängig vom Treibstoffvorrat.

Quelltext 3.147: Moonlander (Requirement 9) – Lander.draw_status()

```

243     def draw_status(self) -> None:
244         self.status_screen.fill("black")
245
246         # Textausgabe
247         font = pygame.font.SysFont("Consolas", 14)
248         labels = f"Geschwindigkeit(px/s):"
249         labels += "\nHöhe(px):"
250         values = f"{self.velocity:>7.0f}"
251         values += f"\n{-1*(self.rect.bottom - Settings.WINDOW.bottom - Settings.HORIZONTAL)}:>7.0f"
252         text_labels = font.render(labels, True, "white")
253         text_values = font.render(values, True, "white")
254         self.status_screen.blit(text_labels, (5, 10))
255         self.status_screen.blit(text_values, (230, 10))
256
257         # Balkenanzeige
258         if self.thrusting:
259             pygame.draw.rect(self.status_screen, (255, 140, 0), (5, 46, 290, 20))
260             pygame.draw.rect(self.status_screen, "grey", (5, 70, 290, 20))
261             ratio = int(290 * self.fuel / self.fuel_initial)
262             pygame.draw.rect(self.status_screen, "green", (5, 70, ratio, 20))

```

Verbleiben noch einige Anpassungen in Game wegen der Ausgabe in mehreren Fenstern. Da wäre das Event-Handling. Wenn in Pygame mehrere Fenster geöffnet werden, muss

das Event `pygame.WINDOWCLOSE` verarbeitet werden (Zeile 304). Das Flag der Hauptprogrammschleife muss auf `False` gesetzt werden und das Fenster des Events muss mit dem `destroy()` manuell zerstört werden.

Quelltext 3.148: Moonlander (Requirement 9) – `Game.watch_for_events()`

```

300  def watch_for_events(self) -> None:
301      for event in pygame.event.get():
302          if event.type == pygame.QUIT:
303              self.running = False
304          elif event.type == pygame.WINDOWCLOSE: # Neu wegen 2 Fenster!
305              self.running = False
306              event.window.destroy()
307          elif event.type == pygame.KEYDOWN:
308              if event.key == pygame.K_ESCAPE:
309                  self.running = False
310              elif event.key == pygame.K_r:
311                  self.restart()
312              elif event.key == pygame.K_SPACE:
313                  self._lander.update(action="thrust")
314          elif event.type == pygame.KEYUP:
315              if event.key == pygame.K_SPACE:
316                  self._lander.update(action="unthrust")

```

In `restart()` wird noch in Zeile 333 der Aufruf des Konstruktor angepasst.

Quelltext 3.149: Moonlander (Requirement 9) – `Game.restart()`

```

333  self._lander = Lander(self.window)      # wegen Statusfenster

```

Das Ganze sieht dann wie in Abbildung 3.27 auf der nächsten Seite aus.

3.3.10 Requirement 10: Spielende und Neustart

Requirement 10 Spielende und Neustart

1. Landet die Mondfähre mit einer Geschwindigkeit $< 5 \text{ px/s}$, ist die Fähre sicher gelandet.
2. Landet sie mit einer schnelleren Geschwindigkeit, gilt sie als zerstört.
3. Der Anwender wird gefragt, ob er das Spiel beenden (`B`) oder neu starten (`N`) möchte.

Bereiten wir Requirement 10.3 dadurch vor, dass wir die Darstellung der Nachfrage in die simple Klasse `Question` kapseln. Es wird einfach ein `Surface`-Objekt mit dem passenden Text erstellt und passend positioniert. In `draw()` wird dieses `Surface`-Objekt dann einfach über die Mondoberfläche unten gelegt.

