

# **Einführung in die 2D-Spieleprogrammierung mit Pygame**

Ralf Adams (TBS1, Bochum)

Version 0.3 vom 8. Februar 2022

# Inhaltsverzeichnis

<b>1 Ziele</b>	<b>3</b>
<b>2 Grundlagen</b>	<b>4</b>
2.1 Das erste Beispiel . . . . .	4
2.2 Grafikprimitive . . . . .	8
2.3 Bitmaps laden und ausgeben . . . . .	12
2.4 Bitmaps bewegen . . . . .	20
2.5 Sprite-Klasse . . . . .	24
2.6 Tastatur . . . . .	30
2.7 Textausgabe mit Fonts . . . . .	37
2.7.1 Default-Font . . . . .	37
2.7.2 Fontliste . . . . .	40
2.8 Textausgabe mit Bitmaps . . . . .	43
2.9 Kollisionserkennung . . . . .	50
2.10 Zeitsteuerung . . . . .	58
2.11 Animation . . . . .	65
2.11.1 Die laufende Katze . . . . .	65
2.11.2 Der explodierende Felsen . . . . .	70
2.12 Maus . . . . .	72
<b>3 Beispielprojekte</b>	<b>78</b>
3.1 Bubbles . . . . .	78
3.1.1 Requirement 1: Standards . . . . .	78
3.1.2 Requirement 2: Blasen erscheinen . . . . .	81
3.1.3 Requirement 3: Blasenzahl . . . . .	84
3.1.4 Requirement 4: Blasenwachstum . . . . .	85
3.1.5 Requirement 5: Mauscursor . . . . .	87
3.1.6 Requirement 6: Blasen zerplatzen . . . . .	88
3.1.7 Requirement 7: Punkttestand . . . . .	89
3.1.8 Requirement 8: Spielende . . . . .	91
3.1.9 Requirement 9: Zeitanpassungen . . . . .	93
3.1.10 Requirement 10: Kollision anzeigen . . . . .	94
3.1.11 Requirement 11: Pause . . . . .	97
3.1.12 Requirement 12: Neustart . . . . .	100
3.1.13 Requirement 13: Sound . . . . .	103

# 1 Ziele

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

Im ersten Teil werden die wichtigsten Konzepte anhand einfacher Beispiele eingeführt. Im zweiten Teil wird ein Spielprojekt vollständig durchprogrammiert und damit der Einsatz der Techniken verdeutlicht.

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

Für eine Rückmeldung bei groben Patzern wäre ich sehr dankbar: [adams@tbs1.de](mailto:adams@tbs1.de).

# 2 Grundlagen

## 2.1 Das erste Beispiel

Quelltext 2.1: Mein erstes *Spiel*, Version 1.0

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

Um Pygame verwenden zu können, muss das Modul `pygame` importiert werden (Zeile 1). Danach stehen uns **Konstanten**, **Funktionen** und **Klassen** des **Namensraums** zur Verfügung.

In Zeile 5 wird die **Umgebungsvariable** gesetzt, die erstmal nichts mit Pygame zu tun hat. Vielmehr wird hier die Umgebungsvariable `SDL_VIDEO_WINDOW_POS` des Betriebssystems gesetzt. Diese steuert die linke obere Startposition meines Fensters bezogen auf den ganzen Bildschirm.

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 6 wird der ganze Pygame-Motor mit `init()` angeworfen. Man könnte auch nur die Komponenten starten, die gerade gebraucht werden wie beispielsweise die Soundunterstützung mit `pygame.mixer.init()`.

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

`SDL_VIDEO_WINDOW_POS`

`init()`

`set_mode()`

kommt in Zeile 9 einen(!) Übergabeparameter – nämlich die Breite und die Höhe des Fensters als ein 2-Tupel. Unser Fenster ist also 600 *px* breit und 400 *px* (siehe [Pixel \(px\)](#)) hoch. Als Rückgabe bekomme ich ein `pygame.Surface`-Objekt, was ungefähr sowas wie ein [Bitmap](#) ist. Dem Fenster kann ich dann noch mit `pygame.display.set_caption()` eine Titelüberschrift verpassen (siehe Zeile 7).

[set\\_caption\(\)](#)

Das Spiel selbst – so wie auch alle zukünftigen Spiele – laufen innerhalb einer [Hauptprogrammschleife](#). Hier startet die Schleife in Zeile 11 und endet in Zeile 18. Innerhalb dieser Schleife werden zukünftig immer drei Dinge passieren:

1. Ereignisse auslesen und verarbeiten: Wie in Zeile 12f. werden Maus-, Tastatur- oder Konsolenereignisse festgestellt und an die Spielemente weitergegeben. In unserem Fall wird lediglich das Anklicken des X im Fenster oben rechts registriert.
2. Zustand der Spielemente aktualisieren: Basierend auf den oben festgestellten Ereignissen und den Zuständen der Spielemente, werden die neuen Zustände ermittelt (Spieler bewegt sich, Geschoß prallt auf, Punkte erhöhen sich etc.). In unserem Fall wird nur das Flag `running` der Hauptprogrammschleife auf `False` gesetzt.
3. Bitmaps der Spielemente malen: Die Spielemente haben eine neue Position oder ein neues Aussehen und müssen deshalb neu gemalt werden. In diesem Minimalbeispiel wird lediglich Zeile 15 der Hintergrund der Spielfläche eingefärbt und anschließend in Zeile 16 der [Doublebuffer](#) mit `pygame.display.flip()` ausgetauscht.

[Doublebuffer](#)  
[flip\(\)](#)

Pygame schleust durch den Aufruf von `pygame.init()` einen Horchposten in das Betriebssystem. Und zwar horcht Pygame die [Message Queue](#) ab. Dort werden vom Betriebssystem alle Meldungen eingesammelt, die durch Ereignisse ausgelöst werden. Dies können [USB](#)-Anschlussmeldungen, [SSD](#)-Fehlermeldungen, Mausaktionen, Programmstarts bzw. -abstürze usw. sein. Pygame fischt nun aus der Message-Queue mit Hilfe von `pygame.event.get()` alle Events, die das Spiel betreffen könnten heraus. Mit Hilfe einer `for`-Schleife iteriere ich nun die Ereignisse und picke die für mich interessanten heraus.

[event.get\(\)](#)

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

[event.type](#)  
[pyga-me.QUIT](#)

Falls ich dieses Signal nicht empfange, läuft die Hauptprogrammschleife fröhlich weiter und füllt in Zeile 15 die gesamte Spielfläche mit `screen.fill()` mit einer Farbe – hier grün – ein. Bitte beachten Sie, dass ähnlich wie in Zeile 9 die Funktion einen Übergabeparameter – nämlich ein 3-Tupel – erwartet. Dieses 3-Tupel kodiert die Farbe durch [RGB](#)-Angaben zwischen 0 und 255.

[RGB](#)

Verbleibt noch Zeile 16: Dort wird die Funktion `pygame.quit()` aufgerufen. Diese Funktion ist quasi das Gegenteil von `pygame.init()` in Zeile 6. Alle reservierten Ressourcen

[quit\(\)](#)

werden wieder freigegeben und die Pygame-Horchposten werden wieder aus dem System entfernt. Rufen Sie diese Funktion unbedingt immer am Ende Ihrer Anwendung auf; beenden Sie nicht einfach das Spiel. Der Unterschied entspricht dem einfachen Herauslaufen aus der Wohnung und dem ordnungsgemäßen Lichtausmachen und Türabschließen beim Verlassen der Wohnung.

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

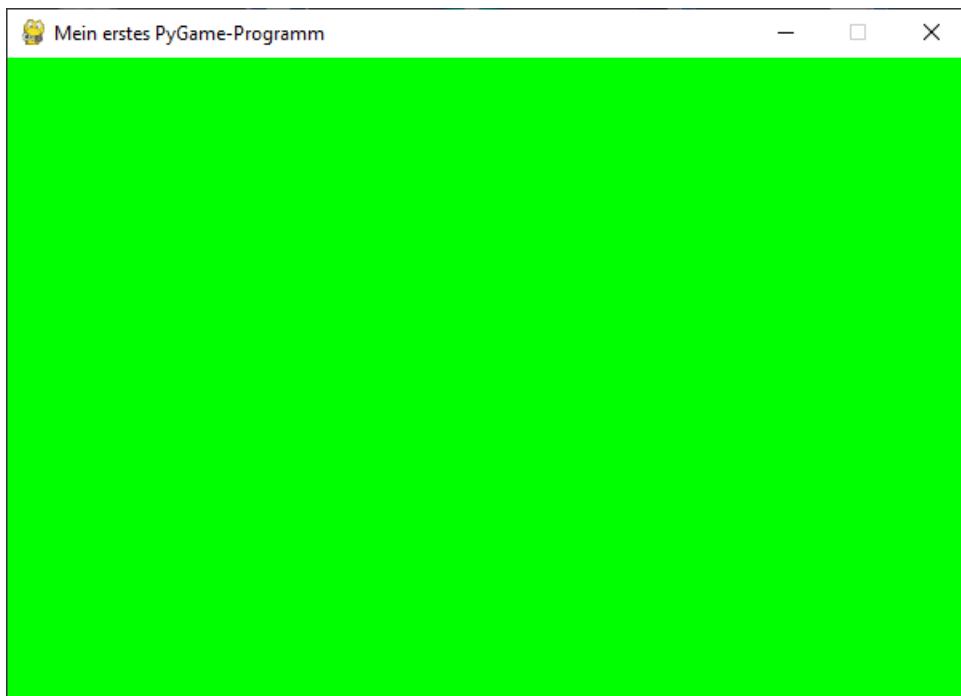


Abbildung 2.1: Eine einfache grüne Spielfläche

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

---

In Zeile 10 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 14. 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.

Clock

tick()

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.

tick\_busy\_loop()



Abbildung 2.3: Ressourcenverbrauch mit Taktung

## Was war neu?

- `import pygame:`  
<https://www.pygame.org/docs/tut/ImportInit.html>

- `os.environ['SDL_VIDEO_WINDOW_POS']:`  
<https://docs.python.org/3/library/os.html#os.environ>
- `pygame.init():`  
<https://www.pygame.org/docs/ref pygame.html#pygame.init>
- `pygame.quit():`  
<https://www.pygame.org/docs/ref pygame.html#pygame.quit>
- `pygame.display.set_mode():`  
[https://www.pygame.org/docs/ref display.html#pygame.display.set\\_mode](https://www.pygame.org/docs/ref display.html#pygame.display.set_mode)
- `pygame.display.set_caption():`  
[https://www.pygame.org/docs/ref display.html#pygame.display.set\\_caption](https://www.pygame.org/docs/ref display.html#pygame.display.set_caption)
- `pygame.display.flip():`  
<https://www.pygame.org/docs/ref display.html#pygame.display.flip>
- `pygame.time.Clock:`  
<https://www.pygame.org/docs/ref time.html#pygame.time.Clock>
- `pygame.time.Clock.tick():`  
<https://www.pygame.org/docs/ref time.html#pygame.time.Clock.tick>
- `pygame.time.Clock.tick_busy_loop():`  
[https://www.pygame.org/docs/ref time.html#pygame.time.Clock.tick\\_busy\\_loop](https://www.pygame.org/docs/ref time.html#pygame.time.Clock.tick_busy_loop)
- `pygame.event.get():`  
<https://www.pygame.org/docs/ref event.html#pygame.event.get>
- `pygame.event.type:`  
<https://www.pygame.org/docs/ref event.html#pygame.event.EventType.type>
- `pygame.QUIT:`  
<https://www.pygame.org/docs/ref event.html#pygame.event.EventType.type>
- `pygame.Surface.fill():`  
<https://www.pygame.org/docs/ref surface.html#pygame.Surface.fill>

## 2.2 Grafikprimitive

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

---

Quelltext 2.3: Mein zweites *Spiel*, Version 1.0

---

```
1 import pygame
2 import pygame.gfxdraw
3 import os
4
# Muss sein!
```

```
5  if __name__ == '__main__':
6      os.environ['SDL_VIDEO_WINDOW_POS'] = "10, 50"
7      pygame.init()
8      pygame.display.set_caption('Mein zweites PyGame-Programm')
9      screen = pygame.display.set_mode((530, 530))
10     clock = pygame.time.Clock()
11
12    grey = pygame.Color(200,200,200)                      # Ein paar Farben
13    red =  pygame.Color(255,0,0)
14    green = pygame.Color(0,255,0)
15    blue =  pygame.Color(0,0,255)
16
17    myrectangle1 = pygame.Rect(10, 10, 20, 30)           # Ein Rechteck-Objekt
18    myrectangle2 = pygame.Rect(60, 10, 20, 30)
19    points1 = ((120, 10), (160, 10), (140, 90))        # Punktliste
20    points2 = ((180, 10), (220, 10), (200, 90))
21
22
23    running = True
24    while running:
25        clock.tick(60)
26        for event in pygame.event.get():
27            if event.type == pygame.QUIT:
28                running = False
29        screen.fill(grey)
30        pygame.draw.rect(screen, red, myrectangle1)          # Gefülltes Rechteck
31        pygame.draw.rect(screen, red, myrectangle2, 3, 10)    # Anderes Rechteck
32        pygame.draw.polygon(screen, green, points1)         # Gefülltes Polygon
33        pygame.draw.polygon(screen, green, points2, 2)       # Nicht gefülltes Polygon
34        pygame.draw.line(screen, red, (5, 230), (240, 230), 3) # Linie
35        pygame.draw.circle(screen, blue, (40, 150), 30)       # Gefüllter Kreis
36        pygame.draw.circle(screen, blue, (110, 150), 30, 2)   # Nicht gefüllter Kreis
37        pygame.draw.circle(screen, blue, (180, 150), 30, 5, True) # Kreisbogenschnitt
38        for i in range(255):
39            for j in range(255):
40                screen.set_at((265+i, 10+j), (255, i, j))      # Punkte Variante
41                1
42                screen.fill((i, j, 255), ((10+i, 265+j), (1, 1)))    # Variante 2
43                pygame.gfxdraw.pixel(screen, 265+i, 265+j, (i, 255, j)) # Variante 3
44
45        pygame.display.flip()
46
47    pygame.quit()
```

---

Der Grundaufbau ist der gleiche wie in Quelltext 2.2 auf Seite 7. Die Unterschiede beginnen in Zeile 12. Die Klasse `pygame.Color` kann Farbinformationen in verschiedenen Formaten inklusive eines [Alpha-Kanals](#) (Transparenz) kodieren; mehr dazu später. Ich verwende hier eine RGB-Kodierung mit Farbkanalwerten zwischen 0 und 255.

Color

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 17 wird also ein Rechteck an der Position (10, 10) mit der Breite von 20 px und einer Höhe von 30 px definiert.

Rect

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

In Zeile 30 zeichnet `pygame.draw.rect()` ein gefülltes Rechteck. Die [Semantik](#) der Pa-

rect()

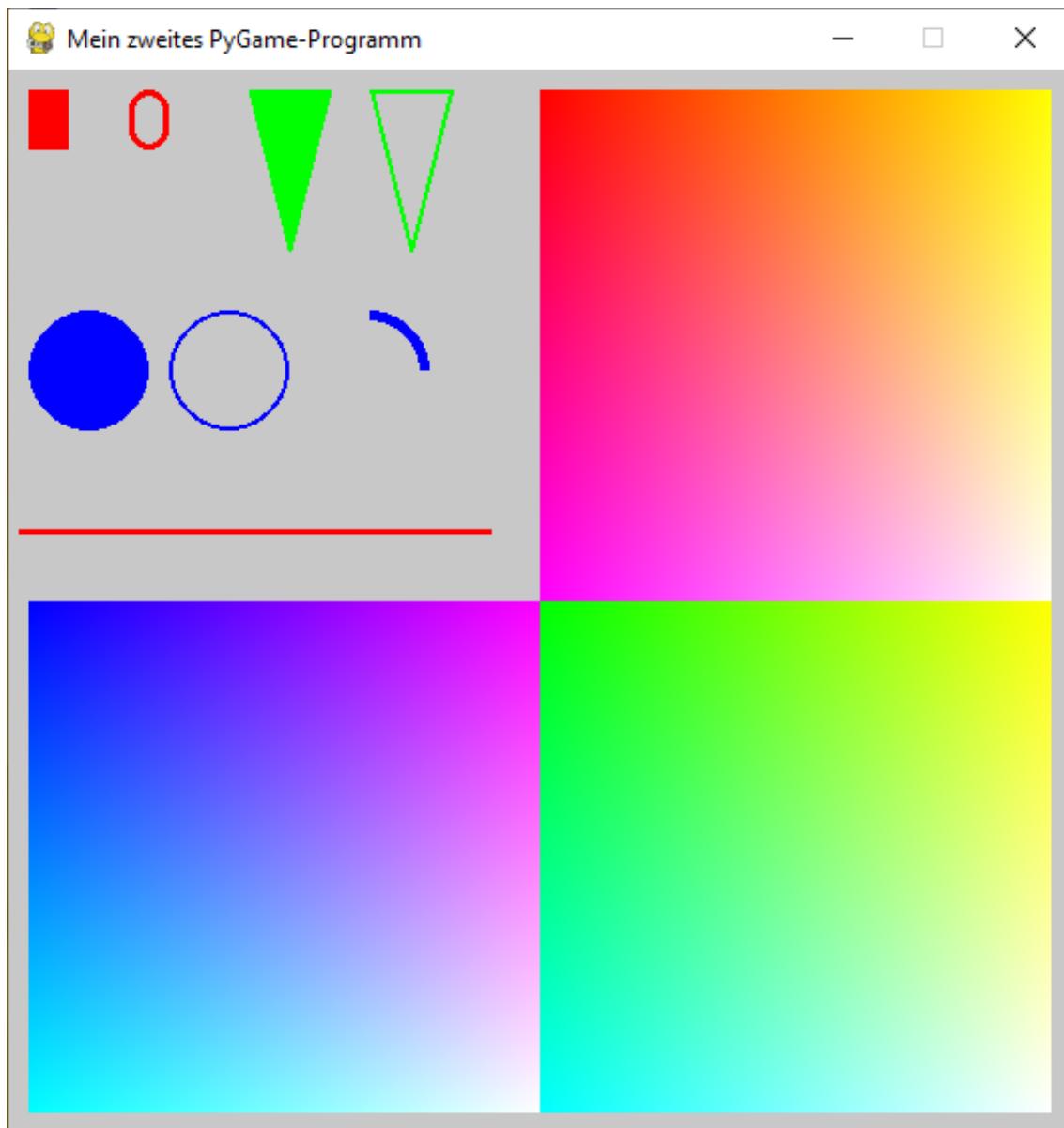


Abbildung 2.4: Einige Grafikprimitive

rameter sollte selbsterklärend sein. Anders der Aufruf von Zeile 31. 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 Lienenzug, der in Pygame durch seine Punkte (Ecken) definiert wird. Ähnlich wie bei den Rechtecken, gibt es gefüllte (Zeile 32) und un gefüllte (Zeile 33) 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.

`polygon()`

Für einzelne Linien gibt es `pygame.draw.line()` bzw. für einen – hier ohne Beispiel – [Linienzug](#) `pygame.draw.lines()`. Ein Beispiel finden Sie in Zeile 34.

`line()`

`lines()`

Ein Kreis wird durch zwei Angaben definiert: Mittelpunkt und Radius. In Zeile 35 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 36). Interessant ist der Kreisbogenausschnitt in Zeile 37. Hier wird über boolsche Variablen gesteuert, welcher Abschnitt des Kreisbogens gezeichnet wird (Näheres in der Pygame-Referenz).

`circle()`

Zum Schluss noch einen klein 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 40 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 41). Ein Möglichkeit einen Pixel über eine Grafikbibliothek zu setzen, ist die experimentelle `gfdxdraw`-Umgebung. In Zeile 42 wird mit `pygame.gfdxdraw.pixel()` ein einzelnes Pixel gesetzt. Die `gfdxdraw`-Umgebung wird nicht automatisch durch `import pygame` importiert (siehe Zeile 2).

`set_at()`

`pixel()`

## Was war neu?

- `import pygame.gfdxdraw:`  
<https://www.pygame.org/docs/ref/gfdxdraw.html>
- `import pygame.gfdxdraw.pixel():`  
<https://www.pygame.org/docs/ref/gfdxdraw.html#pygame.gfdxdraw.pixel>
- `pygame.Color:`  
<https://www.pygame.org/docs/ref/color.html>
- `pygame.Rect:`  
<https://www.pygame.org/docs/ref/rect.html>

- `pygame.draw.rect():`  
<https://www.pygame.org/docs/ref/draw.html#pygame.draw.rect>
- `pygame.draw.polygon():`  
<https://www.pygame.org/docs/ref/draw.html#pygame.draw.polygon>
- `pygame.draw.line():`  
<https://www.pygame.org/docs/ref/draw.html#pygame.draw.line>
- `pygame.draw.lines():`  
<https://www.pygame.org/docs/ref/draw.html#pygame.draw.lines>
- `pygame.draw.circle():`  
<https://www.pygame.org/docs/ref/draw.html#pygame.draw.circle>
- `pygame.Surface.set_at():`  
[https://www.pygame.org/docs/ref/surface.html#pygame.Surface.set\\_at](https://www.pygame.org/docs/ref/surface.html#pygame.Surface.set_at)

## 2.3 Bitmaps laden und ausgeben

Quelltext 2.4: Bitmaps laden und ausgeben, Version 1.0

---

```
1 import pygame
2 import os
3
4 class Settings:
5     window_width = 600
6     window_height = 400
7     fps = 60
8
9
10 if __name__ == '__main__':
11     os.environ['SDL_VIDEO_WINDOW_POS'] = "10, 50"
12     pygame.init()
13
14     screen = pygame.display.set_mode((Settings.window_width, Settings.window_height))
15     pygame.display.set_caption("Bitmaps laden und ausgeben")
16     clock = pygame.time.Clock()
17
18     defender_image = pygame.image.load("images/defender01.png")           #Bitmap laden
19     enemy_image = pygame.image.load("images/alienbig0101.png")
20
21     running = True
22     while running:
23         clock.tick(Settings.fps)
24         for event in pygame.event.get():
25             if event.type == pygame.QUIT:
26                 running = False
27
28         screen.fill((255, 255, 255))
29         screen.blit(enemy_image, (10, 10))                                     #Bitmap ausgeben
30         screen.blit(defender_image, (10, 80))
31         pygame.display.flip()
32
33     pygame.quit()
```

---

In Quelltext 2.4 werden zwei Bitmaps – hier zwei png-Dateien – geladen und auf den Bildschirm ausgegeben.