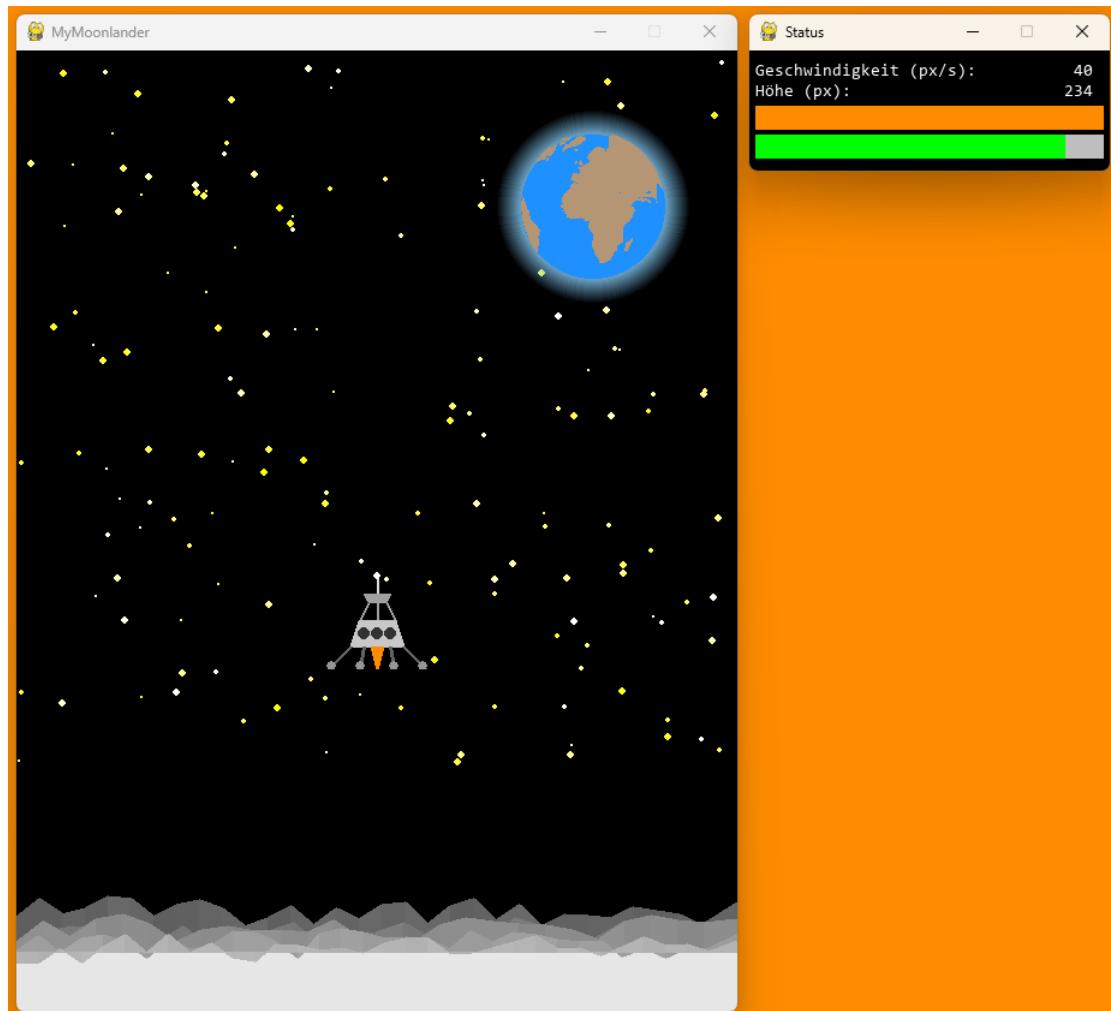


Abbildung 3.27: Moonlander – Jetzt mit Statusfenster

Quelltext 3.150: Moonlander (Requirement 10) – Question

```

316 class Question:
317     def __init__(self, screen:pygame.surface.Surface) -> None:
318         self.font = pygame.font.Font(None, 24)
319         self.screen = screen
320         self.surface = self.font.render("(B)eenden oder (N)eustart?", True, "red")
321         self.rect = self.surface.get_rect()
322         self.rect.centerx = Settings.WINDOW.centerx
323         self.rect.bottom = Settings.WINDOW.bottom - 10
324
325     def draw(self) -> None:
326         self.screen.blit(self.surface, self.rect.topleft)

```

Und wie erfolgt das Auslösen des Beendens oder des Neustarts? Dazu gibt es viele Möglichkeiten. Ich habe mich hier für *Ereignisse* mit Hilfe von `pygame.event.Event()` entschieden. Die Grundidee ist, dass das Berühren des Mondbodens ein Ereignis auslöst: `LANDED`, wenn die Sinkgeschwindigkeit klein genug ist, ansonsten `CRASHED`.

Quelltext 3.151: Moonlander (Requirement 10) – MyEvents

```

11 class MyEvents: #  

12     LANDED = pygame.USEREVENT + 1  

13     CRASHED = pygame.USEREVENT + 2

```

Dazu muss `watch_for_events()` umgeschrieben werden. In der Zeile 357 und Zeile 360 werden die beiden Ereignisse abgefangen. In beiden Fällen wird das neue Flag `landing` auf `False` gesetzt. Dadurch kann ich feststellen, ob ich beispielsweise überhaupt noch Schub auslösen darf oder die Rückfrage nach Beenden oder Neustart angezeigt werden soll. Auch wird an den `Lander` ein `update()` abgesetzt, so dass er ebenfalls von seinem neuen Zustand erfährt; z.B., um im Statusfenster eine passende Ausgabe zu machen.

In Zeile 364 wird deshalb zuerst abgefragt, ob ich überhaupt noch im Landeanflug bin, bevor der Schub ausgelöst wird.

Die Antworten auf die Rückfragen werden ab der Zeile 368 und ab der Zeile 370 verarbeitet. Wurde `B` gedrückt, wird einfach das Flag der Hauptprogrammschleife auf `False` gesetzt; wurde `N` gedrückt, wird der Neustart durch Aufruf von `restart()` ausgelöst.

Quelltext 3.152: Moonlander (Requirement 10) – Game.watch_for_events()

```

350     def watch_for_events(self) -> None:
351         for event in pygame.event.get():
352             if event.type == pygame.QUIT:
353                 self.running = False
354             elif event.type == pygame.WINDOWCLOSE:
355                 self.running = False
356                 event.window.destroy()
357             elif event.type == MyEvents.LANDED: #  

358                 self.landing = False
359                 self.lander.update(mode="landed", velocity=event.velocity)
360             elif event.type == MyEvents.CRASHED: #  

361                 self.landing = False
362                 self.lander.update(mode="crashed", velocity=event.velocity)
363             elif event.type == pygame.KEYDOWN:
364                 if self.landing: #

```

```

365             if event.key == pygame.K_SPACE:
366                 self.lander.update(action="thrust")
367             else:
368                 if event.key == pygame.K_b:          #
369                     self.running = False
370                 elif event.key == pygame.K_n:      #
371                     self.restart()
372                 if event.key == pygame.K_ESCAPE:
373                     self.running = False
374                 elif event.type == pygame.KEYUP:
375                     if event.key == pygame.K_SPACE:
376                         self.lander.update(action="unthrust")

```

In `Game` wird das Attribut `landing` eingefügt, welches sich merkt, ob gerade gelandet wird und wir den Mondboden bereits berühren.

Quelltext 3.153: Moonlander (Requirement 10) – `Game.landing`

```

334     self.landing = True

```

Und `restart()` wird in Zeile 148 um das Zurücksetzen des Flags `landing` erweitert.

Quelltext 3.154: Moonlander (Requirement 10) – `Game.restart()`

```

392     def restart(self) -> None:
393         self.landing = True                      #
394         self.background = Sky(self.screen)
395         self.moon = Moon(self.screen)
396         self.earth = Earth(self.screen)
397         self.lander = Lander(self.window)        # wegen Statusfenster
398         self.question = Question(self.screen)
399         self.running = True

```

Requirement 10.1 und Requirement 10.2 werden in der neuen Methode `check_landing()` in `Lander` umgesetzt. Erreicht der Lander den Mondboden, wird die Geschwindigkeit überprüft. Ist sie zu schnell, wird das Event `CRASHED` ausgelöst, ansonsten `LANDED`. Die Verarbeitung der Ereignisse selbst fand ja oben in `watch_for_events()` (Quelltext 3.152 auf der vorherigen Seite) statt.

Quelltext 3.155: Moonlander (Requirement 10) – `Lander.check_landing()`

```

307     def check_landing(self) -> None:
308         if self.rect.bottom >= Settings.WINDOW.bottom - Settings.HORIZONTAL:
309             if self.velocity > 5:
310                 evt = pygame.event.Event(MyEvents.CRASHED, vvelocity=self.velocity)
311                 pygame.event.post(evt)
312             else:
313                 evt = pygame.event.Event(MyEvents.LANDED, vvelocity=self.velocity)
314                 pygame.event.post(evt)

```

Der Aufruf von `check_landing()` muss nun noch der Methode `update()` von `Game` hinzugefügt werden.

Quelltext 3.156: Moonlander (Requirement 10) – `Game.update()`

```

378     def update(self) -> None:
379         self.background.update()
380         self.lander.update(action="move")           # Soll sich bewegen!
381         self.lander.check_landing()

```

Verbleiben noch einige Anpassungen im Lander. Zunächst wird in Zeile 148 das Attribut `mode` eingefügt, welches sich merkt, in welchem der drei Status sich die Mondfahre gerade befindet: `landing`, `landed` oder `crashed`.

Quelltext 3.157: Moonlander (Requirement 10) – `Lander.mode`

```

138 class Lander:
139     def __init__(self, window: pygame.window.Window) -> None: #
140         self.screen = window.get_surface()
141         self.surface = pygame.surface.Surface((90,81), pygame.SRCALPHA)
142         self.surface_thrusting = pygame.surface.Surface((90,81), pygame.SRCALPHA)
143         self.rect = self.surface.get_rect()
144         self.rect.centerx = Settings.WINDOW.centerx # horizontale Startposition
145         self.rect.top = self.rect.height               # vertikale Startposition
146         self.create_lander()                         # Zeichnen wird ausgelagert
147         self.create_lander_thrusting()               # Zeichnen mit Ausstoß
148         self.mode = "landing"                      # "landing", "landed" oder "crashed"
149         self.thrusting = False                      # Flag, ob beschleunigt wird
150         self.velocity = 0                           # Horizontale Geschwindigkeit
151         self.fuel_initial = Settings.LEVEL["fair"] # Starttreibstoff
152         self.fuel = self.fuel_initial              # Aktueller Treibstoff
153         self.fuel_consumption = 20                 # Treibstoffverbrauch pro Sekunde
154         self.create_status_window(window)

```

Gesetzt bzw. verändert wird dieses Attribut in `update()` ab Zeile 243. Ist dabei der Boden berührt worden – also `landed` oder `crashed` –, wird der Schub abgestellt.

Quelltext 3.158: Moonlander (Requirement 10) – `Lander.update()`

```

234     def update(self, *args: Any, **kwargs: Any) -> None:
235         if "action" in kwargs.keys():
236             if kwargs["action"] == "thrust":
237                 self.thrust(True)
238             elif kwargs["action"] == "unthrust":
239                 self.thrust(False)
240             elif kwargs["action"] == "move":
241                 if self.mode == "landing":
242                     self.move()
243             if "mode" in kwargs.keys():
244                 self.mode = kwargs["mode"]
245                 if self.mode in ("landed", "crashed"):
246                     self.thrust(False)

```

Die Statusausgabe wird nun um das Spielende erweitert. Das Aussehen kann in Abbildung 3.28 auf Seite 204 betrachtet werden.

Quelltext 3.159: Moonlander (Requirement 10) – `Lander.draw_status()`

```

261     def draw_status(self) -> None:
262         self.status_screen.fill("black")
263

```

```

264     # Textausgabe
265     font = pygame.font.SysFont("Consolas", 14, bold=True)
266     labels = f"Geschwindigkeit(px/s):"
267     labels += "\nHöhe(px):"
268     values = f"{self.velocity:>7.0f}"
269     values += f"\n-{1*(self.rect.bottom - Settings.WINDOW.bottom - Settings.HORIZONTAL)}:>7.0f"
270     text_labels = font.render(labels, True, "white")
271     text_values = font.render(values, True, "white")
272     if self.mode == "landed":
273         text_mode = font.render(f"Status:{self.mode}", True, "green")
274     elif self.mode == "crashed":
275         text_mode = font.render(f"Status:{self.mode}", True, "red")
276     else:
277         text_mode = font.render(f"Status:{self.mode}", True, "white")
278     text_mode_rect = text_mode.get_rect(top=100)
279     text_mode_rect.left = self.status_screen.get_rect().centerx - text_mode_rect.centerx
280     self.status_screen.blit(text_labels, (5, 10))
281     self.status_screen.blit(text_values, (230, 10))
282     self.status_screen.blit(text_mode, text_mode_rect.topleft)
283
284     # Balkenanzeige
285     if self.thrusting:
286         pygame.draw.rect(self.status_screen, (255, 140, 0), (5, 46, 290, 20))
287         pygame.draw.rect(self.status_screen, "grey", (5, 70, 290, 20))
288         ratio = int(290 * self.fuel / self.fuel_initial)
289         pygame.draw.rect(self.status_screen, "green", (5, 70, ratio, 20))
290     self.status_window.flip()

```

3.3.11 Requirement 11: Autopilot

Requirement 11 Autopilot

Mit **H** wird der Autopilot ein- bzw. ausgeschaltet.

Zunächst etwas, was überhaupt nichts mit Requirement 11 zu tun hat. Ich möchte, dass sich die physikalischen Größen noch etwas mehr den tatsächlichen Werten anpassen. Der Schub wird auf -2.1 m/s^2 festgelegt und die sichere Landegeschwindigkeit auf 2.5 m/s. Laut NASA-Dokumentation landete Apollo 11 mit 0.7 m/s. Der NASA-Zielwert war 1 m/s und der akzeptable Bereich lag zwischen 0.5 und 2.5 m/s. Die Grenze der Belastbarkeit war mit 3 m/s erreicht. Ein Wert unter 0.5 m/s wäre mit einem unnötigen Treibstoffverbrauch verbunden gewesen.

Quelltext 3.160: Moonlander (Requirement 11) – Einige Konstanten

```

26 THRUST = -2.1 * PIXELS_PER_METER          # = 21 px/s2
27 SAVE_SPEED_LANDING = 2.5 * PIXELS_PER_METER # Sichere Landegeschwindigkeit in px/s

```

So, jetzt zu Requirement 11: In `watch_for_events()` wird der Tastendruck **H** abgefangen und an den **Lander** weitergeleitet.