Das Laden erfolgt über die Funktion `pygame.image.load()`. In Zeile 18f. 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 2.5 bewundern.

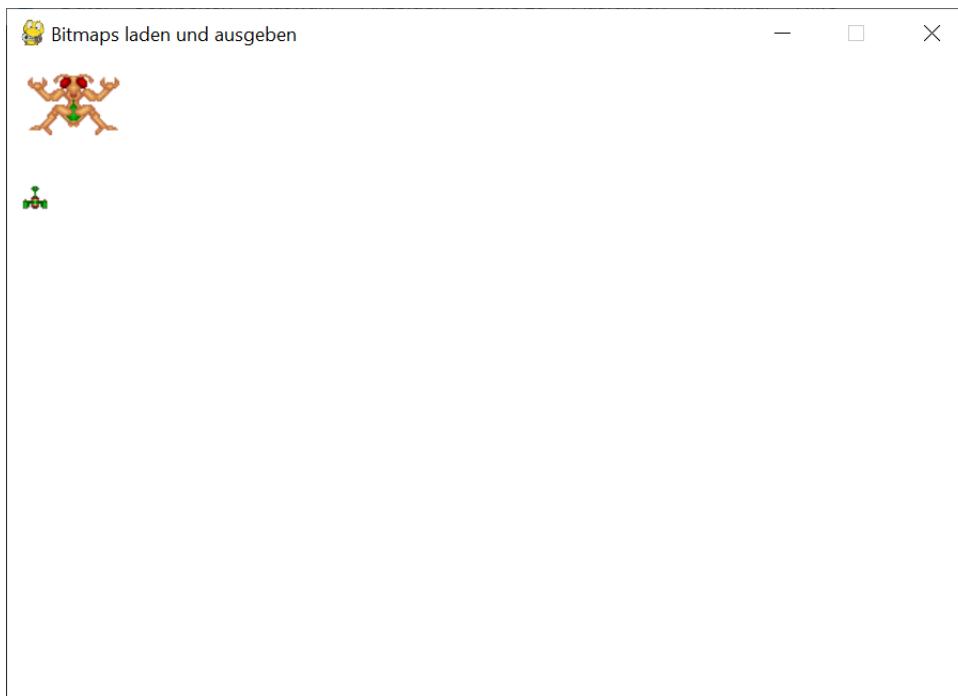


Abbildung 2.5: Bitmaps laden und ausgeben, Version 1.0

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 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 2.5: Bitmaps laden und ausgeben, Version 1.1

---

```
19     defender_image = pygame.image.load("images/defender01.png").convert()      #Bitmap  
20     konvertieren  
20     defender_image = pygame.transform.scale(defender_image, (30,30))          #Bitmap  
skalieren
```

```
21
22     enemy_image = pygame.image.load("images/alienbig0101.png").convert()
23     enemy_image = pygame.transform.scale(enemy_image, (50,45))
```

Die Funktion `pygame.Surface.load()` lieferte mir ja 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 angegebenen (*width, height*) in der Maßeinheit Pixel skaliert. Das Ergebnis an Abbildung 2.6 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()` 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*.

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

- `pygame.Surface.convert_alpha()`: Ganz einfach formuliert wird 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 *halbdurchscheinend* zu lassen. Zum anderen werden Teile des 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.



Abb. 2.6: Größen OK

`convert()`

`scale()`

`Alpha-Kanal`

`convert_alpha()`

`set_colorkey()`

### Quelltext 2.6: Bitmaps laden und ausgeben, Version 1.2

```
19     defender_image = pygame.image.load("images/defender01.png").convert_alpha()      #Bitmap
20     konvertieren
21
22     defender_image = pygame.transform.scale(defender_image, (30,30))
23
24     enemy_image = pygame.image.load("images/alienbig0101.png").convert()           #Unsichtbare
25     enemy_image.set_colorkey((0, 0, 0))                                         Farbe setzen
26
27     enemy_image = pygame.transform.scale(enemy_image, (50,45))
```

In Quelltext 2.6 habe ich beide Varianten mal ausprobiert und in Abbildung 2.7 auf der nächsten Seite können Sie das Ergebnis sehen. Nun sind beide Bitmaps ohne schwarze Umrandung sichtbar, der weiße Hintergrund scheint wieder durch.

Was mir nun 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.

 Bitmaps laden und ausgeben



äquidistant



Abb. 2.7: Transparenz OK

Quelltext 2.7: Bitmap: Positionen, Version 1.4

---

```

1 import pygame
2 import os
3
4
5 class Settings:
6     window_width = 600
7     window_height = 400
8     fps = 60
9     aliens_nof = 7
10
11
12 if __name__ == '__main__':
13     os.environ['SDL_VIDEO_WINDOW_POS'] = "10, 50"
14     pygame.init()
15
16     screen = pygame.display.set_mode((Settings.window_width, Settings.window_height))
17     pygame.display.set_caption("Bitmaps laden und ausgeben")
18     clock = pygame.time.Clock()
19
20     defender_image = pygame.image.load("images/defender01.png").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
34
35     running = True
36     while running:
37         clock.tick(Settings.fps)
38         for event in pygame.event.get():
39             if event.type == pygame.QUIT:
40                 running = False
41
42         screen.fill((255, 255, 255))                                     # Abstand von oben
43         alien_top = 10                                              # Berechnung/Ausgabe
44         for i in range(Settings.aliens_nof):
45             alien_left = space_between_aliens + i * (space_between_aliens + 50)
46             alien_pos = (alien_left, alien_top)
47             screen.blit(alien_image, alien_pos)
48         screen.blit(defender_image, defender_pos)                     # Benutze Position
49         pygame.display.flip()

```

```
50  
51     pygame.quit()
```

---

In Quelltext 2.7 auf der vorherigen Seite sind die obigen Anforderungen umgesetzt worden. Schauen wir uns die einzelnen Aspekte genauer an.

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

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

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

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

---

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

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

---

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

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

---

würden wir die horizontale Mitte des Bildschirmes ausrechnen. Diesen Wert können wir aber nicht einsetzen, da dann der linke Rand des Verteidigers in der horizontalen Mitte stehen würde – also zu weit rechts (siehe Abbildung 2.8).

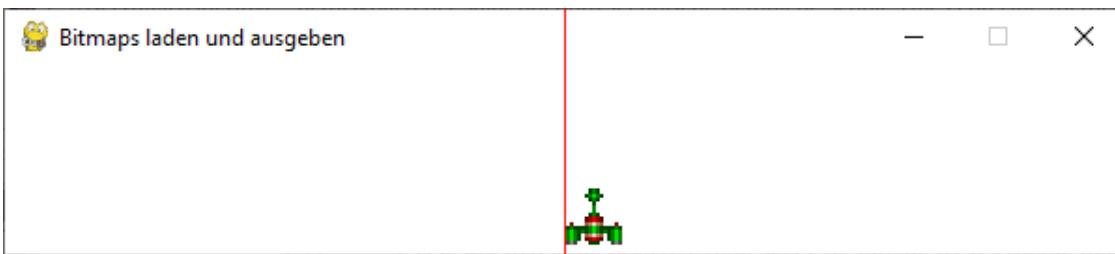


Abbildung 2.8: Bitmaps positionieren (Verteidiger)

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

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

---

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

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

---

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

```
44     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  $50\text{ px}$  beträgt. In eine for-Schleife gegossen, sieht das so aus:

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

---

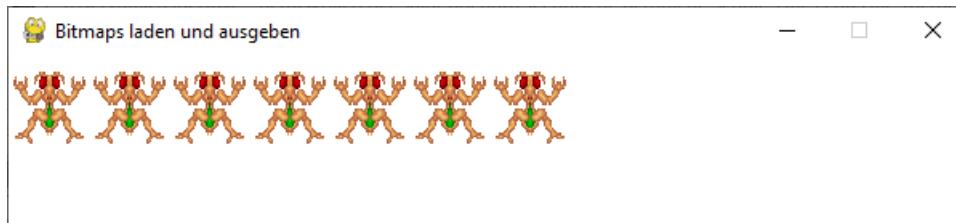


Abbildung 2.9: Bitmaps positionieren (Angreifer, Version 1)

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:

```
31     space_nof = 2
```

---

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

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

---

also:

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

---

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

---

```
29     space_for_aliens = Settings.aliens_nof * 50
```

---

und diesen von der Bildschirmbreite abziehe.

---

```
30     space_available = Settings.window_width - space_for_aliens
```

---

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

---

```
32     space_between_aliens = space_available // space_nof
```

---

Jetzt müssen wir nur noch die Berechnung von `alien_left` anpassen. Erstmal verschieben wir den Start um einen solchen Freiraum (siehe Abbildung 2.10):

---

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

---

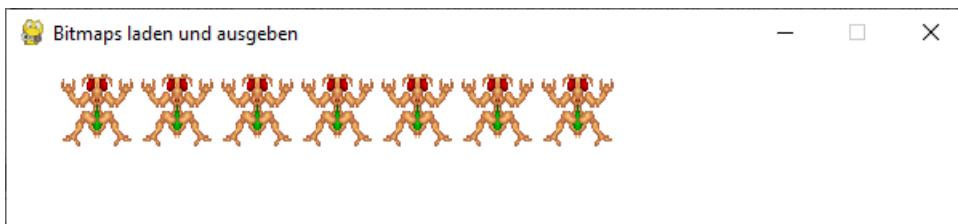


Abbildung 2.10: Bitmaps positionieren (Angreifer, Version 2)

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:

---

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

---

Und schon passt alles (siehe Abbildung 2.11 auf der nächsten Seite).

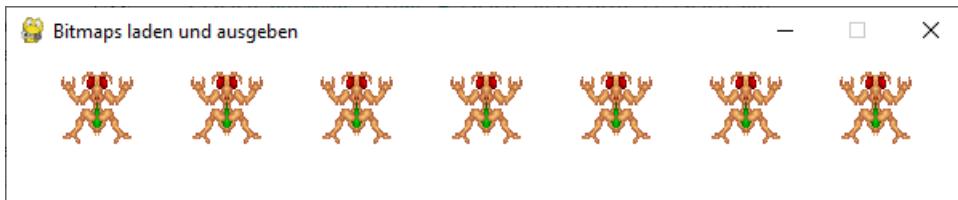


Abbildung 2.11: Bitmaps positionieren (Angreifer, Version 3)

### Was war neu?

Zum Abschluss möchte ich eine kurze Zusammenfassung darüber geben, was wir hier an Informationen erworben haben:

- Die Positionsangaben werden bei der Ausgabe auf dem Bildschirm benötigt. Wir werden später sehen, dass wir die Positionsangaben auch noch für andere Fragestellungen brauchen, wie beispielsweise die [Kollisionserkennung](#).
- Die Positionsangabe bezieht sich immer auf die linke, obere Ecke des Bitmaps.
- Das Koordinatensystem hat seinen 0-Punkt linksoben und nicht linksunten.
- Wir müssen häufig elementare Geometrieberechnungen durchführen und am besten macht man diese Schritt für Schritt.
- Für solche Geometrieberechnungen werden folgende Informationen gebraucht: die Position des Bitmap, seine Breite und Höhe. Breite und Höhe haben wir hier noch als Konstanten verarbeitet, dass ist nicht zukunftsweisend.

Und hier die neuen Klassen bzw. Funktionen:

- `pygame.image` :  
<https://www.pygame.org/docs/ref/image.html>
- `pygame.image.load()` :  
<https://www.pygame.org/docs/ref/image.html#pygame.image.load>
- `pygame.Surface.blit()`:  
<https://www.pygame.org/docs/ref/surface.html#pygame.Surface.blit>
- `pygame.Surface.convert()`:  
<https://www.pygame.org/docs/ref/surface.html#pygame.Surface.convert>
- `pygame.Surface.convert_alpha()`:  
[https://www.pygame.org/docs/ref/surface.html#pygame.Surface.convert\\_alpha](https://www.pygame.org/docs/ref/surface.html#pygame.Surface.convert_alpha)
- `pygame.Surface.set_colorkey()`:  
[https://www.pygame.org/docs/ref/surface.html#pygame.Surface.set\\_colorkey](https://www.pygame.org/docs/ref/surface.html#pygame.Surface.set_colorkey)
- `pygame.transform.scale()`:  
<https://www.pygame.org/docs/ref/transform.html#pygame.transform.scale>

## 2.4 Bitmaps bewegen

In der Zusammenfassung des vorherigen Kapitels haben wir für die Darstellung von Bitmaps notiert, dass wir die linke, obere Ecke als Positionsangabe und die Höhe und Breite beispielsweise für Abstandsberechnungen brauchen. Diese Angaben lassen sich gut einem Rechteck kodieren. Pygame stellt dazu die Klasse `pygame.Rect` zur Verfügung.

Rect

In Abbildung 2.12 finden Sie die meiner Ansicht nach wichtigsten Attribute der Klasse.

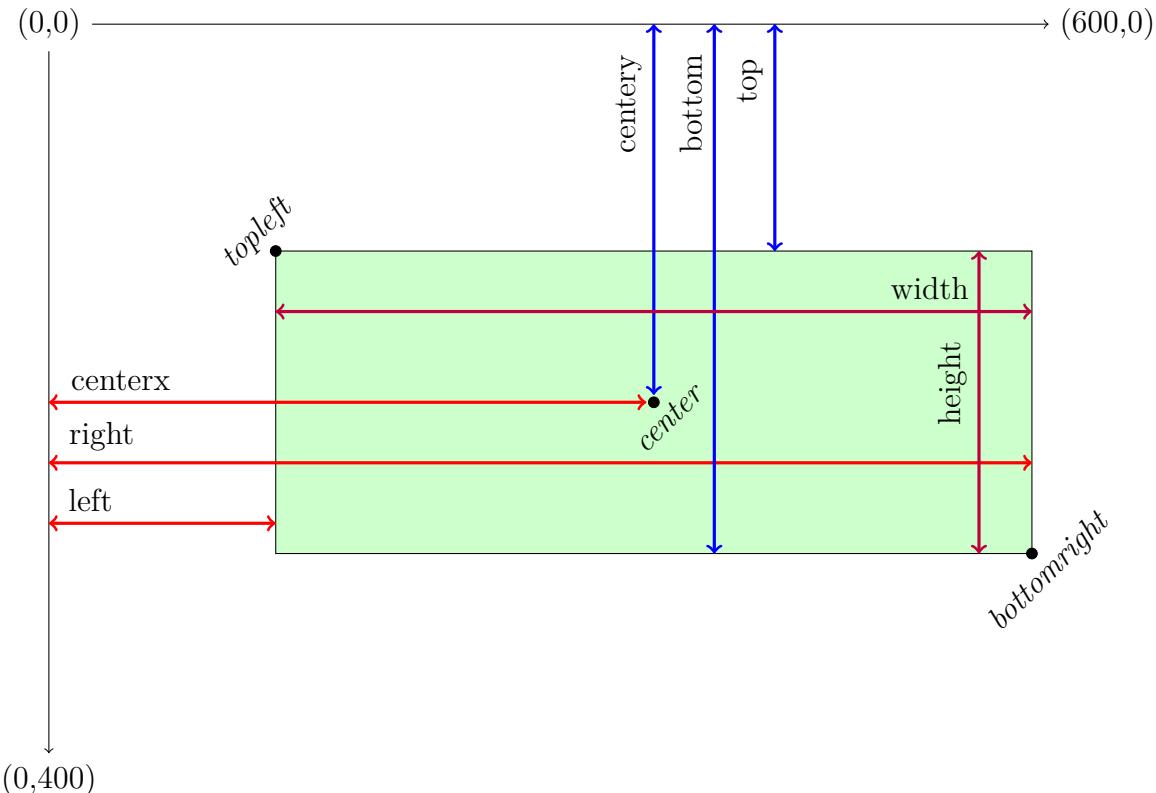


Abbildung 2.12: Elemente eines `Rect`-Objekts

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 2.8 wird die `Rect`-Klasse schon verwendet.

### Quelltext 2.8: Bitmaps bewegen, Version 1.0

---

```
1 import pygame
2 import os
3
4
5 class Settings:
6     window = {'width':600, 'height':100}
7     fps = 60
8     @staticmethod
9     def window_dim():
10         return (Settings.window['width'], Settings.window['height'])
11
12
13 if __name__ == '__main__':
14     os.environ['SDL_VIDEO_WINDOW_POS'] = "10, 50"
15     pygame.init()
16
17     screen = pygame.display.set_mode(Settings.window_dim())
18     pygame.display.set_caption("Bewegung")
19     clock = pygame.time.Clock()
20
21     defender_image = pygame.image.load("images/defender01.png").convert_alpha()
22     defender_image = pygame.transform.scale(defender_image, (30, 30))
23     defender_rect = defender_image.get_rect()                      # Rect-Objekt
24     defender_rect.centerx = Settings.window['width'] // 2      # Nicht nur left
25     defender_rect.bottom = Settings.window['height'] - 5        # Nicht nur top
26
27     running = True
28     while running:
29         clock.tick(Settings.fps)
30         # Events
31         for event in pygame.event.get():
32             if event.type == pygame.QUIT:
33                 running = False
34
35         # Update
36
37         # Draw
38         screen.fill((255, 255, 255))
39         screen.blit(defender_image, defender_rect)                  # blit kann auch rect
40         pygame.display.flip()
41
42     pygame.quit()
```

---

Für `Surface`-Objekte können wir sehr bequem mit `pygame.Surface.get_rect()` das `Rect`-Objekt erstellen lassen (Zeile 23). Die Positionierung kann nun leichter über die Attribute erfolgen. Das Zentrum muss beispielsweise nicht mehr in die Berechnung einfließen, ich kann vielmehr das horizontale Zentrum direkt als halbe Fensterbreite festlegen (Zeile 24). 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 25). Und als Sahnehäubchen kann das `Rect`-Objekt auch noch als Parameter der `blit()`-Funktion übergeben werden (Zeile 39).

`get_rect()`

`blit()`

Das Ergebnis ist unspektakulär (siehe Abbildung 2.13 auf der nächsten Seite) und hat noch nichts mit Bewegung zu tun.



Abbildung 2.13: Bitmaps bewegen, Version 1.0

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`.

---

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

---

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

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

Richtung

Geschwindigkeit

Quelltext 2.9: Bitmaps bewegen, Version 1.2

---

```
26  defender_speed = 2                                # Geschwindigkeit
27  defender_direction_h = 1                          # Richtung
28
29  running = True
30  while running:
31      clock.tick(Settings.fps)
32      # Events
33      for event in pygame.event.get():
34          if event.type == pygame.QUIT:
35              running = False
36
37      # Update
38      defender_rect.left += defender_direction_h * defender_speed # Flexibler
```

---

Diese beiden Informationen werden nun in Quelltext 2.9 dazu genutzt, die Bewegung erheblich flexibler zu gestalten. In Zeile 26 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 27 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 38 zur Berechnung der neuen horizontalen Position genutzt werden.

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

### Quelltext 2.10: Bitmaps bewegen, Version 1.3

---

```
37      # Update
38      defender_rect.left += defender_direction_h * defender_speed
39      if defender_rect.right >= Settings.window['width']: # Rechter Rand erreicht
40          defender_direction_h *= -1                         # Richtungswechsel
41      elif defender_rect.left <= 0:                          # Linker Rand erreicht
42          defender_direction_h *= -1
```

---

Ich hoffe, dass Sie die Idee hinter dem Code erkennen. Nach Berechnung der neuen horizontalen Position, wird in Zeile 39 überprüft, ob der neue rechte Rand die Bildschirmbreite 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 38 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.move()`.

move()

### Quelltext 2.11: Bitmaps bewegen, Version 1.4

---

```
38      newpos = defender_rect.move(defender_direction_h * defender_speed, 0) #
            Testposition
39      if newpos.right >= Settings.window['width']:
40          defender_direction_h *= -1
41      elif newpos.left <= 0:
42          defender_direction_h *= -1
43      else:
44          defender_rect = newpos                      # Übernehme neue Position
```

---

Die neue Funktion taucht in Zeile 38 zum ersten Mal auf. Sie hat zwei Parameter. Mit dem ersten wird die Verschiebung der horizontalen Koordinate angegeben und mit der zweiten die vertikale Verschieben. 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 `new`-Rechteck durchgeführt. Bei einer Kollision werden wie eben die Richtungswerte verändert. Falls keine Kollision mit dem Rand vorliegt, wird `newpos` zu unserem neuen Rechteck für den Verteidiger (Zeile 44).

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



Abbildung 2.14: Der Verteidiger bewegt sich und prallt ab

### Was war neu?

- Richtung und Geschwindigkeit in Variablen kodieren.
- Richtungswechsel durch Vorzeichenwechsel
- Kollisionserkennung kann durch Vergleich von Positionsangaben erfolgen.
- `blit()` verwendet ein Rect-Objekt.
- `pygame.Rect`:  
<https://www.pygame.org/docs/ref/rect.html>
- `pygame.Rect.move()`:  
<https://www.pygame.org/docs/ref/rect.html#pygame.Rect.move>
- `pygame.Surface.get_rect()`:  
[https://www.pygame.org/docs/ref/surface.html#pygame.Surface.get\\_rect](https://www.pygame.org/docs/ref/surface.html#pygame.Surface.get_rect)

## 2.5 Sprite-Klasse

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

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

Sprite

Formulieren wir also die Klasse `Defender` als eine Kindklasse von `Sprite` (Zeile 15).

### Quelltext 2.12: Sprites (1), Version 1.0

---

```
15  class Defender(pygame.sprite.Sprite):                      # Kindklasse von Sprite
16
17      def __init__(self) -> None:                            # Konstruktor
18          super().__init__()
19          self.image = pygame.image.load("images/defender01.png").convert_alpha()
20          self.image = pygame.transform.scale(self.image, (30, 30))
21          self.rect = self.image.get_rect()
22          self.rect.centerx = Settings.window['width'] // 2
23          self.rect.bottom = Settings.window['height'] - 5
24          self.speed = 2
25          self.direction = 1
26
27      def update(self) -> None:                           # Zustandsberechnung
28          newpos = self.rect.move(self.direction * self.speed, 0)
29          if newpos.right >= Settings.window['width']:
30              self.change_direction()
31          elif newpos.left <= 0:
32              self.change_direction()
33          else:
34              self.rect = newpos
35
36      def draw(self, screen) -> None:                     # Malen
37          screen.blit(self.image, self.rect)
38
39      def change_direction(self) -> None:                 # OO style
40          self.direction *= -1
```

---

Die Zeilen des Konstruktors (Zeile 17ff.) entsprechen genau 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 17ff. werden die Kollisionserkennungen und die Zustandsänderungen formuliert. Auch sollte es inhaltlich keine Probleme geben. Neu ist lediglich der Aufruf der Methode `change_direction()`. Diese Methode (Zeile 39) ist mehr OO-like also die vorherige Version. In der objektorientierten Programmierung werden Algorithmen nicht direkt programmiert, sondern man sendet an das Objekt Nachrichten, und diese werden dann intern – und von außen nicht sichtbar wie – umgesetzt. Hier bedeutet dies, dass ich an der entsprechenden Stelle nicht den Richtungswechsel direkt durchführe, sondern mir selbst die Nachricht zusende, dass die Richtung geändert werden muss.

Mit der Methode `draw()` (Zeile 36) wird die Bildschirmausgabe gekapselt.

### Quelltext 2.13: Sprites (2), Version 1.0

---

```
50     clock = pygame.time.Clock()                      # Objekt anlegen
51     defender = Defender()
52
53     running = True
54     while running:
55         clock.tick(Settings.fps)
56         # Events
57         for event in pygame.event.get():
58             if event.type == pygame.QUIT:
59                 running = False
60
61         # Update
62         defender.update()                         # Aufruf
63
64         # Draw
65         screen.fill((255, 255, 255))
66         defender.draw(screen)                    # Aufruf
67         pygame.display.flip()
```

---

Die Verwenung der Klasse `Defender` ist nun denkbar einfach geworden. In der Zeile 51 wird ein Objekt der Klasse erzeugt. In Zeile 62 wird `update()` aufgerufen und in Zeile 66 `draw()`.

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

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

Los geht's: Da wir die Kollisionsprüfung anders organisieren, wird erstmal das `update()` wieder einfach. Dabei wird in Zeile 27 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 2.14: Sprites (1), Version 1.1

---

```
26     def update(self) -> None:
27         self.rect.move_ip(self.direction * self.speed, 0) # Vereinfacht
```

---

Damit die Ränder mal sichtbar werden und ich die Kollision besser erkennbar mache, werden die Ränder nun zu zwei Steinwänden rechts und links.

### Quelltext 2.15: Sprites (2), Version 1.1

---

```
37 class Border(pygame.sprite.Sprite):
38
39     def __init__(self, leftright) -> None:
40         super().__init__()
41         self.image = pygame.image.load("images/brick01.png").convert_alpha()
42         self.image = pygame.transform.scale(self.image, (35, Settings.window['height']))
43         self.rect = self.image.get_rect()
44         if leftright == 'right':
45             self.rect.left = Settings.window['width'] - self.rect.width
```

```
46     def update(self) -> None:
47         pass
48
49
50     def draw(self, screen) -> None:
51         screen.blit(self.image, self.rect)
```

---

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

---

### Quelltext 2.16: Sprites (3), Version 1.1

---

```
64     defender = Defender()
65     border_left = Border('left')
66     border_right = Border('right')
```

---

Bisher war alles easy.

---

### Quelltext 2.17: Sprites (4), Version 1.1

---

```
76     # Update
77     if pygame.sprite.collide_rect(defender, border_left):
78         defender.change_direction()
79     elif pygame.sprite.collide_rect(defender, border_right):
80         defender.change_direction()
81     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 rechtes 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 2.18: Sprites (1), Version 1.2

---

```
60     defender = pygame.sprite.GroupSingle(Defender())
61     all_border = pygame.sprite.Group()
62     all_border.add(Border('left'))
63     all_border.add(Border('right'))
64
65     running = True
66     while running:
67         clock.tick(Settings.fps)
68         # Events
69         for event in pygame.event.get():
70             if event.type == pygame.QUIT:
71                 running = False
72
73         # Update
74         if pygame.sprite.spritecollide(defender.sprite, all_border, False): # !
75             defender.sprite.change_direction()
76         defender.update()
77
```

---

```
78     # Draw
79     screen.fill((255, 255, 255))
80     defender.draw(screen)
81     all_border.draw(screen)           # Mit einem Rutsch
82     pygame.display.flip()
83
84     pygame.quit()
```

---

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

Group

Die erste Stelle ist Zeile 74 und dort wird eine andere Version der Kollisionsprüfung verwendet: `pygame.sprite.spritecollide()`. Der erste Parameter ist *ein Sprite*-Objekt. In unserem Fall ist es der Verteidiger. Der zweite Parameter ist eine Spritegruppe mit allen `Border`-Objekten. Also wird der Verteidiger mit allen Mitgliedern der Gruppe auf Kollisionen überprüft. Dies funktioniert nur, wenn alle Sprites ein `Rect`-Objekt mit dem Namen `rect` als Attribut haben. Der dritte Parameter – hier `False` – steuert, ob das kollidierende Sprite aus der Liste entfernt werden soll. Dieses Feature ist in Spielen recht interessant, will man doch beispielsweise Raumschiffe, die von einem Felsen getroffen wurden, löschen.

spritecollide()

Die zweite Stelle ist Zeile 81. 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`) entfernen, wodurch schon wieder alles einfacher wird.

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 27f.).

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 2.19: Game-Klasse

---

```
52 class Game(object):
53
54     def __init__(self) -> None:
```

```
55         os.environ['SDL_VIDEO_WINDOW_POS'] = "10, 50"
56     pygame.init()
57     self.screen = pygame.display.set_mode(Settings.window_dim())
58     pygame.display.set_caption("Sprite")
59     self.clock = pygame.time.Clock()
60     self.defender = pygame.sprite.GroupSingle(Defender())
61     self.all_border = pygame.sprite.Group()
62     self.all_border.add(Border('left'))
63     self.all_border.add(Border('right'))
64     self.running = False
65
66     def run(self) -> None:
67         self.running = True
68         while self.running:
69             self.clock.tick(Settings.fps)
70             self.watch_for_events()
71             self.update()
72             self.draw()
73         pygame.quit()
74
75     def watch_for_events(self) -> None:
76         for event in pygame.event.get():
77             if event.type == pygame.QUIT:
78                 self.running = False
79
80     def update(self) -> None:
81         if pygame.sprite.spritecollide(self.defender.sprite, self.all_border, False):
82             self.defender.sprite.change_direction()
83             self.defender.update()
84
85     def draw(self) -> None:
86         self.screen.fill((255, 255, 255))
87         self.defender.draw(self.screen)
88         self.all_border.draw(self.screen)
89         pygame.display.flip()
90
91
92 if __name__ == '__main__':
93     game = Game()
94     game.run()
```

---

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.

## Was war neu?

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

- `pygame.sprite.Sprite`:  
<https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.Sprite>
- `pygame.sprite.Group`:  
<https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.Group>
- `pygame.sprite.GroupSingle`:  
<https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.GroupSingle>
- `pygame.sprite.GroupSingle.sprite`:  
<https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.GroupSingle>
- `pygame.sprite.spritecollide()`:  
<https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.spritecollide>
- `pygame.sprite.collide_rect()`:  
[https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.collide\\_rect](https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.collide_rect)
- `pygame.Rect.move_ip()`:  
[https://www.pygame.org/docs/ref/rect.html#pygame.Rect.move\\_ip](https://www.pygame.org/docs/ref/rect.html#pygame.Rect.move_ip)

## 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](#)).

Zunächst bereiten wir die Verteidiger-Klasse vor bzw. wandeln sie ein wenig ab ([Quelltext 2.20](#)). Das Sprite wird nun nicht mehr unten sondern mittig platziert ([Zeile 21](#)). Das Raumschiff soll sich nun auch vertikal bewegen können. Dazu braucht es entweder zwei entsprechende Variablen oder aber ein 2-Tupel. Ich nehme ein 2-Tupel ([Zeile 22](#)), wobei das erste Element der Richtungsvektor der horizontalen und das zweite der vertikalen Richtung ist. Daraus ergeben sich auch die Bedeutungen der Methoden `move_*`(). Der jeweilige Richtungsvektor wird dabei entsprechend der schon oben vorgestellten Semantik gesetzt. In der Methode `update()` werden nun beide Koordinaten berücksichtigt und aktualisiert; analog in `change_direction()`. Bewegen und Stehenbleiben wird einfach dadurch erreicht, dass ich die Geschwindigkeit in `start()` auf 2 bzw. in `stop()` auf 0 setze.

Quelltext 2.20: Bewegung durch Tastatur steuern (1), `Defender`

---

```
14 class Defender(pygame.sprite.Sprite):
15
16     def __init__(self) -> None:
17         super().__init__()
```

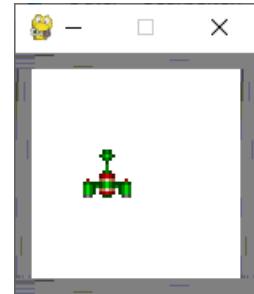


Abb. 2.15: Ränder

```
18         self.image = pygame.image.load("images/defender01.png").convert_alpha()
19         self.image = pygame.transform.scale(self.image, (30, 30))
20         self.rect = self.image.get_rect()
21         self.rect.center = (Settings.window['width'] // 2, Settings.window['height'] //
2) # Zentrum
22         self.direction = (0, 0)                                # 2 Dimensionen
23         self.start()
24         self.move_right()
25
26     def update(self) -> None:
27         self.rect.move_ip(self.direction[0] * self.speed, self.direction[1] * self.speed)
28
29     def draw(self, screen) -> None:
30         screen.blit(self.image, self.rect)
31
32     def move_right(self) -> None:
33         self.direction = (1, 0)
34
35     def move_left(self) -> None:
36         self.direction = (-1, 0)
37
38     def move_up(self) -> None:
39         self.direction = (0, -1)
40
41     def move_down(self) -> None:
42         self.direction = (0, 1)
43
44     def stop(self) -> None:
45         self.speed = 0
46
47     def start(self) -> None:
48         self.speed = 2
49
50     def change_direction(self) -> None:
51         self.direction = (self.direction[0] * -1, self.direction[1] * -1)
```

---

Die Klasse `Border` wird trivialerweise so erweitert, dass alle vier Seiten der Spielfläche nun durch eine Steinwand begrenzt werden (Quelltext 2.21). Dazu wird im Konstruktor abgefragt, auf welcher Seite die Wand hochgezogen werden soll. Rechts und links wird das Bitmap in die Höhe gestreckt und oben und unten in die Breite. Anschließend wird das Rect-Objekt ermittelt und die Position festgelegt.

---

Quelltext 2.21: Bewegung durch Tastatur steuern (2), `Border`

---

```
56 class Border(pygame.sprite.Sprite):
57
58     def __init__(self, whichone) -> None:
59         super().__init__()
60         self.image = pygame.image.load("images/brick01.png").convert_alpha()
61         if whichone == 'right':
62             self.image = pygame.transform.scale(self.image, (10,
63                                         Settings.window['height']))
64             self.rect = self.image.get_rect()
65             self.rect.left = Settings.window['width'] - self.rect.width
66         elif whichone == 'left':
67             self.image = pygame.transform.scale(self.image, (10,
68                                         Settings.window['height']))
69             self.rect = self.image.get_rect()
70             self.rect.left = 0
71         elif whichone == 'top':
72             self.image = pygame.transform.scale(self.image, (Settings.window['width'],
73                                         10))
```

---

```
71         self.rect = self.image.get_rect()
72         self.rect.top = 0
73     elif whichone == 'down':
74         self.image = pygame.transform.scale(self.image, (Settings.window['width'],
75                                                 10))
76         self.rect = self.image.get_rect()
77         self.rect.bottom = Settings.window['height']
```

---

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

Quelltext 2.22: Bewegung durch Tastatur steuern (3), Game-Konstruktor

---

```
91     self.all_border = pygame.sprite.Group()
92     self.all_border.add(Border('left'))
93     self.all_border.add(Border('right'))
94     self.all_border.add(Border('top'))
95     self.all_border.add(Border('down'))
```

---

Kommen wir jetzt zur eigentlichen Tastaturverarbeitung: Das Verwenden einer Taste kann die Ereignistypen `pygame.KEYDOWN` oder `pygame.KEYUP` auslösen. In unserem Beispiel (Zeile 111) 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 2.1 auf der nächsten Seite und Tabelle 2.2 auf Seite 36).

KEYDOWN  
KEYUP

key

Quelltext 2.23: Bewegung durch Tastatur steuern (4), Game.watch\_for\_events()

---

```
107     def watch_for_events(self) -> None:
108         for event in pygame.event.get():
109             if event.type == pygame.QUIT:
110                 self.running = False
111             elif event.type == pygame.KEYDOWN:          # Taste drücken
112                 if event.key == pygame.K_ESCAPE:        # Boss-Taste
113                     self.running = False
114                 elif event.key == pygame.K_RIGHT:       # Pfeiltasten
115                     self.defender.sprite.move_right()
116                 elif event.key == pygame.K_LEFT:
117                     self.defender.sprite.move_left()
118                 elif event.key == pygame.K_UP:
119                     self.defender.sprite.move_up()
120                 elif event.key == pygame.K_DOWN:
121                     self.defender.sprite.move_down()
122                 elif event.key == pygame.K_SPACE:        # Leerzeichen-Taste
123                     self.defender.sprite.stop()
124                 elif event.key == pygame.K_r:
125                     if event.mod & pygame.KMOD_LSHIFT:    # Shift-Taste
126                         self.defender.sprite.stop()
127                     else:
128                         self.defender.sprite.start()
```

---

Fangen wir mit der Boss-Taste an. In Zeile 112 wird über die Konstante `K_ESCAPE` abgefragt, ob die gedrückte Taste die Escape-Taste ist. Wie beim Weg-Xen wird danach einfach das Flag der Hauptprogrammschleife auf `False` gesetzt. Probieren Sie es aus!

K\_ESCAPE

Danach werden mit Hilfe von `K_LEFT`, `K_RIGHT`, `K_UP` und `K_DOWN` ab Zeile 114ff. die vier

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 122 gestoppt.

`K_SPACE`

Um den Einsatz der Shift-Taste (Umschalttaste) mal zu demonstrieren, habe ich hier das `r` doppelt belegt (Zeile 125). 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?

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

Tabelle 2.1: Liste von vordefinierten Tastaturkonstanten

Konstante	Bedeutung	Beschreibung
<code>K_BACKSPACE</code>	<code>\b</code>	Löschen (backspace)
<code>K_TAB</code>	<code>\t</code>	Tabulator
<code>K_CLEAR</code>		Leeren
<code>K_RETURN</code>	<code>\r</code>	Eingabe (return, enter)
<code>K_PAUSE</code>		Pause
<code>K_ESCAPE</code>	<code>^ [</code>	Abbruch (escape)
<code>K_SPACE</code>		Leerzeichen (space)
<code>K_EXCLAIM</code>	<code>!</code>	Ausrufezeichen
<code>K_QUOTEDBL</code>	<code>"</code>	Gänsefußchen
<code>K_HASH</code>	<code>#</code>	Doppelkreuz (hash)
<code>K_DOLLAR</code>	<code>\$</code>	Dollar
<code>K_AMPERSAND</code>	<code>&amp;</code>	Kaufmannsund
<code>K_QUOTE</code>	<code>'</code>	Hochkomma
<code>K_LEFTPAREN</code>	<code>(</code>	Linke runde Klammer
<code>K_RIGHTPAREN</code>	<code>)</code>	Rechte runde Klammer
<code>K_ASTERISK</code>	<code>*</code>	Sternchen
<code>K_PLUS</code>	<code>+</code>	Plus
<code>K_COMMA</code>	<code>,</code>	Komma
<code>K_MINUS</code>	<code>-</code>	Minus
<code>K_PERIOD</code>	<code>.</code>	Punkt
<code>K_SLASH</code>	<code>/</code>	Schrägstrich

Tabelle 2.1: Liste von vordefinierten Tastaturkonstanten (Fortsetzung)

Konstante	Bedeutung	Beschreibung
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
K_l	l	l
K_m	m	m
K_n	n	n
K_o	o	o
K_p	p	p
K_q	q	q

Tabelle 2.1: Liste von vordefinierten Tastaturkonstanten (Fortsetzung)

Konstante	Bedeutung	Beschreibung
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
K_PAGEUP		Hochblättern
K_PAGEDOWN		Runterblättern
K_F1		F1
K_F2		F2
K_F3		F3
K_F4		F4

Tabelle 2.1: Liste von vordefinierten Tastaturkonstanten (Fortsetzung)

Konstante	Bedeutung	Beschreibung
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 Alterntivtaste
K_LALT		Linke Alterntivtaste
K_RMETA		Rechte Metataste
K_LMETA		Linke Metataste
K_LSUPER		Linke Windowstaste
K_RSUPER		Rechte Windowstaste
K_MODE		AltGr Umschalter
K_HELP		Hilfe
K_PRINT		Bildschirmdruck/Screenshot
K_SYSREQ		Systemabfrage
K_BREAK		Abbruch/Unterbrechung
K_MENU		Menü
K_POWER		Ein-/Ausschalten
K_EURO	€	Euro-Währungszeichen
K_AC_BACK		Android Zurückschalter

Tabelle 2.2: Liste von vordefinierten Konstanten zur Tastaturschaltung

Konstante	Beschreibung
KMOD_NONE	Keine Belegungstaste gedrückt
KMOD_LSHIFT	Linke Umschalttaste

Tabelle 2.2: Liste von vordefinierten Konstanten zur Tastaturschaltung (Fortsetzung)

Konstante	Beschreibung
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 2.16: Textausgabe mit Fonts

Bei vielen Spielen werden Informationen nicht nur symbolisch auf die Spielfläche gebracht (z.B. drei Männchen für drei Leben), sondern auch in Schriftform. Eine Möglichkeit dies

zu erreichen, ist die Textausgabe mit Hilfe installierter Fonts. Dabei wird zuerst ein Font-Objekt erstellt und durch ihn ein Surface-Objekt mit dem Text erzeugt ([gerendert](#)). Ich habe dies für ein kleines Beispiel in eine Klasse gekapselt, die Sie ja nach Belieben aufbohren oder anpassen können.

Rendern

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

Quelltext 2.24: Text mit Fonts ausgeben (1), Präambel

---

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

---

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 36 liefert mir die vom Betriebssystem eingestellte Zeichsatzvorgabe. Die Schriftgröße (`fontsize`) legen wir nach Bedarf einfach fest.

Font

get\_default\_font()

Quelltext 2.25: Text mit Fonts ausgeben (2), `TextSprite`

---

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

---

Schauen wir uns nun den Konstruktor etwas genauer an. Die beiden 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 23. 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 26 mit Hilfe von `pygame.font.render()` das Surface-Objekt erzeugen (Zeile 29). Anschließend wird vom Bitmap das Rechteck ermittelt und das Zentrum des Rechtecks auf die gewünschte Position verschoben.

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 54). Darunter soll durch `info` ausgegeben werden, mit welcher Schriftgröße und -farbe der Gruß erzeugt wurde (Zeile 54).

---

### Quelltext 2.26: Text mit Fonts ausgeben (3), Hauptprogramm

---

```
47 if __name__ == '__main__':
48     os.environ['SDL_VIDEO_WINDOW_POS'] = "500, 150"
49     pygame.init()
50     clock = pygame.time.Clock()
51     screen = pygame.display.set_mode(Settings.get_dim())
52     pygame.display.set_caption("Textausgabe mit Fonts")
53
54     hello = TextSprite(24, [255,255,255], (Settings.window['width']//2,
55         Settings.window['height']//2)) # Gruß
55     info = TextSprite(12, [255,0,0], (Settings.window['width']//2,
56         Settings.window['height']-20)) # Fontinfo
56     all_sprites = pygame.sprite.Group()
57     all_sprites.add(hello, info)
58
59     running = True
60     while running:
61         clock.tick(Settings.fps)
62         for event in pygame.event.get():
63             if event.type == QUIT:
64                 running = False
65             elif event.type == KEYDOWN:
66                 if event.key == K_ESCAPE:
67                     running = False
68                 elif event.key == K_KP_PLUS or event.key == K_PLUS:      # Größer
69                     hello.fontsize_update(+1)
70                 elif event.key == K_KP_MINUS or event.key == K_MINUS:  # Kleiner
71                     hello.fontsize_update(-1)
72                 elif event.key == K_r:
73                     if event.mod & KMOD_SHIFT:
74                         hello.fontcolor_update((-1, 0, 0))           # Weniger Rot
75                     else:
76                         hello.fontcolor_update((+1, 0, 0))           # Mehr Rot
77                 elif event.key == K_g:
78                     if event.mod & KMOD_SHIFT:
```

```

79             hello.fontcolor_update((0, -1, 0))           # Weniger Grün
80     else:
81         hello.fontcolor_update((0, +1, 0))           # Mehr Grün
82     elif event.key == K_b:
83         if event.mod & KMOD_SHIFT:
84             hello.fontcolor_update((0, 0, -1))       # Weniger Blau
85         else:
86             hello.fontcolor_update((0, 0, +1))       # Mehr Blau
87
88     info.text = f"size={hello.fontsize}, r={hello.fontcolor[0]},"
89     g={hello.fontcolor[1]}, b={hello.fontcolor[2]}"
90     all_sprites.update()
91     screen.fill((200, 200, 200))
92     all_sprites.draw(screen)
93     pygame.display.flip()
94
pygame.quit()

```

---

Dieser Gruß kann durch die Plus- und Minus-Tasten in seiner Größe verändert werden (Zeile 68ff.). Die Tasten **r**, **g** und **b** werden dazu verwendet, den jeweiligen Farbkanal zu manipulieren. Der Großbuchstabe erhöht den Wert (z.B. in Zeile 74), der Kleinbuchstabe reduziert ihn (z.B. in Zeile 76).