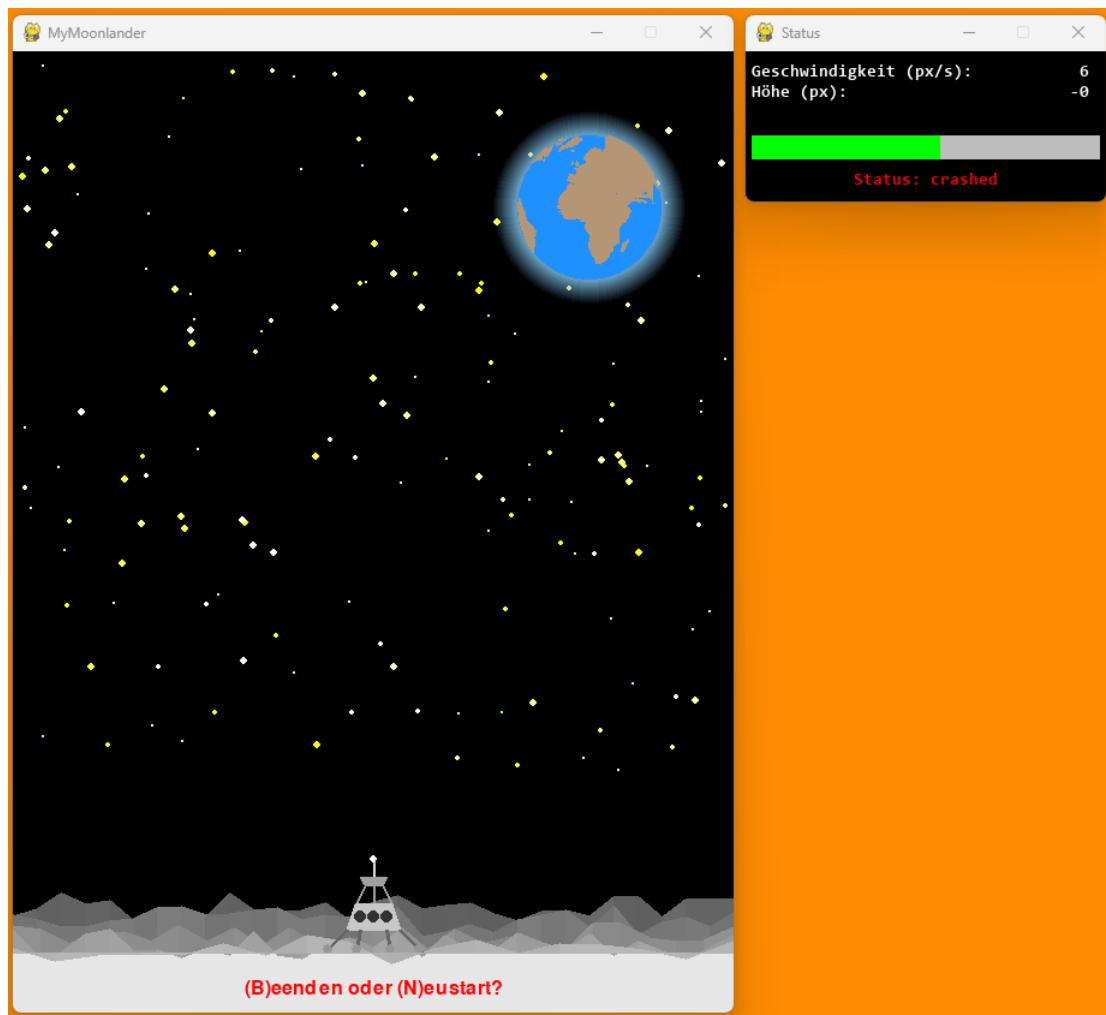


Abbildung 3.28: Moonlander – Beenden oder Neustart?

Quelltext 3.161: Moonlander (Requirement 11) – Erweiterung von `watch_for_events()`

```

383     if self.landing:
384         if event.key == pygame.K_SPACE:
385             self.lander.update(action="thrust")
386         elif event.key == pygame.K_h:
387             self.lander.update(action="toggle_ai")
388     else:

```

Im Konstruktor von `Lander` wird das Flag `ai` definiert und mit `False` initialisiert – wobei `ai` hier ein etwas übertriebener Anspruch ist ;-)

Quelltext 3.162: Moonlander (Requirement 11) – Erweiterung von `Lander.__init__()`

```

153     self.fuel_consumption = 20          # Treibstoffverbrauch pro Sekunde
154     self.ai = False                  # KI-Flag
155     self.create_status_window(window)

```

Die Methode `update()` wird ebenfalls erweitert. Ab Zeile 241 wird das Flag `ai` ein- bzw. ausgeschaltet. Wird es ausgeschaltet, muss ein ggf. vom Autopiloten ausgelöster Schub gestoppt werden. In Zeile 247 wird nun abgefragt, ob der Autopilot aktiv ist und ggf. dem Autopilot die Steuerung übergebenen.

Quelltext 3.163: Moonlander (Requirement 11) – Erweiterung von `Lander.update()`

```

235     def update(self, *args: Any, **kwargs: Any) -> None:
236         if "action" in kwargs.keys():
237             if kwargs["action"] == "thrust":
238                 self.thrust(True)
239             elif kwargs["action"] == "unthrust":
240                 self.thrust(False)
241             elif kwargs["action"] == "toggle_ai":          # KI an/aus
242                 self.ai = not self.ai
243                 if not self.ai:
244                     self.thrust(False)
245             elif kwargs["action"] == "move":
246                 if self.mode == "landing":
247                     if self.ai > 0:                      # KI aktiv?
248                         self.controller()
249                         self.move()
250             if "mode" in kwargs.keys():
251                 self.mode = kwargs["mode"]
252                 if self.mode in ("landed", "crashed"):
253                     self.thrust(False)

```

Bevor wir nun in die Programmierung der Steuerung eintauchen, müssen wir ein wenig mit physikalischen Formeln rumspielen.

Die Formel für die Endgeschwindigkeit im freien Fall ist:

$$v = \sqrt{2 \cdot g \cdot h} \quad (3.1)$$

Diese Gleichung liefert uns bei einer Gravitationskraft g und einer Fallhöhe h die Endgeschwindigkeit v , sofern die Startgeschwindigkeit 0 m/s betrug. Wir wollen aber gar

nicht wissen, wie die Endgeschwindigkeit ist, sondern wir sind eigentlich auf die Höhe h scharf. Ab welcher Höhe müssen wir mit Gegenschub reagieren, um bei unserer Zielgeschwindigkeit anzukommen? Also stellen wir Gleichung 3.1 nach h um:

$$\begin{aligned} v &= \sqrt{2 \cdot g \cdot h} \quad \| x^2 \\ v^2 &= 2 \cdot g \cdot h \quad \| : (2 \cdot g) \\ \frac{v^2}{2 \cdot g} &= h \end{aligned} \quad (3.2)$$

Wir haben es aber nicht mehr nur mit der Gravitation des Mondes, sondern auch mit dem Gegenschub der Fähre zu tun haben. Dabei gilt:

$$acc = g_{Mond} + acc_{Lander} \quad (3.3)$$

Man beachte dabei, dass das Vorzeichen von acc_{Lander} dem des Mondes g_{Mond} entgegengesetzt ist – also negativ ist. Setzen wir also die Gleichung 3.3 in Gleichung 3.2 ein:

$$\begin{aligned} h &= \frac{v^2}{2 \cdot acc} \quad \| \leftarrow 3.3 \\ h &= \frac{v^2}{2 \cdot (g_{Mond} + acc_{Lander})} \end{aligned} \quad (3.4)$$

Und Gleichung 3.4 kann auch schon als Basis für unserer Programmierung dienen. Zuerst wird ab Zeile 256 überprüft, ob ich überhaupt noch im Landeanflug bin. Falls nicht, wird jeder Schub abgestellt und ich bin fertig, da nichts mehr zu tun ist.

In Zeile 260 wird die Nettobeschleunigung aus Gleichung 3.3 berechnet und anschließend die Zielgeschwindigkeit festgelegt. Diese ist mit 50 % weit genug von der Maximalbelastung entfernt. In Zeile 262 wird nun überprüft, ob die aktuelle Geschwindigkeit schon kleiner als diese sichere Geschwindigkeit ist. Wenn ja, ist nicht zu tun, außer, dass der Schub abgestellt werden muss.

Jetzt wird passend zu Gleichung 3.4 die Distanz zum Boden ausgerechnet, bei welcher der Gegenschub beginnen muss. In Zeile 268 wird dann der Gegenschub ausgelöst oder abgeschaltet. Alles klar?