In Abbildung 2.16 auf Seite 37 können Sie eine mögliche Darstellung sehen.

## 2.7.2 Fontliste



Abbildung 2.17: Fontliste

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

Quelltext 2.27: Fontliste (1), Präambel, Settings und Textsprite

```
1 import pygame
2 from pygame.constants import (
3     QUIT, K_ESCAPE, KEYDOWN, K_UP, K_DOWN
4 )
5 import os
6
7
8 class Settings:
9     window = {'width': 700, 'height': 300}
10    fps = 60
11
12    @staticmethod
13    def get_dim():
14        return (Settings.window['width'], Settings.window['height'])
15
16
17 class TextSprite(pygame.sprite.Sprite):
18    def __init__(self, fontname, fontsize=24, fontcolor=[255,255,255], text=''):
19        super().__init__()
20        self.image = None
21        self.rect = None
22        self.fontname = fontname
23        self.fontsize = fontsize
24        self.fontcolor = fontcolor
25        self.fontsize_update(0)
26        self.text = f'{self.fontname}: abcdefghijklmnopqrstuvwxyzßöäü0123456789'
27        self.render()
28
29    def render(self):
30        self.image = self.font.render(self.text, True, self.fontcolor)
31        self.rect = self.image.get_rect()
32
33    def fontsize_update(self, step=1):
34        self.fontsize += step
35        self.font = pygame.font.Font(pygame.font.match_font(self.fontname),
36                                    self.fontsize) #
37
38    def fontcolor_update(self, delta):
39        for i in range(3):
40            self.fontcolor[i] = (self.fontcolor[i] + delta[i]) % 256
41
42    def update(self):
43        self.render()
```

---

Die Klasse `TextSprite` wurde nur wenig auf die Bedürfnisse angepasst. Die Klasse `BigImage` hat nur die Aufgabe, alle `FontSprite`-Images als großes Bild zu verwalten. 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 55) ermittelt. Zuerst wird ermittelt, ob ich das obere oder untere Ende des Bitmaps erreicht habe. Falls ja, wird `top` bzw. `bottom` entsprechend gesetzt, so dass immer der ganze Bildschirm gefüllt wird. Ansonsten wird das `offset`-Rechteck nach oben bzw. nach unten verschoben und mit `pygame.Surface.subsurface()` der Ausschnitt ermittelt.

`subsurface()`

---

### Quelltext 2.28: Fontliste (2), BigImage

---

```
45 class BigImage(pygame.sprite.Sprite):
46     def __init__(self):
47         super().__init__()
```

```
48         self.offset = pygame.Rect((0, 0), Settings.get_dim())
49
50     def create_image(self, width, height):
51         self.image_total = pygame.Surface((width, height))
52         self.image_total.fill((200, 200, 200))
53         self.update(0)
54
55     def update(self, delta):                      # Ermittle der Ausschnitt
56         if self.offset.top + delta >= 0:
57             if self.offset.bottom + delta <= self.image_total.get_rect().height:
58                 self.offset.move_ip(0, delta)
59             else:
59                 self.offset.bottom = self.image_total.get_rect().height
60             else:
61                 self.offset.top = 0
62         self.image = self.image_total.subsurface(self.offset)
63         self.rect = self.image.get_rect()
```

---

Und jetzt das Hauptprogramm. Im ersten Teil wird über `pygame.font.get_fonts()` (Zeile 76) eine Liste aller installierten Fontnamen ermittelt. Dieser Name wird dem Konstruktor von `TestSprite` übergeben. Mit Hilfe der Methode `pygame.font.match_font()` (Zeile 35) 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.

`get_fonts()`  
`match_font()`

### Quelltext 2.29: Fontliste (3), Hauptprogramm (1)

---

```
67 if __name__ == '__main__':
68
69     os.environ['SDL_VIDEO_WINDOW_POS'] = "650, 40"
70
71     pygame.init()
72     clock = pygame.time.Clock()
73     screen = pygame.display.set_mode(Settings.get_dim())
74     pygame.display.set_caption("Fontliste")
75
76     fonts = pygame.font.get_fonts()                      # Ermittle installierte Fonts
77
78     list_of_fontsprites = pygame.sprite.Group()
79     height = 0
80     width = 0
81     for name in fonts:
82         try:
83             t = TextSprite(name, 24, [0,0,255])
84             t.rect.top = height
85             height += t.rect.height
86             width = t.rect.width if t.rect.width > width else width
87             list_of_fontsprites.add(t)
88         except OSError as err:
89             print(f"OS error {err}")
90         except pygame.error as perr:
91             print(f"Pygame error: {perr} with font {name}")
92
93     bigimage = pygame.sprite.GroupSingle(BigImage())
94     bigimage.sprite.create_image(width, height)
95     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 95).

### Quelltext 2.30: Fontliste, Hauptprogramm (2)

```
97     running = True
98     while running:
99         clock.tick(60)
100        for event in pygame.event.get():
101            if event.type == QUIT:
102                running = False
103            elif event.type == KEYDOWN:
104                if event.key == K_ESCAPE:
105                    running = False
106                if event.key == K_UP:
107                    bigimage.update(-Settings.window['height']//3)
108                if event.key == K_DOWN:
109                    bigimage.update(Settings.window['height']//3)
110
111        bigimage.draw(screen)
112        pygame.display.flip()
113
114    pygame.quit()
```

---

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

### Was war neu?

- `pygame.font.Font`:  
<https://www.pygame.org/docs/ref/font.html>
- `pygame.font.get_default_font()`:  
[https://www.pygame.org/docs/ref/font.html#pygame.font.get\\_default\\_font](https://www.pygame.org/docs/ref/font.html#pygame.font.get_default_font)
- `pygame.font.get_fonts()`:  
[https://www.pygame.org/docs/ref/font.html#pygame.font.get\\_fonts](https://www.pygame.org/docs/ref/font.html#pygame.font.get_fonts)
- `pygame.font.match_font()`:  
[https://www.pygame.org/docs/ref/font.html#pygame.font.match\\_font](https://www.pygame.org/docs/ref/font.html#pygame.font.match_font)
- `pygame.font.Font.render()`:  
<https://www.pygame.org/docs/ref/font.html#pygame.font.Font.render>  
`pygame.Surface.subsurface()`:  
<https://www.pygame.org/docs/ref/surface.html#pygame.Surface.subsurface>

## 2.8 Textausgabe mit Bitmaps

Oft erfolgen Textausgaben nicht über Fonts, sondern über eine **Spritelib**. In einer solchen befinden sich dann Schriftzeichen, Symbole oder Ziffern, die dann meist auch in einem besonderen dem Spiel angepassten Design sind. In Abbildung 2.18 auf der nächsten Seite 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 2.20 auf Seite 49).

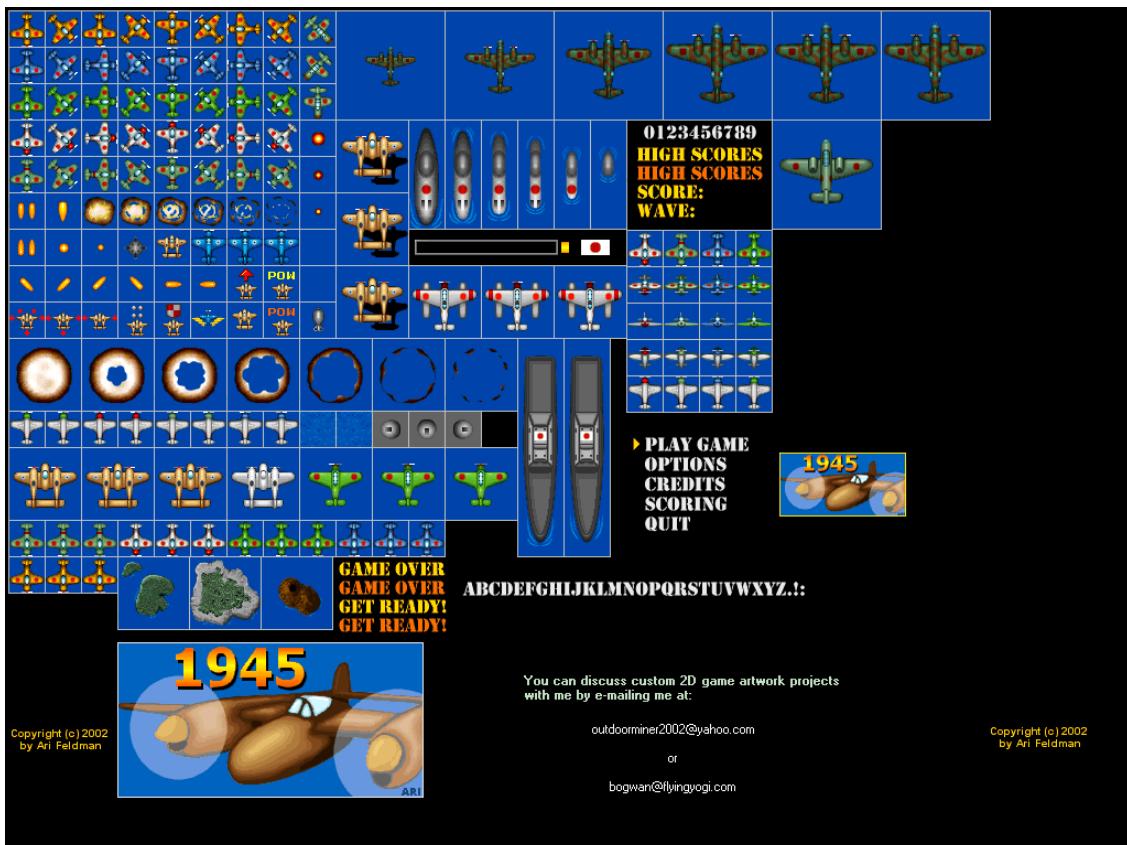


Abbildung 2.18: Beispiel für eine Spritelib

Der erste Teil von Quelltext 2.31 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 2.31: Textbitmaps (1), Präambel und `Settings`

---

```

1 import pygame
2 from pygame import KEYDOWN, QUIT, K_ESCAPE

```

```
3 import os
4
5
6 class Settings:
7     window = {'width': 700, 'height': 650}
8     path = {}
9     path['file'] = os.path.dirname(os.path.abspath(__file__))
10    path['image'] = os.path.join(path['file'], "images")
11
12    @staticmethod
13    def dim():
14        return (Settings.window['width'], Settings.window['height'])
15
16    @staticmethod
17    def filepath(name):
18        return os.path.join(Settings.path['file'], name)
19
20    @staticmethod
21    def imagepath(name):
22        return os.path.join(Settings.path['image'], name)
```

---

Die Klasse **Spritelib** wird eigentlich nur als Kontainer 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 2.19 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 2.19 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 2.19 können Sie die Abstände sehen. Hier sind es jeweils 14 *px*.

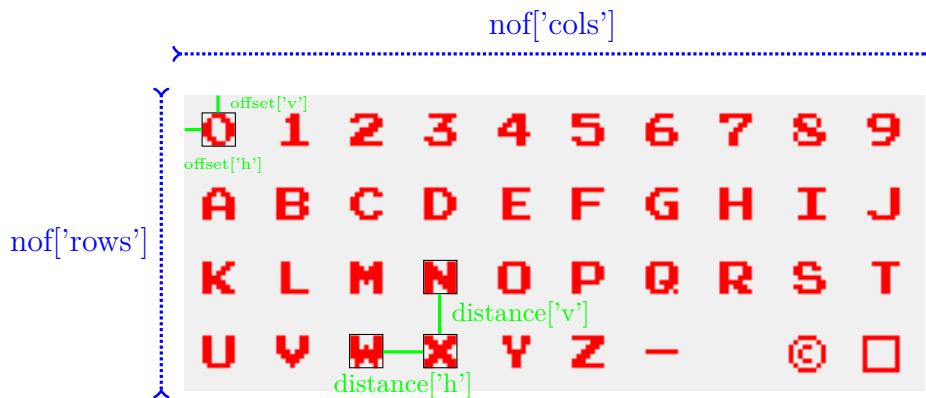


Abbildung 2.19: Bedeutung der Angaben in Spritelib

Quelltext 2.32: Textbitmaps (2), Spritelib

---

```

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

```

---

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

Quelltext 2.33: Textbitmaps (3): Konstruktor von `Letters`


---

```

40 class Letters(object):
41     def __init__(self, spritelib, colornumber) -> None:
42         super().__init__()
43         self.spritelib = spritelib
44         self.letters = {}
45         self.create_letter_bitmap(colornumber)

```

---

In der Methode `create_letter_bitmap()` werden nun die einzelnen Sprites ausgestanzt und in ein Dictionary abgelegt. Die Indizes des Dictionaries werden in Zeile 48 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 50 wird die Position, also die Pixelkoordinaten des ersten Sprites ausgerechnet. Versuchen Sie doch selbst anhand der Angaben in Abbildung 2.19 auf der vorherigen Seite die Arithmetik nachzuvollziehen! Nur Mut, sie ist nicht schwierig, sondern nur lang.

Ab Zeile 51 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 52 wird die obere Koordinate und in Zeile 54 die linke Koordinate der Position berechnet. Wenn Sie Zeile 50 verstanden haben, sollten diese beiden Berechnungen keine Schwierigkeiten mehr bereiten. Höhe und Breite in Zeile 54 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 2.34: Textbitmaps (4): `create_letter_bitmap()` von Letters

---

```
47     def create_letter_bitmap(self, colordnumber):
48         lettername = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c',
49                     'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r',
50                     's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '-', ' ', 'copy', 'square') #
51         index = 0
52         startpos = (self.spritelib.offset['h'], self.spritelib.offset['v'] + colordnumber
53                     * self.spritelib.nof['rows'] * (self.spritelib.letter['height'] +
54                     self.spritelib.distance['v'])) #
55         for row in range(self.spritelib.nof['rows']):                      # Zeilen
56             for col in range(self.spritelib.nof['cols']):                      # Spalten
57                 left = startpos[0] + col * (self.spritelib.letter['width'] +
58                     self.spritelib.distance['h']) #
59                 top = startpos[1] + row * (self.spritelib.letter['height'] +
60                     self.spritelib.distance['v']) #
61                 width, height = self.spritelib.letter.values()      # Größe
62                 r = pygame.Rect(left, top, width, height)
63                 self.letters[lettername[index]] = self.spritelib.image.subsurface(r) #
64                 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 2.35: Textbitmaps (5): `get_letter()` und `get_text()` von Letters

---

```
60     def get_letter(self, letter):
61         if letter in self.letters:
62             return self.letters[letter]
63         else:
```

```
64         return self.letters['square']
65
66     def get_text(self, text):
67         l = len(text) * self.sprite.lib.letter['width']
68         h = self.sprite.lib.letter['height']
69         bitmap = pygame.Surface((l, h))
70         bitmap.set_colorkey((0, 0, 0))
71         for a in range(len(text)):
72             bitmap.blit(self.get_letter(text[a]), (a * self.sprite.lib.letter['width'], 0))
73
74     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 94: 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 96: 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 2.36: Textbitmaps (6): `TextBitmaps`

---

```
76 class TextBitmaps(object):
77     def __init__(self):
78         pygame.init()
79         self.screen = pygame.display.set_mode(Settings.dim())
80         pygame.display.set_caption('Textausgabe mit Bitmaps')
81         self.clock = pygame.time.Clock()
82         self.filename = "chars.png"
83         self.running = False
84         self.input = ""
85
86     def watch_for_events(self):
87         for event in pygame.event.get():
88             if event.type == QUIT:
89                 self.running = False
90             elif event.type == KEYDOWN:
91                 if event.key == K_ESCAPE:
92                     self.running = False
93                 elif event.key == pygame.K_BACKSPACE:
94                     self.input = self.input[:-1]           # Letztes Zeichen abschneiden
95                 else:
96                     self.input += event.unicode        # Tastaturwert als unicode-Zeichen
97
98     def run(self):
99         sprite.lib = Spritelib(self.filename)
100        letters = Letters(sprite.lib, 2)
101        self.running = True
102        while self.running:
103            self.clock.tick(60)
104            self.watch_for_events()
105            self.screen.fill((200, 200, 200))
106            self.screen.blit(letters.get_text(self.input), (400, 200))
107            sprite.lib.draw(self.screen)
108            pygame.display.flip()
109
110        pygame.quit()
```

---

Und der Vollständigkeit halber:

---

Quelltext 2.37: Textbitmaps (7): Hauptprogramm

---

```
113 if __name__ == '__main__':
114     os.environ['SDL_VIDEO_WINDOW_POS'] = "10, 50"
115
116     demo = TextBitmaps()
117     demo.run()
```

---

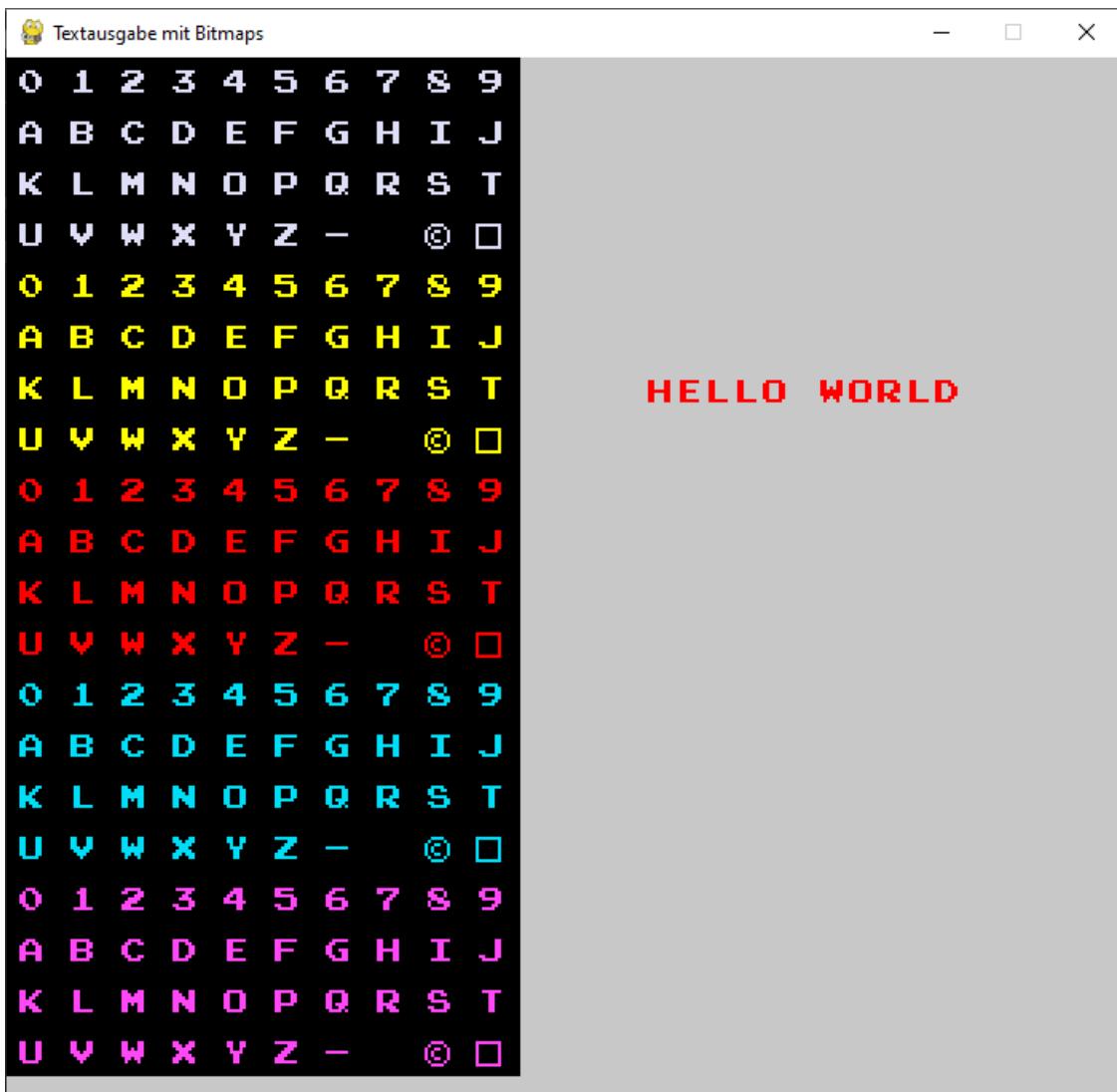


Abbildung 2.20: Textausgabe mit Bitmaps

## Was war neu?

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

## 2.9 Kollisionserkennung

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

- **Rechtecküberschneidung:** Wir haben schon bei 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
```

---

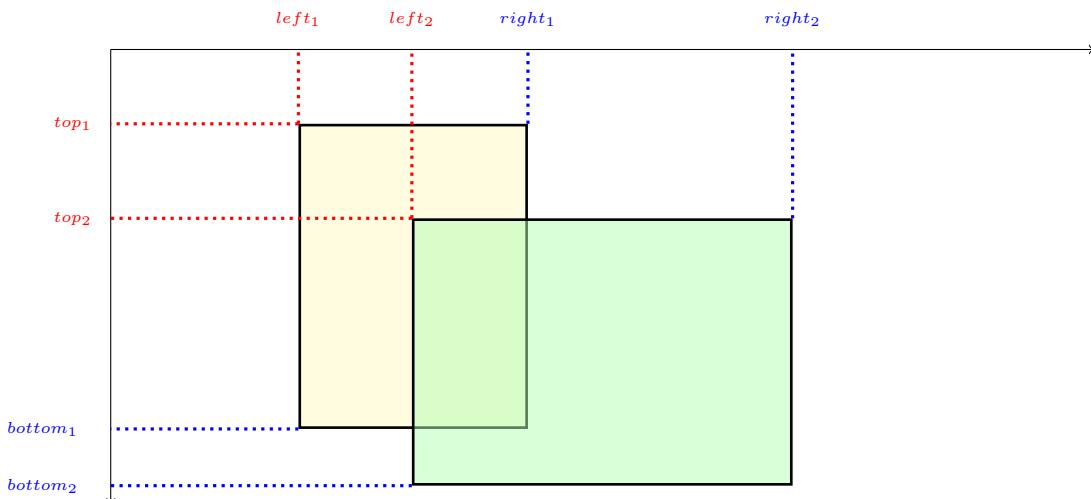


Abbildung 2.21: Kollisionserkennung mit Rechtecken

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

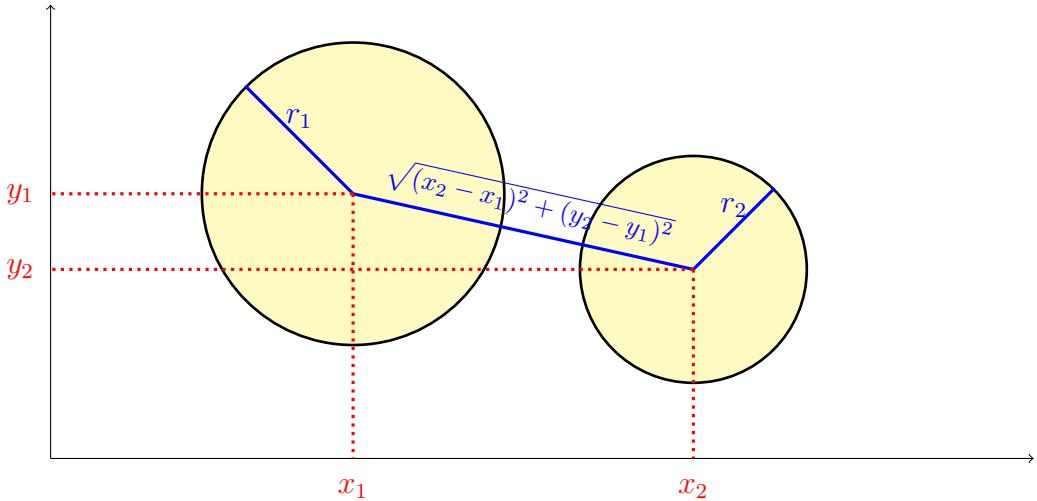


Abbildung 2.22: Kollisionserkennung mit Kreisen

- **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 innherhalb 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 2.23 auf der nächsten Seite sehen wir vier Sprites: eine Mauer, ein Raumschiff, ein Monster und ein Geschoss. Keine der Sprites berühren sich.

In Abbildung 2.24 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



Abbildung 2.23: Kollisionsprüfung: 4 Sprites ohne Kollision

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.

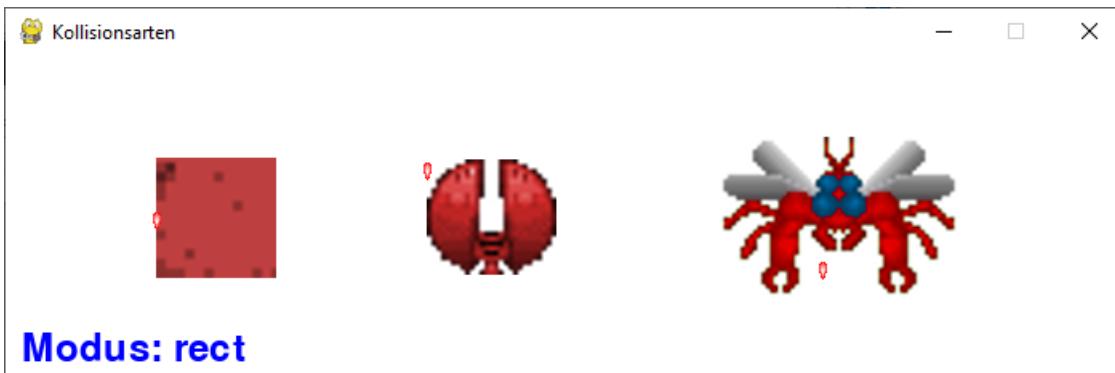


Abbildung 2.24: Kollisionsprüfung durch Rechtecke (Montage)

Anders sieht es aus, wenn wir die Kollision durch die Innenkreise bestimmen lassen (Abbildung 2.25 auf der nächsten Seite). 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 2.26 auf der nächsten Seite). 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 Pixeln. Probieren Sie es ruhig aus, das Geschoss mal nach rechts bzw. links zu bewegen, und Sie werden die pixelgenaue Kollisionserkennung anhand des Farbwechs-



Abbildung 2.25: Kollisionsprüfung durch Kreise (Montage)

sels sofort sehen.



Abbildung 2.26: Kollisionsprüfung durch Masken (Montage)

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

---

### Quelltext 2.38: Kollisionsarten (1): Präambel und `Settings`

---

```
1 import pygame
2 import os
3
4
5 class Settings(object):
6     window = {'width':700, 'height':200}
7     fps = 60
8     title = "Kollisionsarten"
9     path = {}
10    path['file'] = os.path.dirname(os.path.abspath(__file__))
11    path['image'] = os.path.join(path['file'], "images")
12
13    @staticmethod
14    def dim():
15        return (Settings.window['width'], Settings.window['height'])
```

```
17     @staticmethod
18     def filepath(name):
19         return os.path.join(Settings.path['file'], name)
20
21     @staticmethod
22     def imagepath(name):
23         return os.path.join(Settings.path['image'], name)
24     modus = "rect"
```

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 Sprite zwei Bilder geladen: eines für den Zustand *nicht getroffen* und eines für *getroffen*.

### Quelltext 2.39: Kollisionsarten (2): **Obstacle**

---

```
27 class Obstacle(pygame.sprite.Sprite):
28     def __init__(self, filename1, filename2) -> None:
29         super().__init__()
30         self.image_normal =
31             pygame.image.load(Settings.imagepath(filename1)).convert_alpha()
32         self.image_hit = pygame.image.load(Settings.imagepath(filename2)).convert_alpha()
33         self.image = self.image_normal
34         self.rect = self.image.get_rect()                      # Rechteck
35         self.mask = pygame.mask.from_surface(self.image)      # Maske
36         self.radius = self.rect.width // 2                   # Innenkreis
37         self.rect.centery = Settings.window['height'] // 2
38         self.hit = False
39
40     def update(self):
41         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 2.40: Kollisionsarten (3): **Bullet**

---

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

```
44     def __init__(self, picturefile) -> None:
45         super().__init__()
46         self.image = pygame.image.load(Settings.imagepath(picturefile)).convert_alpha()
47         self.rect = self.image.get_rect()
48         self.radius = self.rect.width // 2
49         self.mask = pygame.mask.from_surface(self.image)
50         self.rect.center = (10, 10)
51         self.directions = {'stop':(0, 0), 'down':(0, 1), 'up':(0, -1), 'left':(-1, 0),
52                           'right':(1, 0)}
53         self.set_direction('stop')
54
55     def update(self):
56         self.rect.move_ip(self.speed)
57
58     def set_direction(self, direction):
59         self.speed = self.directions[direction]
```

---

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

Quelltext 2.41: Kollisionsarten (4): Konstruktor von `Game`, Konstruktor

---

```
61 class Game(object):
62     def __init__(self) -> None:
63         super().__init__()
64         os.environ['SDL_VIDEO_WINDOW_POS'] = "10, 50"
65         pygame.init()
66         pygame.display.set_caption(Settings.title)
67         self.font = pygame.font.Font(pygame.font.get_default_font(), 24)
68         self.screen = pygame.display.set_mode(Settings.dim())
69         self.clock = pygame.time.Clock()
70         self.bullet = pygame.sprite.GroupSingle(Bullet("shoot.png"))
71         self.all_obstacles = pygame.sprite.Group()
72         self.all_obstacles.add(Obstacle("brick1.png", "brick2.png"))
73         self.all_obstacles.add(Obstacle("raumschiff1.png", "raumschiff2.png"))
74         self.all_obstacles.add(Obstacle("alienbig1.png", "alienbig2.png"))
75         self.running = False
```

---

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

Quelltext 2.42: Kollisionsarten (5): `run()` und `watch_for_events()` von `Game`

---

```
77     def run(self):
78         self.resize()
79
80         self.running = True
81         while self.running:
82             self.clock.tick(Settings.fps)
83             self.watch_for_events()
84             self.update()
85             self.draw()
86
87         pygame.quit()
88
89     def watch_for_events(self):
90         for event in pygame.event.get():
91             if event.type == pygame.QUIT:
92                 self.running = False
93             elif event.type == pygame.KEYDOWN:
94                 if event.key == pygame.K_ESCAPE:
95                     self.running = False
```

```
96         elif event.key == pygame.K_DOWN:
97             self.bullet.sprite.set_direction('down')
98         elif event.key == pygame.K_UP:
99             self.bullet.sprite.set_direction('up')
100        elif event.key == pygame.K_LEFT:
101            self.bullet.sprite.set_direction('left')
102        elif event.key == pygame.K_RIGHT:
103            self.bullet.sprite.set_direction('right')
104        elif event.key == pygame.K_r:
105            Settings.modus = "rect"
106        elif event.key == pygame.K_c:
107            Settings.modus = "circle"
108        elif event.key == pygame.K_m:
109            Settings.modus = "mask"
110    elif event.type == pygame.KEYUP:
111        self.bullet.sprite.set_direction('stop')
```

---

Ebenso so update() und draw();

---

### Quelltext 2.43: Kollisionsarten (6): update() und draw() von Game

---

```
113     def update(self):
114         self.check_for_collision()
115         self.bullet.update()
116         self.all_obstacles.update()
117
118     def draw(self):
119         self.screen.fill((255, 255, 255))
120         self.all_obstacles.draw(self.screen)
121         self.bullet.draw(self.screen)
122         text_surface_modus = self.font.render("Modus: {}".format(Settings.modus), True,
123             (0, 0, 255))
124         self.screen.blit(text_surface_modus, dest=(10, Settings.window['height']-30))
125         pygame.display.flip()
```

---

Die Methode `resize()` hat nichts mit der eigentlichen Kollisionsprüfung zu tun, sondern soll nur sicherstellen, dass die `Obstacle`-Objekte äquidistant auf die Fensterbreite verteilt werden. Die erste `for`-Schleife ermittelt mir die Summe der Breiten der `Obstacle`-Objekte. Diese Info brauche ich, um in Zeile 130 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 2.44: Kollisionsarten (7): resize() von Game

---

```
126     def resize(self):
127         total_width = 0
128         for s in self.all_obstacles:
129             total_width += s.rect.width
130         padding = (Settings.window['width'] - total_width) // 4      # Abstand
131         for i in range(len(self.all_obstacles)):
132             if i == 0:
133                 self.all_obstacles.sprites()[i].rect.left = padding
134             else:
```

```
135         self.all_obstacles.sprites()[i].rect.left =
136             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 2.45: Kollisionsarten (8): `check_for_collision()` von Game

---

```
137     def check_for_collision(self):
138         if Settings.modus == "circle":
139             for s in self.all_obstacles:
140                 s.hit = pygame.sprite.collide_circle(self.bullet.sprite, s)
141         elif Settings.modus == "mask":
142             for s in self.all_obstacles:
143                 s.hit = pygame.sprite.collide_mask(self.bullet.sprite, s)
144         else:
145             for s in self.all_obstacles:
146                 s.hit = pygame.sprite.collide_rect(self.bullet.sprite, s)
```

---

Noch ein Hinweis: Die letzte `for`-Schleife hätten wir abkürzen können. Die Kollisionsprüfung mit Rechtecken auf eine 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 2.46: Kollisionsarten (9): Variante von `check_for_collision()` von Game

---

```
145         hits = pygame.sprite.spritecollide(self.bullet.sprite, self.all_obstacles,
146                                             False)
146         for s in self.all_obstacles:
147             s.hit = s in hits
```

---

Und zu guter letzt noch der Aufruf:

---

### Quelltext 2.47: Kollisionsarten (10): Der Aufruf von Game

---

```
149 if __name__ == '__main__':
150     game = Game()
151     game.run()
```

---

## Was war neu?

- `pygame.mask.from_surface():`  
[https://www.pygame.org/docs/ref/mask.html#pygame.mask.from\\_surface](https://www.pygame.org/docs/ref/mask.html#pygame.mask.from_surface)
- `pygame.sprite.collide_circle():`  
[https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.collide\\_circle](https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.collide_circle)
- `pygame.sprite.collide_mask():`  
[https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.collide\\_mask](https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.collide_mask)
- `pygame.sprite.collide_rect():`  
[https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.collide\\_rect](https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.collide_rect)
- `pygame.sprite.spritecollide():`  
<https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.spritecollide>

## 2.10 Zeitsteuerung

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

Schauen wir uns zunächst die Bildschirmausgabe von Quelltext 2.48ff. in Abbildung 2.27 auf der nächsten Seite 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.

Bevor wir die Zeitsteuerung selbst angehen, ein kurzer Blick ins Programm. Präambel und die Klasse `Settings` kommen mit nichts Neuem um die Ecke. Lediglich die Kodierung der Bewegungsrichtung wurde mit aufgenommen (Zeile 12).

---

### Quelltext 2.48: Zeitsteuerung (1), Version 1.0: Präambel und `Settings`

---

```
1 import pygame
2 import os
3
4
5 class Settings(object):
```

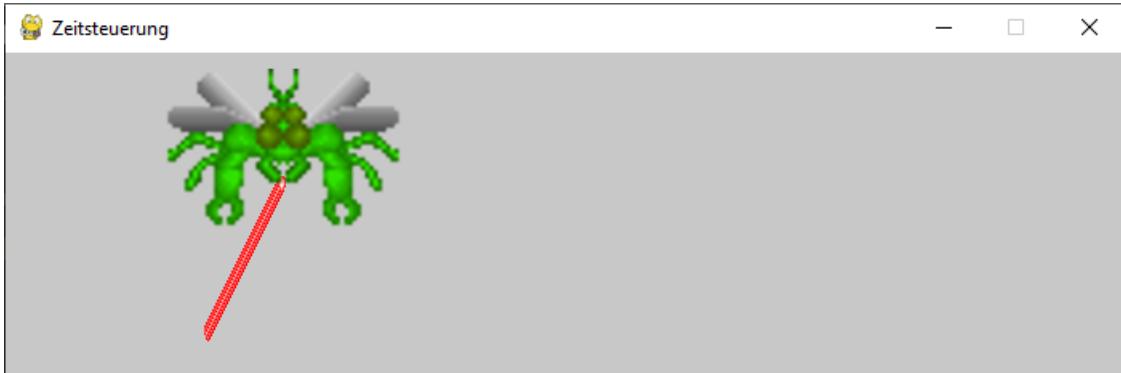


Abbildung 2.27: Feuerball ohne Zeitsteuerung

```

6     window = {'width':700, 'height':200}
7     fps = 60
8     title = "Zeitsteuerung"
9     path = {}
10    path['file'] = os.path.dirname(os.path.abspath(__file__))
11    path['image'] = os.path.join(path['file'], "images")
12    directions = {'stop':(0, 0), 'down':(0, 1), 'up':(0, -1), 'left':(-1, 0),
13                  'right':(1, 0)} #
14
15    @staticmethod
16    def dim():
17        return (Settings.window['width'], Settings.window['height'])
18
19    @staticmethod
20    def filepath(name):
21        return os.path.join(Settings.path['file'], name)
22
23    @staticmethod
24    def imagepath(name):
25        return os.path.join(Settings.path['image'], name)

```

---

Die Klasse `Enemy` liefert auch nichts Weltbewegendes. Lediglich Zeile 37 verwendet etwas Python: Die Werte des Tupel `direction` werden mit dem Skalar `speed` multipliziert und liefern damit den Bewegungsvektor. Die entsprechende Python-Technik nennt sich **List Comprehension**. Die gleiche Technik wird bei der Bewegung des Feuerballs in Zeile 54 nochmal verwendet. Mit 10 Pixel Abstand pendelt der Enemy immer von links nach rechts bzw. umgekehrt.

List Comprehension

 Quelltext 2.49: Zeitsteuerung (2), Version 1.0: `Enemy`


---

```

27 class Enemy(pygame.sprite.Sprite):
28     def __init__(self, filename) -> None:
29         super().__init__()
30         self.image = pygame.image.load(Settings.imagepath(filename)).convert_alpha()
31         self.rect = self.image.get_rect()
32         self.rect.topleft = (10, 10)
33         self.direction = Settings.directions['right']
34         self.speed = 1
35
36     def update(self):
37         self.rect.move_ip([self.speed*x for x in self.direction]) # Liste multiplizieren

```

```
38     if self.rect.left < 10:
39         self.direction = Settings.directions['right']
40     elif self.rect.right >= Settings.window['width'] - 10:
41         self.direction = Settings.directions['left']
```

---

Auch Bullet ist in weiten Teilen eine Wiederholung. Interessant dürfte Zeile 56 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 2.27 auf der vorherigen Seite dadurch erkennen, dass 30 px vor dem unteren Bildschirmrand der Feuerball verschwindet.

`kill()`

### Quelltext 2.50: Zeitsteuerung (3), Version 1.0: Bullet

---

```
44 class Bullet(pygame.sprite.Sprite):
45     def __init__(self, picturefile) -> None:
46         super().__init__()
47         self.image = pygame.image.load(Settings.imagepath(picturefile)).convert_alpha()
48         self.rect = self.image.get_rect()
49         self.rect.center = (10, 10)
50         self.direction = Settings.directions['down']
51         self.speed = 2
52
53     def update(self):
54         self.rect.move_ip([self.speed*x for x in self.direction]) # Liste multiplizieren
55         if self.rect.top > Settings.window['height'] - 30:
56             self.kill() # Selbstzerstörung
```

---

Im Konstruktor 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 75 richten. Durch den Aufruf von `pygame.time.Clock.tick()` wird das Spiel getaktet – hier auf das 1/60 einer Sekunde.

`tick()`

### Quelltext 2.51: Zeitsteuerung (4), Version 1.0: Konstruktor und `run()` von Game

---

```
60 class Game(object):
61     def __init__(self) -> None:
62         super().__init__()
63         os.environ['SDL_VIDEO_WINDOW_POS'] = "10, 50"
64         pygame.init()
65         pygame.display.set_caption(Settings.title)
66         self.screen = pygame.display.set_mode(Settings.dim())
67         self.clock = pygame.time.Clock()
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):
73         self.running = True
74         while self.running:
75             self.clock.tick(Settings.fps) # Taktung
76             self.watch_for_events()
```

```
77         self.update()
78         self.draw()
79     pygame.quit()
```

---

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

---

### Quelltext 2.52: Zeitsteuerung (5), Version 1.0: `watch_for_events()` und `draw()` von Game

---

```
81     def watch_for_events(self):
82         for event in pygame.event.get():
83             if event.type == pygame.QUIT:
84                 self.running = False
85             elif event.type == pygame.KEYDOWN:
86                 if event.key == pygame.K_ESCAPE:
87                     self.running = False
88
89     def draw(self):
90         self.screen.fill((200, 200, 200))
91         self.all_bullets.draw(self.screen)
92         self.enemy.draw(self.screen)
93     pygame.display.flip()
```

---

Die Methode `update()` ist nur bzgl. Zeile 75 erwähnenswert, da dort ein neuer Feuerball erzeugt/abgeworfen wird, indem die Methode `new_bullet()` aufgerufen wird. Dabei wird zunächst ein neuer Feuerball erzeugt und in `b` geparkt, da wir noch die Position festlegen wollen, soll doch der Feuerball nicht bei (0, 0) starten. Die Startposition ergibt sich aus der aktuellen Position des Enemys. Das horizontale Zentrum von Feuerball und Enemy soll gleich sein. Das vertikale Zentrum etwas nach unten verschoben; sieht besser aus. Erst danach wird der Feuerball der Spritegruppe hinzugefügt.