Der Lander ist damit hier für dieses Skript zu Ende programmiert und sollte bei sicherer Landung mit dem Autopiloten wie in Abbildung 3.29 auf der nächsten Seite aussehen.

Quelltext 3.164: Moonlander (Requirement 11) – Lander.controler()

```
255 def controler(self):  
256     if self.mode in ("landed", "crashed"):           # Landevorgang beendet?  
257         self.thrust(False)  
258         return  
259  
260     acc = -1 * (Settings.THRUST + Settings.GRAVITY)    # Netto-Beschleunigung  
261     v_save = Settings.SAVE_SPEED_LANDING * 0.5          # 50% Puffer  
262     if self.velocity <= v_save:                          # schon langsam?  
263         self.thrust(False)  
264         return  
265  
266     brake_distance = self.velocity ** 2 / (2 * acc)  
267     ground_distance = (Settings.WINDOW.height - 50) - self.rect.bottom  
268     self.thrust(ground_distance <= brake_distance)    # benötigte Bremsweg >= Resthöhe?
```

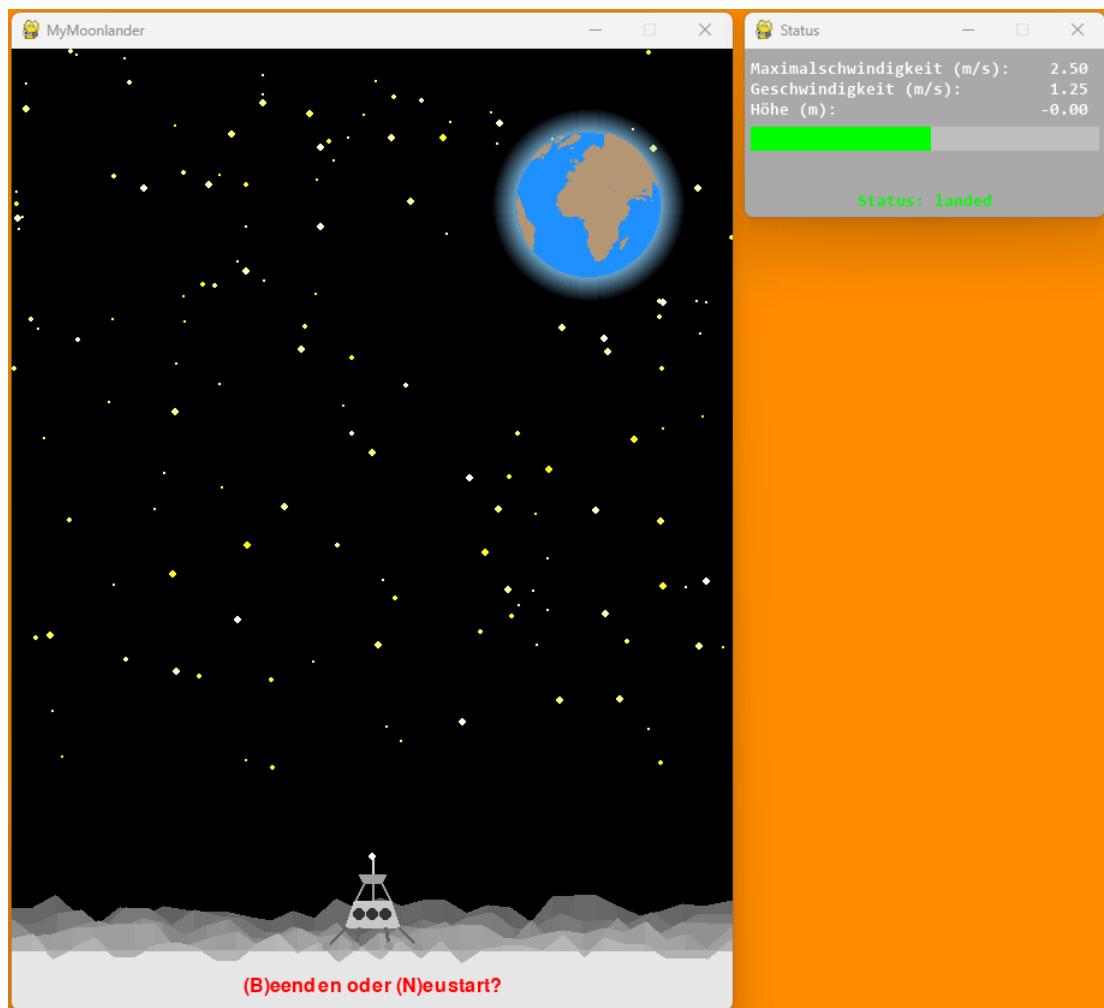


Abbildung 3.29: Moonlander – Autopilot

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)	25
2.13	Bitmaps positionieren (Angreifer, Version 2)	26
2.14	Bitmaps positionieren (Angreifer, Version 3)	27
2.15	Elemente eines <code>Rect</code> -Objekts	28
2.16	Bitmaps bewegen, Version 1.0	30
2.17	Der Verteidiger bewegt sich und prallt ab	32
2.18	Nicht normalisierte Bewegung	35
2.19	Vergleich des Positionsfehlers von $1/fps$ und <code>pygame.clock.tick()</code>	38
2.20	Vergleich des Positionsfehlers von <code>Rect</code> und <code>FRect</code>	40
2.21	Vergleich der Positionsfehler mit unterschiedlichen Genauigkeiten	41
2.22	Normalisierte Bewegung mit $1/fps$	42
2.23	Normalisierte Bewegung mit <code>pygame.clock.tick()</code>	42
2.24	Normalisierte Bewegung mit <code>pygame.clock.tick()</code> (float)	43
2.25	Normalisierte Bewegung mit <code>time.time()</code>	43
2.26	Textausgabe mit Fonts	60
2.27	Fontliste	63
2.28	Beispiel für eine Spritelib	67
2.29	Bedeutung der Angaben in Spritelib	69
2.30	Textausgabe mit Bitmaps	72
2.31	Kollisionserkennung mit Rechtecken	74
2.32	Kollisionserkennung mit Kreisen	75
2.33	Vier Sprites	76
2.34	Rechtecksprüfung (Montage)	76
2.35	Kreisprüfung (Montage)	76
2.36	Maskenprüfung (Montage)	76