---

### Quelltext 2.53: Zeitsteuerung (6), Version 1.0: `update()` und `new_bullet()` von Game

---

```
92         self.enemy.draw(self.screen)
93     pygame.display.flip()
94
95     def update(self):
96         self.new_bullet()                                     # Feuerballabwurf
97         self.all_bullets.update()
98         self.enemy.update()
99
100    def new_bullet(self):
101        b = Bullet("shoot.png")
102        b.rect.centerx = self.enemy.sprite.rect.centerx
103        b.rect.centery = self.enemy.sprite.rect.centery + 20
104        self.all_bullets.add(b)
```

---

Zurück zum eigentlichen Problem. Wir haben oben festgestellt, dass durch `Settings.fps` und dem Aufruf von `tick()` in Zeile 75 die Anwendung auf das 1/60 einer Sekunde getaktet ist. Mit anderen Worten: Derzeit werden 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 2.54: Zeitsteuerung (7), Version 1.1: Konstruktor von `Game`

---

```
60  class Game(object):
61      def __init__(self) -> None:
62          super().__init__()
63          os.environ['SDL_VIDEO_WINDOW_POS'] = "10, 50"
64          pygame.init()
65          pygame.display.set_caption(Settings.title)
66          self.screen = pygame.display.set_mode(Settings.dim())
67          self.clock = pygame.time.Clock()
68          self.enemy = pygame.sprite.GroupSingle(Enemy("alienbig1.png"))
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 2.28 auf der nächsten Seite: Es sind nur noch zwei Feuerbälle sichtbar.

Quelltext 2.55: Zeitsteuerung (8), Version 1.1: `new_bullet()` von `Game`

---

```
102     def new_bullet(self):
103         self.time_counter += 1                         # Erhöhe pro Takt um 1
104         if self.time_counter >= self.time_range:       # Wenn Obergrenze erreicht
105             b = Bullet("shoot.png")
106             b.rect.centerx = self.enemy.sprite.rect.centerx
107             b.rect.centery = self.enemy.sprite.rect.centery + 20
108             self.all_bullets.add(b)
109             self.time_counter = 0                        # Setze Zähler wieder zurück
```

---

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.

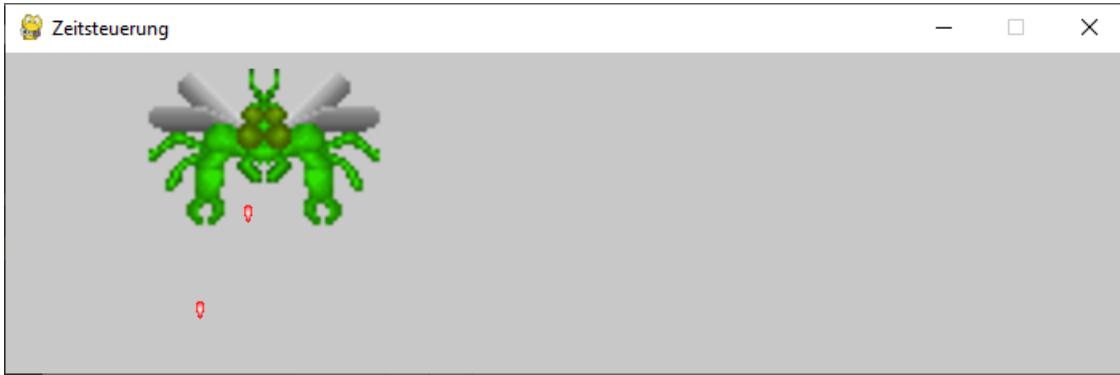


Abbildung 2.28: Feuerball mit Zeitsteuerung

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

`get_ticks()`

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

Quelltext 2.56: 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 103 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 103.

Quelltext 2.57: Zeitsteuerung (10), Version 1.2: `new_bullet()` von `Game`

---

```
102     def new_bullet(self):
103         if pygame.time.get_ticks() >= self.time_stamp + self.time_duration: # Wenn
104             # Intervallgrenze erreicht
105             b = Bullet("shoot.png")
106             b.rect.centerx = self.enemy.sprite.rect.centerx
107             b.rect.centery = self.enemy.sprite.rect.centery + 20
108             self.all_bullets.add(b)
109             self.time_stamp = pygame.time.get_ticks()                         # Neuer
110             # Intervallstart
```

---

Da wir diese Logik mehrfach brauchen, habe ich das ganze in der Klasse `Timer` gekapselt.

`Timer`

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

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

Quelltext 2.58: Zeitsteuerung (11), Version 1.3: Timer

---

```
26 class Timer(object):
27     def __init__(self, duration, with_start = True):
28         self.duration = duration
29         if with_start:
30             self.next = pygame.time.get_ticks()
31         else:
32             self.next = pygame.time.get_ticks() + self.duration
33
34     def is_next_stop_reached(self):
35         if pygame.time.get_ticks() > self.next:
36             self.next = pygame.time.get_ticks() + self.duration
37             return True
38         return False
```

---

Wie wird dieser Timer nun verwendet? Zunächst wird im Konstruktor ein entsprechendes Objekt erzeugt (Zeile 84); die beiden Variablen von eben werden nicht mehr gebraucht.

Quelltext 2.59: Zeitsteuerung (12), Version 1.3: Timer-Objekt erzeugen

---

```
83     self.all_bullets = pygame.sprite.Group()
84     self.bullet_timer = Timer(500)                                     # Timer ohne Verzögerung
85     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 116 abgefragt, ob das Intervallende erreicht wurde und fertig!

Quelltext 2.60: Zeitsteuerung (13), Version 1.3: Timer-Objekt verwenden

---

```
115     def new_bullet(self):
116         if self.bullet_timer.is_next_stop_reached():           # Wenn Intervallgrenze erreicht
117             b = Bullet("shoot.png")
118             b.rect.centerx = self.enemy.sprite.rect.centerx
119             b.rect.centery = self.enemy.sprite.rect.centery + 20
120             self.all_bullets.add(b)
```

---

## Was war neu?

- `pygame.sprite.Sprite.kill()`:  
<https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.Sprite.kill>
- `pygame.time.get_ticks()`:  
[https://www.pygame.org/docs/ref/time.html#pygame.time.get\\_ticks](https://www.pygame.org/docs/ref/time.html#pygame.time.get_ticks)

## 2.11 Animation

Eine Animation ist eigentlich eine Art *Filmchen* innerhalb eines Spiels. Beispiele für sinnvolle Animationen sind Bewegungen oder 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 2.29: Animation einer Katze: Einzelsprites

Die Einzelbilder des Bewegungsbeispiels können Sie in Abbildung 2.29 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 2.61 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 2.61: Animation einer Katze (1), Version 1.0: Präambel, Timer und Settings

---

```
1 import pygame
2 from pygame.constants import (QUIT, K_KP_PLUS, K_KP_MINUS, K_ESCAPE, KEYDOWN)
3 import os
4
5
6
7 class Settings(object):
8     window = {'width':300, 'height':200}
9     fps = 60
10    title = "Animation"
11    path = {}
12    path['file'] = os.path.dirname(os.path.abspath(__file__))
13    path['image'] = os.path.join(path['file'], "images")
14    directions = {'stop':(0, 0), 'down':(0, 1), 'up':(0, -1), 'left':(-1, 0),
15                  'right':(1, 0)}
16
17    @staticmethod
18    def dim():
19        return (Settings.window['width'], Settings.window['height'])
20
21    @staticmethod
22    def filepath(name):
23        return os.path.join(Settings.path['file'], name)
24
25    @staticmethod
26    def imagepath(name):
27        return os.path.join(Settings.path['image'], name)
28
29 class Timer(object):
30     def __init__(self, duration, with_start = True):
31         self.duration = duration
32         if with_start:
33             self.next = pygame.time.get_ticks()
34         else:
35             self.next = pygame.time.get_ticks() + self.duration
36
37     def is_next_stop_reached(self):
38         if pygame.time.get_ticks() > self.next:
39             self.next = pygame.time.get_ticks() + self.duration
40             return True
41         return False
42
43     def change_duration(self, delta=10):
44         self.duration += delta
45         if self.duration < 0:
46             self.duration = 0
```

---

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

In der Methode `update()` wird nun abhängig vom `Timer`-Objekt das Attribut `imageindex` immer um 1 erhöht und dieses Bitmap dann dem Attribut `image` zugewiesen, damit die schon bekannten `Sprite`-Features genutzt werden können. Die Methode `change_anima-`

`tion_time()` reicht seinen Übergabeparameter einfach nur an das `Timer`-Objekt weiter. Damit sind eigentlich alle vorbereitenden Aktivitäten abgeschlossen.

Quelltext 2.62: Animation einer Katze (2), Version 1.0: Cat

---

```
49  class Cat(pygame.sprite.Sprite):
50      def __init__(self):
51          super().__init__()
52          self.images = []
53          for i in range(6):                      # Animations-Sprites laden
54              bitmap = pygame.image.load(Settings.imagepath(f"cat{i}.bmp")).convert()
55              bitmap.set_colorkey((0,0,0))
56              self.images.append(bitmap)
57          self.imageindex = 0
58          self.image = self.images[self.imageindex]
59          self.rect = self.image.get_rect()
60          self.rect.center = (Settings.window['width'] // 2, Settings.window['height'] // 2)
61          self.animation_time = Timer(100)
62
63      def update(self):
64          if self.animation_time.is_next_stop_reached():
65              self.imageindex += 1
66              if self.imageindex >= len(self.images):
67                  self.imageindex = 0
68              self.image = self.images[self.imageindex]
69              # implement game logic here
70
71      def change_animation_time(self, delta):
72          self.animation_time.change_duration(delta)
```

---

Die Klasse `CarAnimation` ist nur die übliche Kapselung des Hauptprogramms. In Zeile 84 wird das `Cat`-Objekt erzeugt und in ein `GroupSingle` gestopft.

Quelltext 2.63: Animation einer Katze (3), Version 1.0: Konstruktor und `run()`

---

```
75  class CatAnimation(object):
76      def __init__(self) -> None:
77          super().__init__()
78          os.environ['SDL_VIDEO_WINDOW_POS'] = "10, 50"
79          pygame.init()
80          self.screen = pygame.display.set_mode(Settings.dim())
81          pygame.display.set_caption(Settings.title)
82          self.clock = pygame.time.Clock()
83          self.font = pygame.font.Font(pygame.font.get_default_font(), 12)
84          self.cat = pygame.sprite.GroupSingle(Cat()) # Meine Katze
85          self.running = False
86
87      def run(self) -> None:
88          self.running = True
89          while self.running:
90              self.clock.tick(Settings.fps)
91              self.watch_for_events()
92              self.update()
93              self.draw()
94          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 2.64: Animation einer Katze (4), Version 1.0: `watch_for_events()`

---

```
96     def watch_for_events(self) -> None:
97         for event in pygame.event.get():
98             if event.type == QUIT:
99                 self.running = False
100            elif event.type == KEYDOWN:
101                if event.key == K_ESCAPE:
102                    self.running = False
103                elif event.key == K_KP_PLUS:
104                    self.cat.sprite.change_animation_time(-10)
105                elif event.key == K_KP_MINUS:
106                    self.cat.sprite.change_animation_time(10)
```

---

Der restliche Quelltext (Quelltext 2.65) sollte selbsterklärend sein. Wenn Sie das Programm nun starten, sollte eine animierte Katzenbewegung zu sehen sein. Probieren Sie doch mal aus, die Animationsgeschwindigkeit zu verändern.

Quelltext 2.65: Animation einer Katze (5), Version 1.0: `update()` und `draw()`

---

```
108    def update(self) -> None:
109        self.cat.update()
110
111    def draw(self) -> None:
112        self.screen.fill((200, 200, 200))
113        self.cat.draw(self.screen)
114        text_image = self.font.render(f"animation time:
115            {self.cat.sprite.animation_time.duration}", True, (255, 255, 255))
116        text_rect = text_image.get_rect()
117        text_rect.centerx = Settings.window['width'] // 2
118        text_rect.bottom = Settings.window['height'] - 50
119        self.screen.blit(text_image, text_rect)
120        pygame.display.flip()
121
122
123 if __name__ == '__main__':
124     anim = CatAnimation()
125     anim.run()
```

---

Wie bei der Zeitsteuerung stört mich, dass die Animationslogik über die Klasse `Cat` verteilt ist, was meiner Ansicht nach ein Verstoß gegen das SRP ist. Bauen wir doch eine einfache Animationsklasse (siehe Quelltext 2.66 auf der nächsten Seite).

Schauen wir uns die Übergabeparameter 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 mach 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 14). Wird keine Angabe gemacht, bleibt die Transparenz des geladenen Sprites erhalten. Wird eine Farbangabe gemacht, wird diese mit `set_colorkey()` in Zeile 59 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 2.66: Animation (6), Version 1.1: Animation

---

```
49  class Animation(object):
50      def __init__(self, namelist, endless, animationtime, colorkey=None):
51          self.images = []
52          self.endless = endless
53          self.timer = Timer(animationtime)
54          for filename in namelist:
55              if colorkey == None:
56                  bitmap = pygame.image.load(Settings.imagepath(filename)).convert_alpha()
57              else:
58                  bitmap = pygame.image.load(Settings.imagepath(filename)).convert()
59                  bitmap.set_colorkey(colorkey)           # Transparenz herstellen
60              self.images.append(bitmap)
61          self.imageindex = -1
62
63      def next(self):
64          if self.timer.is_next_stop_reached():
65              self.imageindex += 1
66              if self.imageindex >= len(self.images):
67                  if self.endless:
68                      self.imageindex = 0
69                  else:
70                      self.imageindex = len(self.images) - 1
71          return self.images[self.imageindex]
72
73      def is-ended(self):
74          if self.endless:
75              return False
76          elif self.imageindex >= len(self.images) - 1:
77              return True
78          else:
79              return False
```

---

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 87. Die Dateinamen lassen sich schön einfach generieren, da

sie durchnummiert wurden. Die Katze solle endlos laufen und dabei 1 100 ms zeitlichen Abstand zwischen den Sprites haben. In `update()` wird dann einfach die Methode `next()` aufgerufen.

---

### Quelltext 2.67: Animation einer Katze (7), Version 1.1: Cat

---

```
84 class Cat(pygame.sprite.Sprite):
85     def __init__(self):
86         super().__init__()
87         self.animation=Animation([f"cat{i}.bmp" for i in range(6)], True, 100, (0,0,0)) #
88         self.image = self.animation.next()
89         self.rect = self.image.get_rect()
90         self.rect.center = (Settings.window['width'] // 2, Settings.window['height'] // 2)
91
92     def update(self):
93         self.image = self.animation.next()
```

---

## 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 `Felsen` an. In Zeile 87 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 Zufallszahlengenerator geraten, wobei ein gewisser Abstand zu den Rändern gewahrt wird. In Zeile 92 wird das `Animation`-Objekt erzeugt. Dabei werden die Dateinamen der Animationsbitmaps wieder in der Reihenfolge der Animation eingelesen. Die Bitmaps können Sie in Abbildung 2.30 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 93 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 bin<sup>1</sup>.

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 101 auf sich? Sie dienen rein optischen Zwecken. Die Abmaße der Explosionssprites sind nicht immer gleich und werden durch das `rect`-Objekt immer auf die linke, obere Koordinate ausgerichtet, was zu einem Ruckeln führen würde. So merke ich mir

---

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

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 2.68: Animation einer Explosion (1): Rock

---

```
84  class Rock(pygame.sprite.Sprite):
85      def __init__(self):
86          super().__init__()
87          rocknb = random.randint(6,9)                                     # Felsennummer
88          self.image =
89              pygame.image.load(Settings.imagepath(f"felsen{rocknb}.png")).convert_alpha()
90          self.rect = self.image.get_rect()
91          self.rect.centerx = random.randint(self.rect.width,
92                                              Settings.window['width']-self.rect.width)
93          self.rect.centery = random.randint(self.rect.height,
94                                              Settings.window['height']-self.rect.height)
95          self.anim = Animation([f"explosion0{i}.png" for i in range(1, 5)], False, 50) #
96          self.timer_lifetime = Timer(random.randint(100, 2000), False)    # Lebenszeit
97          self.bumm = False
98
99      def update(self):
100         if self.timer_lifetime.is_next_stop_reached():
101             self.bumm = True
102         if self.bumm:
103             self.image = self.anim.next()                                     # Zentrum
104             c = self.rect.center
105             self.rect = self.image.get_rect()
106             self.rect.center = c
107         if self.anim.isEnded():
108             self.kill()
```

---



Abbildung 2.30: 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 118 wird ein `Timer`-Objekt angelegt, welches zwei Felsen pro Sekunde erstellen soll und in Zeile 139 wird dieser abgefragt.

Quelltext 2.69: Animation einer Explosion (2): ExplosionAnimation

---

```
109  class ExplosionAnimation(object):
110      def __init__(self) -> None:
111          super().__init__()
112          os.environ['SDL_VIDEO_WINDOW_POS'] = "10, 50"
113          pygame.init()
114          self.screen = pygame.display.set_mode(Settings.dim())
115          pygame.display.set_caption(Settings.title)
116          self.clock = pygame.time.Clock()
117          self.all_rocks = pygame.sprite.Group()
118          self.timer_newrock = Timer(500)                                     # Timer
```

```
119         self.running = False
120
121     def run(self) -> None:
122         self.running = True
123         while self.running:
124             self.clock.tick(Settings.fps)
125             self.watch_for_events()
126             self.update()
127             self.draw()
128             pygame.quit()
129
130     def watch_for_events(self) -> None:
131         for event in pygame.event.get():
132             if event.type == QUIT:
133                 self.running = False
134             elif event.type == KEYDOWN:
135                 if event.key == K_ESCAPE:
136                     self.running = False
137
138     def update(self) -> None:
139         if self.timer_newrock.is_next_stop_reached():          # 500ms?
140             self.all_rocks.add(Rock())
141             self.all_rocks.update()
142
143     def draw(self) -> None:
144         self.screen.fill((0, 0, 0))
145         self.all_rocks.draw(self.screen)
146         pygame.display.flip()
```

---

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

### Was war neu?

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

## 2.12 Maus

Wie wohl viele Spiele durch Tastatur oder Controller gesteuert werden, wird auch oft die Maus verwendet. Hier 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.

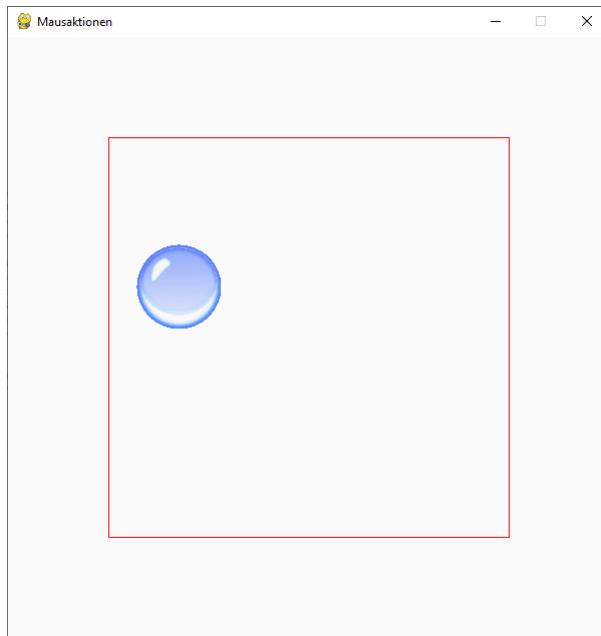


Abbildung 2.31: Mausaktionen

- Ein Klick mit dem Mausrad beendet die Anwendung.

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 71 wird das `Ball`-Objekt erzeugt.

---

### Quelltext 2.70: Mausaktionen: Statics und Konstruktor von `Game`

---

```
53 class Game:  
54     window: Dict[str, int] = {"width": 600, "height": 600}  
55     path: Dict[str, str] = {}  
56     path["file"] = os.path.dirname(os.path.abspath(__file__))  
57     path["image"] = os.path.join(path["file"], "images")  
58     inner_rect = pygame.Rect(100, 100, window["width"] - 200, window["height"] - 200)  
59  
60     @staticmethod  
61     def get_dim() -> Tuple[int, int]:  
62         return (Game.window["width"], Game.window["height"])  
63  
64     def __init__(self) -> None:  
65         os.environ["SDL_VIDEO_WINDOW_POS"] = "650, 70"  
66         pygame.init()  
67         self._fps = 60  
68         self._clock = pygame.time.Clock()  
69         self._screen = pygame.display.set_mode(Game.get_dim())  
70         pygame.display.set_caption("Mausaktionen")  
71         self._ball = Ball()                                     # Ball-Objekt  
72         self._running = True
```

---

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

---

### Quelltext 2.71: Mausaktionen: Game.run()

---

```
74     def run(self) -> None:
75         while self._running:
76             self._clock.tick(self._fps)
77             self.watch_for_events()
78             self.update()
79             self.draw()
80         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 89 wird der `event.type` abgefragt und anschließend wird ermittelt, welche Maustaste denn gedrückt wurde.

MOUSE-  
BUTTON-  
DOWN  
MOUSE-  
BUTTONUP

Dazu liefern mir diese beiden Mausevents zwei Attribute: `event.button` und `event.pos`. In Tabelle 2.3 auf Seite 77 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.

event.button

Wird also die linke Maustaste gedrückt (Zeile 90), 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 94). Das Mausrad wird ebenfalls wie ein Mausbutton verarbeitet. Je nach Drehrichtung wird dabei ein anderer Zahlenkode zurückgeliefert (siehe Zeile 96 und Zeile 98). Wird das Mausrad gedrückt – also geklickt – soll ja das Spiel beendet werden. In Zeile 92 wird dies abgefragt und umgesetzt.

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

event.pos

---

### Quelltext 2.72: Mausaktionen: Game.watch\_for\_events()

---

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

---

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 105

wird dies durch die Methode `pygame.mouse.set_visible()` erreicht. Diese steuert, ob der Systemmauszeiger – welcher Ausprägung auch immer – angezeigt werden soll oder nicht.

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

`set_visible()`

`get_pos()`

`collidepoint()`

### Quelltext 2.73: Mausaktionen: `Game.update()` und `Game.draw()`

---

```
101     def update(self):
102         self._ball.update(center=pygame.mouse.get_pos())
103         pygame.mouse.set_visible(
104             not Game.inner_rect.collidepoint(pygame.mouse.get_pos()))
105         )                                         # Unsichtbar?
106         self._ball.update(go=True)
107
108     def draw(self) -> None:
109         self._screen.fill((250, 250, 250))
110         pygame.draw.rect(self._screen, (255, 0, 0), Game.inner_rect, 1)
111         self._ball.draw(self._screen)
112         pygame.display.flip()
```

---

Verbleibt noch die Klasse `Ball`. Diese enthält zwar keine direkten Mausaktionen mehr, aber die Methode `update()` sieht nun ganz anders als bei den vorherigen Beispielen aus. In früheren Beispielen wurden Methoden wie `rotate()` oder `resize()` direkt aus `watch_for_events()` oder vergleichbaren Methoden von `Game` aufgerufen. Das ist auch soweit in Ordnung, aber wenn man diese Kindklassen von `pygame.sprite.Sprite` einer `pygame.sprite.Group` oder `pygame.sprite.GroupSingle` hinzugefügt 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 2.19 auf Seite 28. 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 18, Zeile 29, Zeile 32 oder Zeile 35. 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 2.74: Mausaktionen: Ball

---

```
8  class Ball(pygame.sprite.Sprite):
9      def __init__(self) -> None:
10         super().__init__()
11         fullfilename = os.path.join(Game.path["image"], "blue2.png")
12         self.image_orig: pygame.surface.Surface =
13             pygame.image.load(fullfilename).convert_alpha()
13         self._scale = 10
14         self.image: pygame.surface.Surface = pygame.transform.scale(self.image_orig,
15             (self._scale, self._scale))
15         self.rect: pygame.rect.Rect = self.image.get_rect()
16
17     def update(self, *args: Any, **kwargs: Any) -> None:
18         if "go" in kwargs.keys():                                     # Parameter vorhanden?
19             if kwargs["go"]:
20                 self.rect.left = max(self.rect.left, Game.inner_rect.left)
21                 self.rect.right = min(self.rect.right, Game.inner_rect.right)
22                 self.rect.top = max(self.rect.top, Game.inner_rect.top)
23                 self.rect.bottom = min(self.rect.bottom, Game.inner_rect.bottom)
24                 c = self.rect.center # remember old center
25                 self.image = pygame.transform.scale(self.image_orig, (self._scale,
26                     self._scale))
26                 self.rect = self.image.get_rect()
27                 self.rect.center = c # set center to old position
28
29         if "rotate" in kwargs.keys():                                     #
30             self.rotate(kwargs["rotate"])
31
32         if "scale" in kwargs.keys():                                     #
33             self.resize(kwargs["scale"])
34
35         if "center" in kwargs.keys():                                     #
36             self.set_center(kwargs["center"])
37
38     def draw(self, screen: pygame.surface.Surface) -> None:
39         screen.blit(self.image, self.rect)
40
41     def rotate(self, angle: float) -> None:
42         self.image_orig = pygame.transform.rotate(self.image_orig, angle)
43
44     def resize(self, delta: int) -> None:
45         if self.rect.width < Game.inner_rect.width:
46             if self._scale + delta > 0:
47                 self._scale += delta
48
49     def set_center(self, center: Tuple[int, int]) -> None:
50         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?

- `pygame.Rect.collidepoint()`:  
<https://www.pygame.org/docs/ref/rect.html#pygame.Rect.collidepoint>
- `pygame.transform.rotate()`:  
<https://www.pygame.org/docs/ref/transform.html#pygame.transform.rotate>
- `pygame.mouse.get_pos()`:  
[https://www.pygame.org/docs/ref/mouse.html#pygame.mouse.get\\_pos](https://www.pygame.org/docs/ref/mouse.html#pygame.mouse.get_pos)
- `pygame.mouse.set_visible()`:  
[https://www.pygame.org/docs/ref/mouse.html#pygame.mouse.set\\_visible](https://www.pygame.org/docs/ref/mouse.html#pygame.mouse.set_visible)
- `pygame.MOUSEBUTTONDOWN`, `pygame.MOUSEBUTTONUP`:  
<https://www.pygame.org/docs/ref/event.html>
- Liste der Mausbuttons: Tabelle 2.3

Tabelle 2.3: 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)

# 3 Beispielprojekte

## 3.1 Bubbles

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

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

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

### 3.1.1 Requirement 1: Standards

---

#### Requirement 1 Standardfunktionalität

---

1. Fenster hat eine angemessene Größe.
  2. Hintergrund ist eine passende Bitmap oder einfarbig.
  3. Beendet wird mit der ESC-Taste oder per Mausklick auf rote „X“.
  4. Alle Bitmaps werden als `pygame.sprite.Sprite`-Objekt erzeugt und nach dem Laden konvertiert und passend skaliert.
  5. Alle Bitmaps – außer dem Hintergrund – sind transparent.
  6. Alle Bitmaps – bis auf „Einzelkämpfer“ oder Hintergrund – werden in `pygame.sprite.Group`-Objekten abgelegt.
  7. Das Spiel hat eine Ablaufgeschwindigkeit von 60 fps.
- 

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.

#### Quelltext 3.1: Bubbles (Requirement 1.1) – Präambel und Settings

---

```
1 import os
2 from typing import Tuple
3
4 import pygame
5 from pygame.constants import K_ESCAPE, KEYDOWN, QUIT
6
7
8 class Settings:
9     window = {"width": 1220, "height": 1002}
10    fps = 60
11    path = {}
12    path["file"] = os.path.dirname(os.path.abspath(__file__))
13    path["image"] = os.path.join(path["file"], "images")
14    path["sound"] = os.path.join(path["file"], "sounds")
15    caption = 'Fingerübung "Bubbles"'
16
17    @staticmethod
18    def get_dim() -> Tuple[int, int]:
19        return (Settings.window["width"], Settings.window["height"])
```

---

Es wird gefordert, dass das Fenster eine angemessene Größe hat. Mit 1220 *px*x1002 *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` oder `path`) und wird deshalb hier nicht weiter erläutert.

#### Quelltext 3.2: Bubbles (Requirement 1.2) – Background

---

```
22 class Background(pygame.sprite.Sprite):
23     def __init__(self, filename: str = "aquarium.png") -> None:
24         super().__init__()
25         self.image = pygame.image.load(os.path.join(Settings.path["image"],
26                                         filename)).convert()
27         self.image = pygame.transform.scale(self.image, Settings.get_dim())
28         self.rect = self.image.get_rect()
```

---

Die Klasse `Background` ist ein einzelnes Sprite, welches nur geladen und passend skaliert wird. Da sich der Hintergrund nicht ändert, muss kein `update()` implementiert werden. Auch wird keine Transparenz benötigt. Dass wir eine eigene Kindklasse programmieren, ist ein wenig wie mit Pistolen auf Spatzen schießen. Wir hätten es auch als `Sprite`-Objekt implementieren können. Ich habe das hier nur der Übersichtlichkeit wegen gemacht. Das Hintergrundbild ist in Abbildung 3.1 auf der nächsten Seite zu sehen.

#### Quelltext 3.3: Bubbles (Requirement 1) – Game

---

```
30 class Game:
31     def __init__(self) -> None:
32         pygame.init()
33         self._screen = pygame.display.set_mode(Settings.get_dim())
34         pygame.display.set_caption(Settings.caption)
35         self._clock = pygame.time.Clock()
36         self._background = pygame.sprite.GroupSingle(Background())
37         self._running = True
38
39     def watch_for_events(self) -> None:
```

```
40     for event in pygame.event.get():
41         if event.type == QUIT:
42             self._running = False
43         elif event.type == KEYDOWN:
44             if event.key == K_ESCAPE:
45                 self._running = False
46
47     def draw(self) -> None:
48         self._background.draw(self._screen)
49         pygame.display.flip()
50
51     def update(self) -> None:
52         pass
53
54     def run(self) -> None:
55         self._running = True
56         while self._running:
57             self._clock.tick(Settings.fps)
58             self.watch_for_events()
59             self.update()
60             self.draw()
61
62     pygame.quit()
```

---

In der Klasse `Game` werden in `__init__()` die Pygame üblichen Methoden `init()`, `set_mode()`, `clock()` und `set_caption()` am Start aufgerufen. Auch wird der Hintergrund in ein `pygame.sprite.GroupSingle`-Objekt abgelegt und das Flag der Hauptprogrammschleife erzeugt. Die Methoden `run()`, `watch_for_events()`, `update()` und `draw()` enthalten nur Basisfunktionalitäten, die hier nicht weiter erläutert werden müssen. Durch diese Methoden ist aber schon der generelle Ablauf des Spiels vorgegeben. Alle weiteren Eigenschaften des Spiels, sind nur noch Erweiterungen dieses Ablaufs, keine Veränderungen mehr.

Zum Schluss erfolgt nur noch der Aufruf (siehe Quelltext 3.4 auf der nächsten Seite).



Abbildung 3.1: Bubbles: Hintergrundbild (aquarium.png)

#### Quelltext 3.4: Bubbles (Requirement 1) – Aufruf

---

```
65 if __name__ == "__main__":
66     os.environ["SDL_VIDEO_WINDOW_POS"] = "10, 30"
67     game = Game()
68     game.run()
```

---

Damit sind alle Unterpunkte von Requirement 1 auf Seite 78, die hier Anwendung finden, erfüllt.

### 3.1.2 Requirement 2: Blasen erscheinen

#### Requirement 2 Blasen erscheinen

---

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

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



Abb. 3.2: Blase

Die Spielfläche ist ein Rechteck mit einem Abstand zum linken und oberen Bildschirmrand – `left` und `top` – und einer Breite (`width`) und Höhe (`height`). In Zeile 19 werden die entsprechenden Werte festgehalten. Der Abstand von Spielfeldrand und der Blasen untereinander wird in Zeile 18 entsprechend Requirement 2.4 mit 10 *px* definiert. Der Startradius – und damit der minimale Radius – wird wegen Requirement 2.3 in Zeile 17 mit 15 *px* festgelegt. Während des Spielens ist mir aufgefallen, dass kleinere Startradien einfach zu schlecht gesehen werden.

#### Quelltext 3.5: Bubbles (Requirement 2) – Ergänzungen in Settings

---

```
17     radius = {"min": 15}                                # Radius Startwert
18     distance = 50                                     # Rand-/Blasenabstand
19     playground = pygame.Rect(90, 90, 1055, 615)       # Rechteck im Aquarium
```

---

Die Klasse `Timer` ist exakt die oben in Kapitel 2.10 auf Seite 58 beschriebene; dort wird alles erklärt.

#### Quelltext 3.6: Bubbles (Requirement 2) – Timer

```
26 class Timer:
27     def __init__(self, duration: int, with_start: bool = True) -> None:
28         self.duration = duration
29         if with_start:
30             self._next = pygame.time.get_ticks()
31         else:
32             self._next = pygame.time.get_ticks() + self.duration
33
34     def is_next_stop_reached(self) -> bool:
35         if pygame.time.get_ticks() > self._next:
36             self._next = pygame.time.get_ticks() + self.duration
37         return True
38     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 noch 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 3.7: Bubbles (Requirement 2) – Bubble

---

```
49 class Bubble(pygame.sprite.Sprite):
50     def __init__(self, filename: str = "blase1.png") -> None:
51         super().__init__()
52         self.radius = Settings.radius["min"]
53         self.image = pygame.image.load(os.path.join(Settings.path["image"],
54                                         filename)).convert_alpha()
55         self.image = pygame.transform.scale(self.image, (Settings.radius["min"],
56                                         Settings.radius["min"]))
57         self.rect = self.image.get_rect()
58
59     def update(self) -> None:
60         pass
61
62     def randompos(self) -> None:
63         bubbledistance = Settings.distance + Settings.radius["min"]
64         centerx = randint(Settings.playground.left + bubbledistance,
65                           Settings.playground.right - bubbledistance)
66         centery = randint(Settings.playground.top + bubbledistance,
67                           Settings.playground.bottom - bubbledistance)
68         self.rect.center = (centerx, centery)
```

---

In der Zeile 73 wird für den Background ein `GroupSingle`-Objekt angelegt; das erspart mir ein wenig Tipparbeit, da ich `update()` und `draw()` über die üblichen `SpriteGroup`-Mechanismen aufrufen kann. Zeile 74 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 3.8: Bubbles (Requirement 2) – Konstruktor von Game

---

```
67 class Game:
68     def __init__(self) -> None:
69         pygame.init()
70         self._screen = pygame.display.set_mode(Settings.get_dim())
```

```
71     pygame.display.set_caption(Settings.caption)
72     self._clock = pygame.time.Clock()
73     self._background = pygame.sprite.GroupSingle(Background()) # Timer 500ms
74     self._timer_bubble = Timer(500, False)                      # Timer 500ms
75     self._all_bubbles = pygame.sprite.Group()                   # Alle Blasen
76     self._running = True
```

---

In der Methode `draw()` werden lediglich die `draw()`-Methoden der Spritegruppen aufgerufen. Die Reihenfolge ist wichtig, damit der Hintergrund nicht die Bubbles übermalt ;-)  
Auch `update()` wurde angepasst, es ruft jetzt die Methode `spawn_bubble()` auf und delegiert damit die Aufgabe, neue Blasen zu erzeugen.

#### Quelltext 3.9: Bubbles (Requirement 2) – `draw()` und `update()` von Game

---

```
86     def draw(self) -> None:
87         self._background.draw(self._screen)
88         self._all_bubbles.draw(self._screen)
89         pygame.display.flip()
90
91     def update(self) -> None:
92         self.spawn_bubble()
```

---

Die Grundidee hinter `spawn_bubble()` ist, solange eine Position für eine neue Blase zu raten, bis man eine genügend 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 100) und nach der Kollisionsprüfung wieder auf seinen Ursprungswert reduziert (Zeile 102).

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.

sprite-  
collide()  
collide\_-  
circle()

#### Quelltext 3.10: Bubbles (Requirement 2) – `spawn_bubble()` von Game

---

```
94     def spawn_bubble(self) -> None:
95         if self._timer_bubble.is_next_stop_reached():
96             b = Bubble()
97             tries = 100
98             while tries > 0:
99                 b.randompos()
100                b.radius += Settings.distance           # Abstand zu Blasen
101                collided = pygame.sprite.spritecollide(b, self._all_bubbles, False,
102                                              pygame.sprite.collide_circle())
103                b.radius -= Settings.distance          # Alter Radius!
104                if collided:
105                    tries -= 1
106                else:
107                    self._all_bubbles.add(b)
108                    break
```

---

Das Ergebnis können Sie in Abbildung 3.3 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 3.3: Bubbles: Die Blasen haben beim Start einen Mindestabstand

#### 3.1.3 Requirement 3: Blasenanzahl

---

##### Requirement 3 Blasenanzahl

---

*Die maximale Anzahl der Blasen soll von der Spielfeldgröße abhängen.*

---

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

---

Quelltext 3.11: Bubbles (Requirement 3) – Ergänzung von `Settings`

---

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

---

Diese Obergrenze aus Zeile 20 wird in Zeile 97 abgefragt. Nur wenn die maximale Anzahl noch nicht erreicht wurde, wird eine neue Blase erzeugt.

---

Quelltext 3.12: Bubbles (Requirement 3) – Ergänzung von `Game` in `spawn_bubbles()`

---

```
95     def spawn_bubble(self) -> None:
96         if self._timer_bubble.is_next_stop_reached():
97             if len(self._all_bubbles) <= Settings.max_bubbles: # Platz?
98                 b = Bubble()
99                 tries = 100
100                while tries > 0:
101                    b.randompos()
102                    b.radius += Settings.distance
103                    collided = pygame.sprite.spritecollide(b, self._all_bubbles, False,
104                                                pygame.sprite.collide_circle)
105                    b.radius -= Settings.distance
106                    if collided:
107                        tries -= 1
108                    else:
109                        self._all_bubbles.add(b)
                                break
```

---

Der Rest des Programmes bleibt unverändert.

### 3.1.4 Requirement 4: Blasenwachstum

---

#### Requirement 4 Blasenwachstum

---

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

Der Sinn von Requirement 4.1 ist das Einsparen von Rechenzeit. Im Spiel werden immer wieder Blasen mit einem bestimmten Radius starten und dann wachsen. Jedes mal das Bitmap auf die passende Größe zu skalieren, würde Rechenzeit verschwenden – schließlich wird die gleiche Blase ja mit den Radien mehrfach vorkommen. Aus diesem Grund ist es sinnvoll, einmal die Blase in alle möglichen Radien zu skalieren und das Ergebnis in einem Dictionary abzulegen. Der Key ist dabei der jeweilige Radius (siehe Zeile 56). scale()

Die Methode `get()` liefert mir dann zu einem Radius das passend skalierte und schon fertige Image. Vorab wird in den Zeilen 59 und 60 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 3.13: Bubbles (Requirement 4.1) – BubbleContainer

---

```
50 class BubbleContainer:  
51     def __init__(self) -> None:  
52         image = pygame.image.load(os.path.join(Settings.path["image"],  
53             "blase1.png")).convert_alpha()  
54         self._images = {  
55             i: pygame.transform.scale(image, (i * 2, i * 2))  
56             for i in range(Settings.radius["min"], Settings.radius["max"] + 1)  
57         }  
58     def get(self, radius: int) -> pygame.surface.Surface:  
59         radius = max(Settings.radius["min"], radius)           # Untere Grenze  
60         radius = min(Settings.radius["max"], radius)           # Obere Grenze  
61         return self._images[radius]
```

---

Bisher wurde nur ein Startwert und damit eine untere Grenze für den Blasenradius in `Settings` definiert. Diese Definition wird nun in Zeile 17 passend zu Requirement 4.2 um die Angabe eines maximalen Radius erweitert.

Quelltext 3.14: Bubbles (Requirement 4.2) – Erweiterung von Settings

---

```
17     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 69 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 75). Der neue Radius wird in Zeile 73 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 74 und 77 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 3.15: Bubbles (Requirement 4) – Ergänzung von `Bubble`

---

```
64  class Bubble(pygame.sprite.Sprite):
65      def __init__(self, bubble_container: BubbleContainer, filename: str = "blase1.png"):
66          super().__init__()
67          self._bubble_container = bubble_container           # Verweis auf Container
68          self.radius = Settings.radius["min"]
69          self.image = self._bubble_container.get(self.radius)    # Zugriff auf Bubbles
70          self.rect = self.image.get_rect()
71
72      def update(self) -> None:
73          self.radius = min(self.radius + 1, Settings.radius["max"]) # Neuer Radius
74          center = self.rect.center                                # Alter Kreismittelpunkt
75          self.image = self._bubble_container.get(self.radius)    # Neues größeres Image
76          self.rect = self.image.get_rect()
77          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 112).

---

#### Quelltext 3.16: Bubbles (Requirement 4) – Ergänzung von `update()` in `Game`

---

```
111     def update(self) -> None:
112         self._all_bubbles.update()                         # Bubbles aktualisieren
113         self.spawn_bubble()
```

---

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

---

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

---

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

---

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



Abbildung 3.4: Bubbles: Die Blasen sind gewachsen/verwachsen

### 3.1.5 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 3.5 liefert mir aber einen einfachen Ansatz, das Problem zu lösen.

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

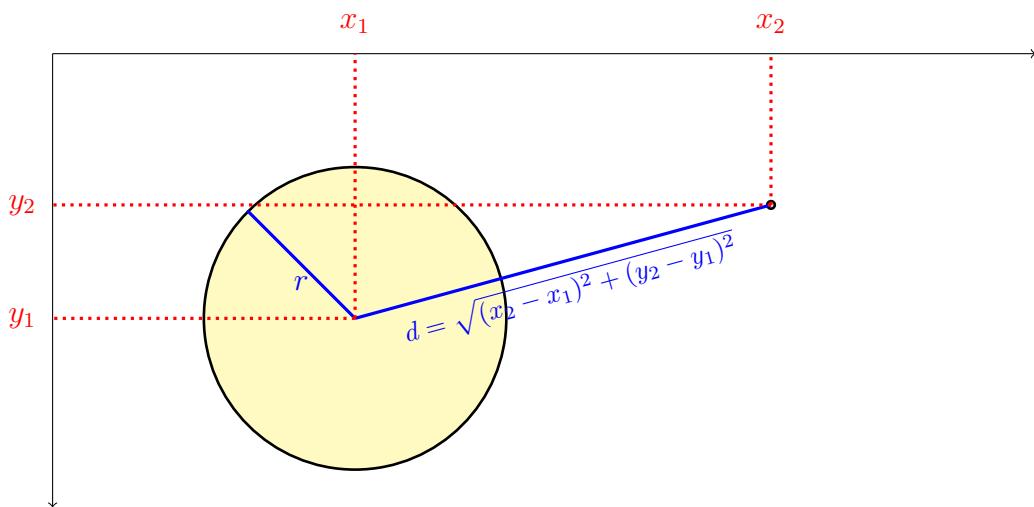


Abbildung 3.5: Kollisionserkennung: Punkt innerhalb des Kreises ([Satz des Pythagoras](#))?

#### Quelltext 3.18: Bubbles (Requirement 5) – collidepoint() in Bubble

---

```
86     def collidepoint(self, point: Tuple[int, int]) -> bool:
87         deltax = point[0] - self.rect.centerx
88         deltay = point[1] - self.rect.centery
89         return sqrt(deltax * deltax + deltay * deltay) <= self.radius
```

---

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 142 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 3.19: Bubbles (Requirement 5) – set\_mousecursor() in Game

---

```
138     def set_mousecursor(self) -> None:
139         is_over = False
140         pos = pygame.mouse.get_pos()
141         for b in self._all_bubbles:
142             if b.collidepoint(pos):                                     # Innerhalb?
143                 is_over = True
144                 break
145         if is_over:
146             pygame.mouse.set_cursor(pygame.SYSTEM_CURSOR_HAND)
147         else:
148             pygame.mouse.set_cursor(pygame.SYSTEM_CURSOR_CROSSHAIR)
```

---

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

### 3.1.6 Requirement 6: Blasen zerplatzen

---

#### Requirement 6 Blasen zerplatzen

---

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

---

Für die Umsetzung dieser Anforderung ist schon mit der Implementierung der Methode `Bubble.collidepoint()` fast alles erledigt. Wir müssen diese Methode nur geschickt einsetzen – es sind in der Tat nur wenige Restarbeiten nötig. In `watch_for_events()` wird zunächst der linke Mausklick abgefangen (Zeile 111) und die aktuelle Mausposition an die – neu erstellte Methode – `sting()` übergeben (Zeile 113).