2.37 Feuerball ohne Zeitsteuerung	82
2.38 Feuerball mit Zeitsteuerung	86
2.39 Animation einer Katze: Einzelsprites	89
2.40 Animation einer Explosion: Einzelsprites	95
2.41 Mausaktionen	97
2.42 Sound: Stereoeffekt	107
2.43 Dirty Sprite – Demo-Spiel	114
2.44 Performancevergleich mit Testkonfiguration 2	121
2.45 Testkonfiguration 2 mit privatem PC	122
2.46 Testkonfiguration 2 mit Surface Pro	122
2.47 Testkonfiguration 2 mit privatem PC und reduzierter fps	122
2.48 Bestätigung der Unterschiede	122
2.49 Selbst erstellte Events	127
3.1 Pong: der Hintergrund	135
3.2 Pong: mit Schläger, Ball und Punktstand	143
3.3 Pong: Schlägerfarbe markiert KI-Modus (links KI, rechts manuell)	145
3.4 Pong: Hilfe-Bildschirm	150
3.5 Bubbles: Hintergrundbild (aquarium.png)	152
3.6 Blase	154
3.7 Bubbles: Die Blasen haben beim Start einen Mindestabstand	157
3.8 Bubbles: Die Blasen sind gewachsen/verwachsen	160
3.9 Kollisionserkennung: Punkt innerhalb des Kreises?	161
3.10 Bubbles: Ausgabe Punktestand	165
3.11 Bubbles – Kollision mit dem Rand	166
3.12 Bubbles – Kollision der Blasen	166
3.13 Blase 2	169
3.14 Bubbles: Kollision anzeigen	171
3.15 Bubbles: Pausenbildschirm	172
3.16 Bubbles: Neustartbildschirm	174
3.17 Moonlander – Der Hintergrund	181
3.18 Gebirge (1)	182
3.19 Gebirge (2)	183
3.20 Gebirge (3)	184
3.21 Erde	187
3.22 mit Kontinenten	187
3.23 Sternenhimmel	189
3.24 Lander ohne Antrieb	191
3.25 Lander mit Antrieb	191
3.26 Moonlander – Der Lander	193
3.27 Moonlander – Jetzt mit Statusfenster	199
3.28 Moonlander – Beenden oder Neustart?	204
3.29 Moonlander – Autopilot	207

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. [71](#)

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, ... [53](#)

Dictionary Eine Datenstruktur, welche Werte unter einem einzigartigen Schlüssel ablegt. Andere Namen sind: Zuordnungstabelle, assoziatives Array, Hashtable. [69](#)

Doublebuffer Dies ist ein zweiter Speicherbereich, der genauso groß ist wie der Bildschirmspeicher. Wird jetzt etwas auf die Spielfläche gezeichnet, passiert dies zunächst auf diesem zweiten Speicher. Erst wenn alle Spielelemente ihr neues Aussehen gemalt haben, wird mit einem Schlag der alte Bildschirmspeicher mit dem zweiten ausgetauscht. Bei bestimmten Hardware- oder Grafikkonfigurationen kann es passieren, dass der Bildschirmspeicher neu gemalt wird, obwohl das Spiel noch nicht alle neuen Zustände abgebildet hat. Dadurch können hässliche Artefakte entstehen. Durch das Doublebuffering wird dieser Effekt vermieden. [6](#)

DTP-Punkt Maßeinheit für Schriftgrößen. [216](#)

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:` [156](#)

Event Ein Event dient in der Softwaretechnik zur Steuerung des Programmflusses. Das Programm wird nicht linear durchlaufen, sondern es werden spezielle Ereignisbehandlungsrouterien (engl. listener, observer, event handler) immer dann ausgeführt, wenn ein bestimmtes Ereignis auftritt. Ereignisorientierte Programmierung gehört zu den parallelen Programmiertechniken, hat also deren Vor- und Nachteile (Quelle: Wikipedia). [126](#)

fade Kommt vom englischen *to fade* für *verblassen*. In der Musik und bei Grafiken unterscheidet man einen *fadein* und einen *fadeout*. Bei einem *fadein* erscheint das Bild langsam bzw. wird die Lautstärke bei 0 beginnend auf die Ziellautstärke erhöht. Ein *fadeout* tut das Gegenteil. [104](#)

Flag Eine meist boolsche Variable, die eine Operation/Schleife ein- und ausschaltet. [7](#)

Fließkommazahl Bei einer Fließkommazahl werden Zahlen als Summen von Zweierpotenzen dargestellt, wobei der Exponent auch negativ sein kann. Beispiel: $6,75 = 2^2 + 2^1 - 1 + 2^{-2}$. Da der Speicherplatz beschränkt ist oder es für bestimmte Zahlen keine endliche Darstellung gibt, bricht die Summenbildung zu irgendeinem Zeitpunkt ab. Die dabei nicht mehr berücksichtigten Summanden führen zu den Rundungsfehlern. [37](#)

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. [60](#)

frames per second Maximale Anzahl der Bilder pro Sekunde. [7](#), [216](#)

Framework In der Informatik ist damit eine Arbeitsumgebung gemeint. Dies können einzelne Klassen, Funktionsbibliotheken oder ganze **Integrated Development Environment (IDE)** sein. [45](#)

Funktion Eine Funktion ist in der Programmierung ein Anweisungsblock mit einem Namen. Sie können Parametersätze haben und Ergebnisse zurückliefern. In der Regel gilt dabei das Prinzip, dass alle Werte innerhalb der Funktion lokal sind. [5](#)

Ganzzahl Bei einer Ganzkommazahl werden Zahlen als Summen von Zweierpotenzen dargestellt, wobei der Exponent Null oder positiv ist. Beispiel: $17 = 2^4 + 2^0$. Der Wertebereich ist durch den Speicherplatz, den man einer ganzen Zahl zur Verfügung stellt, definiert. Stehen n Bits zur Verfügung, können ohne Vorzeichen Zahlen im Bereich $[0, 2^n - 1]$ und mit Vorzeichen im Bereich $[2^{n-1}, 2^{n-1} - 1]$ dargestellt werden. [38](#)

Generative Pre-trained Transformer ChatGPT ist ein Prototyp eines Chatbots, also eines textbasierten Dialogsystems als Benutzerschnittstelle, der auf maschinellem Lernen beruht. Den Chatbot entwickelte das US-amerikanische Unternehmen OpenAI, das ihn im November 2022 veröffentlichte. (Quelle: Wikipedia). [216](#)

Grad (°) Maßeinheit für einen Winkel. Der Vollkreis hat dabei 360°. [101](#)

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 heruntergefahrene 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. [211](#), [216](#)

Joint Photographic Experts Group Verlustbehaftete komprimierte Bildinformationen. [210](#), [216](#)

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](#), [213](#)

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. [75](#)

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. [86](#), [216](#)

mp3 Abkürzung von *ISO MPEG Audio Layer 3*. Eine im wesentlichen vom deutschen Elektrotechnikingenieur und Mathematiker Karlheinz Brandenburg entwickeltes Kodierungs- und Kompressionsverfahren von Sound und Musik. [104](#)

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. [216](#)

ogg Kodierung von Sound-Dateien. Kommt vom englischen *to ogg*. Ziel war eine lizenfreie, einfache und effiziente Kodierung von Sound. [104](#)

Pixel Die kleinste bei gegebener Auflösung ansteuerbare Bildschirmfläche. [6](#), [216](#)

Polygon Ein geschlossener [Linienzug](#). Wird meist durch eine Folge von Punkten definiert, wobei der letzte Punkt mit dem ersten verbunden wird. [12](#), [212](#)

Portable Network Graphics Verlustfrei komprimierte Bildinformationen. [210](#), [216](#)

Pygame Pygame ist ein Verbund von Modulen, der die Entwicklung von Computerspielen in Python unterstützt. [4](#)

Pylance Pylance ist die Standard Python-Erweiterung von Visual Code zur Unterstützung der Python-Programmierung. Seine wentsentlichen Features sind die Typüberwachung und Autovervollständigung. [100](#)

Python Python ist eine höhere Interpretersprache mit prozeduralen und objektorientierten Paradigmen. Sie wurde 1991 von Guido van Rossum entwickelt und erfreut sich derzeit größter Beliebtheit. [4](#)

Radiant (rad) Maßeinheit für einen Winkel. Der Vollkreis hat dabei 2π rad. [101](#)

Red Green Blue Additive Farbkodierung. [216](#)

Rendern Das Erzeugen eines Bildes – meist in Bitmap-Format – aus einer Bildbeschreibungsangabe. [60](#)

Satz des Pythagoras In einem rechtwinkligen Dreieck ist die Summe der Kathetenquadrate gleich dem Hypotenusenquadrat: $c^2 = \sqrt{a^2 + b^2}$. Der Satz ist nach dem Mathematiker *Pythagoras von Samos* (um 570 v.Chr. bis um 510 v.Chr.) benannt. [161](#)

Semantik Bedeutung einer Angabe. Wird meist in Abgrenzung zu [Syntax](#) einer Angabe verwendet. [12](#), [214](#)

Signatur Die Signatur einer Funktion/Methode beschreibt die formalen Eigenschaften, die von einer Funktion/Methode von außen sichtbar ist. Dazu gehören die Sichtbarkeit, der Rückgabetyp, der Name und die Übergabeparameter. [51](#)

Simple Direct Media Layer Eine plattformunabhängige API für die Programmierung von Grafiken, Sounds und Eingabegräte. [216](#)

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. [49](#), [216](#)

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. [110](#)

Slicing Eine Technik, mit deren Hilfe man Teilmengen aus Strings oder Arrays bequem ausschneiden oder extrahieren kann. [71](#)

Solid-State-Drive Festspeicherplattentechnologie, welche nicht auf magnetische Prinzipien, sondern auf Halbleitertechnik basiert. [216](#)

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](#), [214](#)

Spritelib Meist eine Grafikdatei im Bitmap-Format, welches viele einzelne [Sprites](#) enthält. [67](#)

Stereofonie Verfahren um mit mehr als einer Schallquelle einen mehrdimensionalen Schalleindruck zu erzeugen. [216](#)

Syntax Form oder Grammatik einer Angabe. Wird meist in Abgrenzung zur [Semantik](#) einer Angabe verwendet. [214](#)

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*. [173](#)

Trade-off Jeder Vorteil wird durch einen Nachteil erkauft. Algorithmisch muss dann anhand der Datenlage abgewägt werden, ob in der Gesamtbetrachtung der Nutzen die Kosten überwiegt. Beispiel: Durch die Verwendung von Indizes werden Zugriffe auf Datenbankinhalte dramatisch beschleunigt (Nutzen). Um diese Beschleunigung zu erreichen, müssen Dateien angelegt werden, und das Anlegen, Ändern

und Löschen von Daten wird langsamer, da diese Dateien dann mit gepflegt werden müssen (Kosten). [120](#)

True Type Font Die Schriftinformation wird nicht im Bitmap-Format, sondern in einer Art Vektorgrafikformat abgespeichert. Dadurch lassen sich *beliebige* Schriftgrößen generieren. [216](#)

Unicode Ein Verfahren zur Kodierung von Zeichen und Symbolen. Gängige Umsetzungen sind UTF-8, UTF-16 und UTF-32. [71](#)

Universal Serial Bus Bitserielles Datenübertragungsprotokoll. [216](#)

Windows Bitmap Format Bildinformationen im Windows Bitmap-Format. [210](#), [216](#)

Akronyme

bmp Windows Bitmap Format. [210](#), *Glossar: Windows Bitmap Format*

ChatGPT Generative Pre-trained Transformer. [134](#), *Glossar: Generative Pre-trained Transformer*

fps frames per second. [7](#), *Glossar: frames per second*

IDE Integrated Development Environment. [211](#), *Glossar: Integrated Development Environment*

jpeg Joint Photographic Experts Group. [210](#), *Glossar: Joint Photographic Experts Group*

ms Millisekunden. [86](#), *Glossar: Millisekunden*

OO Objektorientiert. [60](#), *Glossar: Objektorientiert*

png Portable Network Graphics. [210](#), *Glossar: Portable Network Graphics*

pt DTP-Punkt. [60](#), *Glossar: DTP-Punkt*

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

RGB Red Green Blue. [7](#), [210](#), *Glossar: Red Green Blue*

SDL Simple Direct Media Layer. *Glossar: Simple Direct Media Layer*

SRP Single Responsibility Principle. [49](#), *Glossar: Single Responsibility Principle*

SSD Solid-State-Drive. [6](#), *Glossar: Solid-State-Drive*

Stereo Stereofonie. [107](#), *Glossar: Stereofonie*

ttf True Type Font. [64](#), *Glossar: True Type Font*

USB Universal Serial Bus. [6](#), *Glossar: Universal Serial Bus*

Index

- äquidistant, 23
- __main__, 50
- Alpha-Kanal, 11, 23, 148
- Animation, 89
- Autopilot, 203
- assoziatives Array, 210
- Ball, 139
- Bitmap, 21
 - ausgeben, 21
 - bewegen, 28
 - laden, 21
- Blasen erscheinen, 154
- Blasen zerplatzen, 162
- Blasenanzahl, 157
- Blasenwachstum, 158
- Bubbles, 151
- Dictionary, 53, 69
- Dirty Sprite, 114
- Doublebuffer, 6
- deltatime, 32, 35, 39, 83
- Erde, 186
- Ereignis, 54
- Event, 126
- Farbe
 - Alpha-Kanal, 11
 - Information, 11
 - Namen, 12
- Farbnamen, 19, 115, 124
- Flag, 6, 173
- Font, 60
 - farbe, 61
 - größe, 61
- Frame, 114
- Framerate, 122
- fade, 104
- float, 38
- GRAVITY, 15
- Gegenschub, 194
- Geschwindigkeit, 30
- Grafikprimitive, 11
 - Kreis, 11, 12
 - Linie, 11, 12
 - Polygon, 12
 - Punkt, 11, 13
 - Rechteck, 12
- Gravitation und Aufsetzen, 192
- Hashtable, 210
- Hauptprogrammschleife, 6
- Hintergrundmusik, 104
- int, 38
- Kanal, 108
- Kollision, 74
- Kollision anzeigen, 168
- Kollisionserkennung
 - Kreis, 74
 - Pixel, 75
 - Rechteck, 74
- Kreis, 11, 12
- Lander, 189
- Linie, 11, 12
- Maske, 75
- Maus, 97
- Mauscursor, 160
- Mausrad, 98

Mondoberfläche, 181
Moonlander, 178
main loop, 6
mp3, 104
Neustart, 173
ogg, 104
Pause, 171
Polygon, 12
Pong, 134
Punkt, 11, 13
Punkte, 141, 143, 144, 148
Punktestand, 163
Pythagoras, Satz von, 161
Rechteck, 12, 154
Rendern, 60
Richtung, 30
Richtungswechsel, 31
Schläger, 136
Schwerkraft, 15
Shift-Taste, 55
Sound, 146, 176
Soundausgaben, 103
Soundeffekte, 104
Spielende, 165
Spielende und Neustart, 198
Sprite, 45
 self.image, 45
 self.rect, 45
Standardfunktionalität, 134, 151, 178
Statusanzeige, 196
Sterne, 188
screen, 6
self.mask, 77
self.radius, 77, 154
self.rect, 77, 154
Tastatur, 53
 -konstanten, 55
 -schalter, 59
Timer, 87, 154, 168
Transparenz, 11, 23

Index für den Namensraum `pygame`

Color, [11](#), [19](#)
Event, [54](#)
 type, [127](#)
KEYDOWN, [54](#), [55](#)
KEYUP, [54](#), [55](#)
KEY, [55](#)
KMOD_LSHIFT, [55](#)
K_DOWN, [54](#)
K_ESCAPE, [54](#)
K_LEFT, [54](#)
K_RIGHT, [54](#)
K_SPACE, [55](#)
K_UP, [54](#)
MOUSEBUTTONDOWN, [98](#), [101](#), [162](#)
MOUSEBUTTONUP, [98](#), [101](#)
NUMEVENTS, [128](#), [133](#)
QUIT, [6](#), [9](#)
Rect, [164](#)
 collidepoint(), [99](#), [101](#)
 height, [154](#)
 left, [154](#)
 move(), [51](#)
 move_ip(), [51](#)
 top, [154](#)
 width, [154](#)
SYSTEM_CURSOR_CROSSHAIR, [161](#)
SYSTEM_CURSOR_HAND, [161](#)
Surface, [6](#), [135](#)
 blit(), [21](#), [27](#), [30](#)
 convert(), [22](#), [27](#)
 convert_alpha(), [23](#), [27](#)
 fill(), [7](#), [9](#)
 get_rect(), [29](#), [44](#), [77](#)
 set_at(), [13](#), [20](#)
 set_colorkey(), [23](#), [27](#), [77](#)
 subsurface(), [64](#), [66](#), [70](#), [73](#)
USEREVENT, [128](#), [133](#)
Window, [6](#), [9](#), [153](#)
 flip(), [6](#), [9](#)
 get_surface(), [6](#), [10](#)
 position, [6](#), [10](#)
 size, [16](#), [20](#)
 title, [6](#), [10](#)
clock
 tick(), [37](#), [39](#)
constants, [101](#)
display
 flip(), [119](#)
 update(), [119](#), [124](#)
draw
 circle(), [12](#), [19](#)
 line(), [12](#), [20](#)
 lines(), [12](#), [20](#)
 polygon(), [12](#), [20](#)
 rect(), [12](#), [20](#)
event
 get(), [133](#)
event
 Event(), [142](#)
 Event, [128](#), [133](#), [200](#)
 unicode, [71](#), [73](#)
 button, [98](#), [102](#)
 get(), [6](#), [9](#)
 key, [54](#)
 mod, [55](#)
 post(), [128](#), [133](#)
 pos, [98](#)
 type, [6](#), [9](#)
font
 Font, [60](#), [65](#)

```

        render(), 61, 66
        get_default_font(), 60, 66
        get_fonts(), 64, 66
        match_font(), 64, 66
    gfxdraw
        pixel(), 13, 20
    image, 27
        load(), 21, 27
    init(), 5, 9, 33, 104, 153
    key, 54
    locals
        WINDOWPOS_CENTERED, 133
    mask
        from_surface(), 77, 81
    math
        Vector2D, 53
        Vector2, 38, 44
        Vector3, 44
    mixer
        Channel, 108, 112
        play(), 109, 112
        set_volume(), 109, 113
        Sound, 104, 108, 113, 176
        get_volume(), 104, 113
        play(), 105, 108, 113, 176
        set_volume(), 110, 113
        find_channel(), 108, 113
        init(), 6, 103, 113
        music
            fadeout(), 105, 113
            get_volume(), 104, 113
            load(), 104, 113
            pause(), 106, 113
            play(), 104, 105, 113
            set_volume(), 104, 106, 110, 113
            unpause(), 106, 113
        mouse
            get_pos(), 13, 20, 99, 101, 161, 162
            get_pressed(), 13, 20
            set_visible(), 99, 101
    quit(), 7, 9
    rect
        FRect, 28, 38, 44
        bottomright, 28

```

```

        bottom, 28
        centerx, 28
        centery, 28
        center, 28
        height, 28
        left, 28
        move(), 44, 45
        move_ip(), 46
        right, 28
        size, 28
        topleft, 28
        top, 28
        width, 28
    Rect, 12, 20, 28, 44
        bottomright, 28
        bottom, 28
        centerx, 28
        centery, 28
        center, 28
        contains(), 166
        height, 28
        left, 28
        move(), 32, 44
        right, 28
        size, 28
        topleft, 28
        top, 28
        width, 28
    sprite
        DirtySprite, 117, 124
        dirty, 114, 117, 124
        GroupSingle, 49, 51, 135
        sprite, 49, 51
        Group, 48, 51, 118, 138, 164
        sprites(), 166
        LayeredDirty, 118, 124
        clear(), 119, 124
        draw(), 119, 124
        set_timing_threshold(), 120, 125
        set_timing_threshold(), 125
        Sprite, 45, 52
        kill(), 83, 88, 95, 162
        update(), 51
        collide_circle(), 80, 81, 156

```

collide_mask(), [80](#), [81](#)
collide_rect(), [47](#), [52](#), [80](#), [81](#), [143](#)
spritecollide(), [48](#), [52](#), [80](#), [81](#), [156](#)
DirtySprite, [151](#)
LayeredGroup, [151](#)
time
 Clock, [8](#), [9](#), [153](#)
 tick(), [8](#), [9](#), [83](#)
 tick_busy_loop(), [8](#), [9](#)
get_ticks(), [33](#), [44](#), [86](#), [88](#)
set_timer(), [132](#), [133](#)
wait(), [170](#)
transform
 rotate(), [101](#), [101](#)
 scale(), [22](#), [27](#), [158](#)