MOUSE-BUTTONUP  
get\_pos()

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

#### Quelltext 3.20: Bubbles (Requirement 6) – `watch_for_event()` in Game

```
104     def watch_for_events(self) -> None:
105         for event in pygame.event.get():
106             if event.type == QUIT:
107                 self._running = False
108             elif event.type == KEYDOWN:
109                 if event.key == K_ESCAPE:
110                     self._running = False
111             elif event.type == pygame.MOUSEBUTTONUP:           # Mausklick?
112                 if event.button == 1:                         # left
113                     self.sting(pygame.mouse.get_pos())          # Aufruf
```

---

Die Methode `sting()` ist nun denkbar simpel. Es werden alle Bubble-Objekte durchwandert und dahingehend abgefragt, ob die Mausposition innerhalb des Radius liegt (Zeile 155). Wenn *Ja*, dann wird das entsprechende Objekt aus der Spritegroup mit `kill()` entfernt.

`kill()`

#### Quelltext 3.21: Bubbles (Requirement 6) – `sting()` in Game

```
153     def sting(self, mousepos) -> None:
154         for bubble in self._all_bubbles:
155             if bubble.collidepoint(mousepos):            # Innerhalb?
156                 bubble.kill()
```

---

### 3.1.7 Requirement 7: Punktestand

#### Requirement 7 Punktestand

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

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

#### Quelltext 3.22: Bubbles (Requirement 7.1) – Erweiterung von `Settings`

```
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 95 wird einfach der Radius der Blase auf den Punktestand addiert.

#### Quelltext 3.23: Bubbles (Requirement 7.2) – `stung()` in `Bubble`

---

```
93     def stung(self):
94         self.kill()
95         Settings.points += self.radius  
                                # Increment points
```

---

Die Methode `stung()` in `Game` muss dazu passend verändert werden (Zeile 177).

#### Quelltext 3.24: Bubbles (Requirement 7.2) – `sting()` in `Game`

---

```
174     def sting(self, mousepos) -> None:
175         for bubble in self._all_bubbles:
176             if bubble.collidepoint(mousepos):
177                 bubble.stung()  
                                # Nach Bubble verschoben
```

---

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

Rect

#### Quelltext 3.25: Bubbles (Requirement 7.3) – Erweiterung von `Settings`

---

```
22     box = pygame.Rect(90, 770, 1055, 1300)  
                                # 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 3.26: Bubbles (Requirement 7.3) – `Points`

---

```
98 class Points(pygame.sprite.Sprite):
99     def __init__(self) -> None:
100         super().__init__()
101         self._font = pygame.font.Font(pygame.font.get_default_font(), 18)
102
103     def update(self) -> None:
104         self.image = self._font.render(f"Points: {Settings.points}", True, (255, 0, 0))
105         self.rect = self.image.get_rect()
106         self.rect.left = Settings.box.left
107         self.rect.top = Settings.box.top
```

---

Verbleiben einige Erweiterungen in `Game`. Im Konstruktor wird das `Points`-Objekt in einer `GroupSingle` verwaltet.

#### Quelltext 3.27: Bubbles (Requirement 7.3) – Erweiterung des Konstruktors von `Game`

---

```
117     self._background = pygame.sprite.GroupSingle(Background())
118     self._points = pygame.sprite.GroupSingle(Points())  
                                # Points
119     self._timer_bubble = Timer(500, False)
```

---

Und in `draw()` und `update()` werden die entsprechenden Aufrufe eingefügt: siehe Zeile 137 und Zeile 142.

Quelltext 3.28: Bubbles (Requirement 7.3) – Erweiterung von `update()`, `draw()` in Game

---

```
134     def draw(self) -> None:
135         self._background.draw(self._screen)
136         self._all_bubbles.draw(self._screen)
137         self._points.draw(self._screen)                      # Punkte ausgeben
138         pygame.display.flip()
139
140     def update(self) -> None:
141         self._all_bubbles.update()
142         self._points.update()                                # Punkte aktualisieren
143         self.spawn_bubble()
144         self.set_mousecursor()
```

---

In Abbildung 3.6 können Sie die Punkteausgabe in der unteren Hälfte sehen. Diese Fläche könnte man später auch noch für eine Liste der besten zehn Punktestände oder andere Ausgaben verwenden.



Abbildung 3.6: Bubbles: Ausgabe Punktestand

### 3.1.8 Requirement 8: Spielende

---

#### Requirement 8 Spielende

---

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

Die Grundstruktur unseres Spiels ermöglicht es, diese Anforderung recht leicht durch eine Erweiterung von `update()` in Game zu realisieren. 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 144) 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.

Quelltext 3.29: Bubbles (Requirement 8) – Erweiterung von `update()` in Game

---

```
143     def update(self) -> None:
144         if self.check_bubblecollision():                      # Spielende?
145             self._running = False
146         else:
```

```

147         self._all_bubbles.update()
148         self._points.update()
149         self.spawn_bubble()
150         self.set_mousecursor()

```

Die beiden verschachtelten `for`-Schleifen ab Zeile 186 durchwandern die Gruppe der Blasen zweimal und vermeiden dabei zwei Dinge:

`sprites()`

- Eine Blase darf sich nicht mit sich selbst vergleichen: Daher beginnt der Index der inneren Schleife immer um eins versetzt zum aktuellen Index der äußeren Schleife, und der äußere Schleifenindex endet vor dem letzten Element der Blasengruppe.
- Wenn Blase 1 schon mit Blase 2 verglichen wurde, sollte Blase 2 nicht nochmal mit Blase 1 verglichen werden: Auch dies wird durch den versetzten Index erreicht.

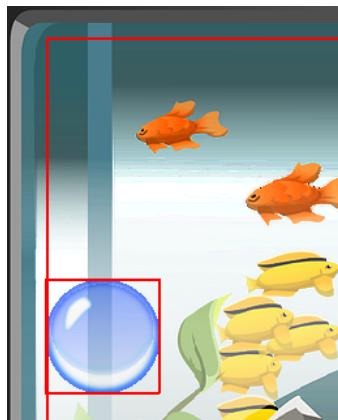


Abbildung 3.7: Bubbles – Kollision mit dem Rand

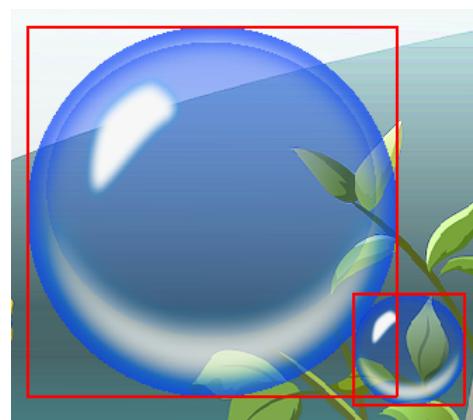


Abbildung 3.8: Bubbles – Kollision der Blasen

In Zeile 190 wird Requirement 8.1 überprüft. Dabei wird die auf der Kreisform basierende Kollisionsprüfung mit `collide_circle()` verwendet. In der Zeile 192 und Zeile 194 wird Requirement 8.2 umgesetzt. Dabei wird ausgenutzt, dass die Spielfläche ein Rechteck ist und das Sprite ebenfalls ein Rechteck besitzt. Die Methode `pygame.Rect.contains()` überprüft dabei, ob ein Rechteck innerhalb eines anderen liegt. Ist dies nicht der Fall – also verlässt die Blase die Spielfläche –, liegt eine Kollision vor.

`contains()`

#### Quelltext 3.30: Bubbles (Requirement 8) – `check_bubblecollision()` in Game

```

185     def check_bubblecollision(self) -> bool:
186         for index1 in range(0, len(self._all_bubbles) - 1):      # Bubbles prüfen
187             for index2 in range(index1 + 1, len(self._all_bubbles)):
188                 bubble1 = self._all_bubbles.sprites()[index1]
189                 bubble2 = self._all_bubbles.sprites()[index2]
190                 if pygame.sprite.collide_circle(bubble1, bubble2):    # Blasen kollidieren
191                     return True
192                 if not Settings.playground.contains(bubble1):        # Blase1 berührt Rand
193                     return True
194                 if not Settings.playground.contains(bubble2):        # Blase2 berührt Rand
195                     return True
196         return False

```

In Abbildung 3.7 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 3.8 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 3.31: Bubbles ()Requirement 8) – Hilfslinien in Game

---

```
137     pygame.draw.rect(self._screen, (255, 0, 0), Settings.playground, 2)
138     for b in self._all_bubbles:
139         pygame.draw.rect(self._screen, (255, 0, 0), b.rect, 2)
```

---

## 3.1.9 Requirement 9: Zeitanpassungen

---

### Requirement 9 Zeitanpassungen

---

1. Die Blasen sollen nur noch alle 100 ms wachsen.
  2. Zu Beginn erscheint pro Sekunde eine Blase. Für jede 10 s Spielzeit, verkürzt sich das Intervall um 100 ms. Dabei darf der Wert von 400 ms nicht unterschritten werden.
- 

Die Blasen erscheinen und wachsen zu schnell. Die Zeiten sollen so angepasst werden, dass ein angenehmes Spielen möglich ist.

Für Requirement 9.1 wird in `Bubble` ein Timer-Objekt benötigt. Dieses wird im Konstruktor mit den richtigen Werten in Zeile 74 angelegt. In der Methode `update()` wird dann in Zeile 77 abgefragt, ob die 100 ms schon verstrichen sind. Wenn *Ja*, wird der Radius vergrößert; ansonsten passiert nichts.

Timer

---

#### Quelltext 3.32: Bubbles (Requirement 9.1) – Bubble

---

```
67 class Bubble(pygame.sprite.Sprite):
68     def __init__(self, bubble_container: BubbleContainer, filename: str = "blase1.png"):
69         super().__init__()
70         self._bubble_container = bubble_container
71         self.radius = Settings.radius["min"]
72         self.image = self._bubble_container.get(self.radius)
73         self.rect = self.image.get_rect()
74         self._timer_growth = Timer(100, False) # Wachstumsgeschwindigkeit
75
76     def update(self) -> None:
77         if self._timer_growth.is_next_stop_reached(): # Wachstum bremsen
78             self.radius = min(self.radius + 1, Settings.radius["max"])
79             center = self.rect.center
80             self.image = self._bubble_container.get(self.radius)
81             self.rect = self.image.get_rect()
82             self.rect.center = center
```

---

Die andere Zeitanpassung wird in `Game` gesteuert. Auch dazu wird ein `Timer`-Objekt verwendet. Dieses wird im Konstruktor in Zeile 121 erzeugt.

---

#### Quelltext 3.33: Bubbles (Requirement 9.2) – Konstruktor von `Game`

---

```
121     self._timer_bubble_duration = Timer(10000, False)      # Steuert Dauer
```

---

Für das Erstellen einer neuen Blase ist `spawn_bubble()` zuständig. Deshalb muss auch dort die entsprechende Anpassung erfolgen. Alle 10 s wird die Bedingung in Zeile 156 `True`. Im Anweisungsblock der Verzweigung wird dann für den Timer, der das Erscheinen der Blasen taktet (`_timer_bubble`) die Intervalldauer um 100 ms reduziert. Die Maximumsbildung in der Zeile danach stellt sicher, dass die Intervalldauer mindestens 400 ms beträgt.

---

#### Quelltext 3.34: Bubbles (Requirement 9.2) – `spawn_bubble()` von `Game`

---

```
155     def spawn_bubble(self) -> None:
156         if self._timer_bubble_duration.is_next_stop_reached():    # Schneller werden
157             self._timer_bubble.duration = max(self._timer_bubble.duration - 100, 400)
158         if self._timer_bubble.is_next_stop_reached():
159             if len(self._all_bubbles) <= Settings.max_bubbles:
160                 b = Bubble(self._bubble_container)
161                 tries = 100
162                 while tries > 0:
163                     b.randompos()
164                     b.radius += Settings.distance
165                     collided = pygame.sprite.spritecollide(b, self._all_bubbles, False,
166                                                 pygame.sprite.collide_circle)
167                     b.radius -= Settings.distance
168                     if collided:
169                         tries -= 1
170                     else:
171                         self._all_bubbles.add(b)
172                         break
```

---

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

### 3.1.10 Requirement 10: Kollision anzeigen

---

#### Requirement 10 Kollision anzeigen

---

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

---

Bisher beendet sich das Spiel so schnell, dass ich nicht überprüfen kann, ob ich eigentlich zu Recht verloren habe, oder ob das Programm spinnt. Ich möchte durch diese Anforderung die beiden kollidierenden Blasen oder die Blase, die den Rand berührt, andersfarbig sehen können. Ich habe dazu die Blase rot eingefärbt (siehe Abbildung 3.9).



Abb. 3.9: Blase 2

Dazu braucht es einen zweiten `BubbleContainer` mit den skalierten roten Blasen. Um auf diese leichter zugreifen zu können, sind diese aus `Game` nach `Settings` verschoben worden.

In Zeile 24 ist dazu ein Dictionary angelegt worden. Unter einem Schlüssel kann ich dort nun beliebige `BubbleContainer`-Objekte ablegen.

---

#### Quelltext 3.35: Bubbles (Requirement 10) – Erweiterung von `Settings`

---

```
24     bubble_container = {}# Mehrere
```

---

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

---

#### Quelltext 3.36: Bubbles (Requirement 10) – Änderung Konstruktor von `BubbleContainer`

---

```
54  class BubbleContainer:  
55      def __init__(self, filename: str) -> None:# Jetzt mit Dateinamen  
56          image = pygame.image.load(os.path.join(Settings.path["image"],  
57                                         filename)).convert_alpha()  
58          self._images = {  
59              i: pygame.transform.scale(image, (i * 2, i * 2))  
59              for i in range(Settings.radius["min"], Settings.radius["max"] + 1)}
```

---

Der Konstruktor von `Game` füllt nun das Dictionary `bubble_container` in `Settings` auf (Zeile 122 und Zeile 123).

---

#### Quelltext 3.37: Bubbles (Requirement 10) – Änderung vom Konstruktor von `Game`

---

```
117  def __init__(self) -> None:  
118      pygame.init()  
119      self._screen = pygame.display.set_mode(Settings.get_dim())  
120      pygame.display.set_caption(Settings.caption)  
121      self._clock = pygame.time.Clock()  
122      Settings.bubble_container["blue"] = BubbleContainer("blase1.png") # blau  
123      Settings.bubble_container["red"] = BubbleContainer("blase2.png") # rot  
124      self._background = pygame.sprite.GroupSingle(Background())  
125      self._points = pygame.sprite.GroupSingle(Points())  
126      self._timer_bubble = Timer(1000, False)  
127      self._timer_bubble_duration = Timer(10000, False)  
128      self._all_bubbles = pygame.sprite.Group()  
129      self._running = True
```

---

Durch das neue Attribut `mode` (Zeile 71) 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 81 in `update()` erwähnt.

---

#### Quelltext 3.38: Bubbles (Requirement 10) – Konstruktor von `Bubble` und `update()`

---

```
68  class Bubble(pygame.sprite.Sprite):  
69      def __init__(self) -> None:  
70          super().__init__()  
71          self.mode = "blue"# Farbmodus
```

```
72     self.radius = Settings.radius["min"]
73     self.image = Settings.bubble_container[self.mode].get(self.radius)
74     self.rect = self.image.get_rect()
75     self._timer_growth = Timer(100, False)
76
77     def update(self) -> None:
78         if self._timer_growth.is_next_stop_reached():
79             self.radius = min(self.radius + 1, Settings.radius["max"])
80             center = self.rect.center
81             self.image = Settings.bubble_container[self.mode].get(self.radius) #
82             self.rect = self.image.get_rect()
83             self.rect.center = center
```

---

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

Quelltext 3.39: Bubbles (Requirement 10) – `set_mode()` in `Bubble`

---

```
85     def set_mode(self, mode: str) -> None:
86         self.mode = mode
87         self.image = Settings.bubble_container[self.mode].get(self.radius)
```

---

Jetzt muss nur noch im Falle einer Kollision – also eines Spielendes – der Modus geändert werden. In Abbildung 3.10 auf der nächsten Seite können Sie sehen, wie die beiden kollidierenden Blasen rot erscheinen.

Quelltext 3.40: Bubbles (Requirement 10) – `check_bubblecollision()` in `Game`

---

```
196     def check_bubblecollision(self) -> bool:
197         for index1 in range(0, len(self._all_bubbles) - 1):
198             for index2 in range(index1 + 1, len(self._all_bubbles)):
199                 bubble1 = self._all_bubbles.sprites()[index1]
200                 bubble2 = self._all_bubbles.sprites()[index2]
201                 if pygame.sprite.collide_circle(bubble1, bubble2):
202                     bubble1.set_mode("red")
203                     bubble2.set_mode("red")
204                     return True
205                 if not Settings.playground.contains(bubble1):
206                     bubble1.set_mode("red")
207                     return True
208                 if not Settings.playground.contains(bubble2):
209                     bubble2.set_mode("red")
210                     return True
211
212     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 221).

`wait()`

Quelltext 3.41: Bubbles (Requirement 10) – Wartezeit in `run()`

---

```
213     def run(self) -> None:
214         self._running = True
215         while self._running:
216             self._clock.tick(Settings.fps)
217             self.watch_for_events()
218             self.update()
219             self.draw()
220
```

```
221     pygame.time.wait(2000)           # Kurz warten
222     pygame.quit()
```

---



Abbildung 3.10: Bubbles: Kollision anzeigen

#### 3.1.11 Requirement 11: Pause

---

##### Requirement 11 Pause

---

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

---

Die Idee hinter dieser Anforderung ist, dass eine notwendige Unterbrechung nicht zwangsläufig bedeutet, dass man verliert. In Abbildung 3.11 auf der nächsten Seite können Sie sehen, wie der Pausenbildschirm aussehen sollte. Zunächst werden die Pygame-Konstanten KEYUP und K\_p importiert (Zeile 7).

---

##### Quelltext 3.42: Bubbles (Requirement 11) – Zusätzliche Konstanten

---

```
7 from pygame.constants import (K_ESCAPE, KEYDOWN,   #
8                               KEYUP, QUIT, K_p)
```

---

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

---

##### Quelltext 3.43: Bubbles (Requirement 11) – Konstruktor in Game

---

```
148     self._running = True
149     self._pause = Pause()           #
```

---

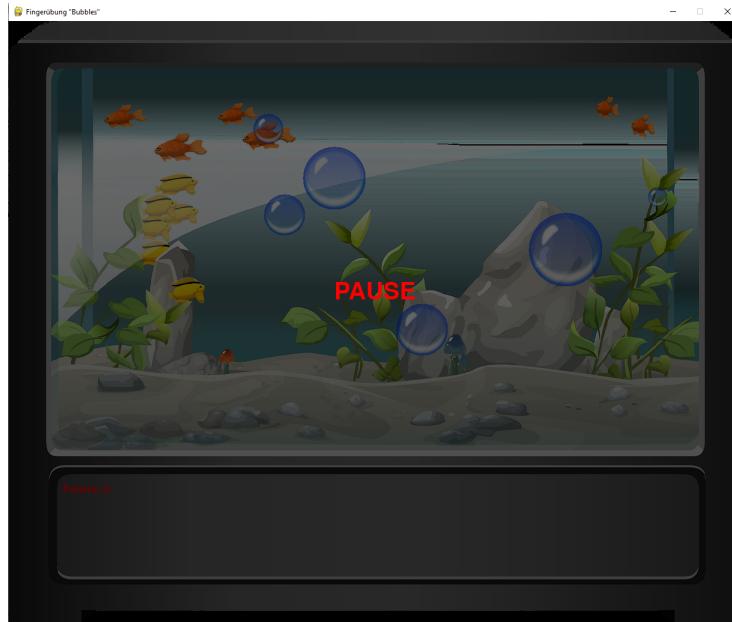


Abbildung 3.11: Bubbles: Pausenbildschirm

In `watch_for_events()` wird nun abgefragt, ob die P-Taste (Zeile 160) oder die rechte Maustaste (Zeile 165) gedrückt wurde. Falls *Ja*, wird der Boolesche-Wert des Flags negiert ([Toggling](#)).

---

#### Quelltext 3.44: Bubbles (Requirement 11) – `watch_for_events()` in Game

---

```
151
152     def watch_for_events(self) -> None:
153         for event in pygame.event.get():
154             if event.type == QUIT:
155                 self._running = False
156             elif event.type == KEYDOWN:
157                 if event.key == K_ESCAPE:
158                     self._running = False
159                 elif event.type == KEYUP:
160                     if event.key == K_p:                      #
161                         self._pausing = not self._pausing
162                     elif event.type == pygame.MOUSEBUTTONUP:
163                         if event.button == 1:                  # left
164                             self.sting(pygame.mouse.get_pos())
165                         elif event.button == 3:                # right
```

---

Für die Darstellung der Pause, habe ich die – vielleicht etwas überflüssige – Klasse `Pause` implementiert. Im Konstruktor wird ein graues Image mit einer Halbtransparenz erzeugt (Zeile 60). Ebenso wird der große rote Schriftzug *Pause* gerendert. Für beide sind `Image`- und `Rect`-Objekte vorhanden, so dass sie leicht über `draw()` ausgegeben werden können.

---

#### Quelltext 3.45: Bubbles (Requirement 11) – Pause

---

```
55  class Pause:
56      def __init__(self) -> None:
57          super().__init__()
58          self._image = pygame.image.load(os.path.join(Settings.path["image"],
59                                         "hintergrund.png")).convert_alpha()
60          self._image = pygame.transform.scale(self._image, Settings.get_dim())
61          self._image.set_alpha(150)                                     # Halbtransparenz
62          self._rect_image = self._image.get_rect()
63          font_bigsize = pygame.font.Font(pygame.font.get_default_font(), 40)
64          self._text = font_bigsize.render("PAUSE", True, (255, 0, 0))
65          self._rect_text = self._text.get_rect()
66          self._rect_text.centerx = Settings.window["width"] // 2
67          self._rect_text.centery = Settings.window["height"] // 2 - 50
68
69      def draw(self, screen: pygame.surface.Surface) -> None:
70          screen.blit(self._image, self._rect_image)
```

---

Im Konstruktor von Game wird ein Objekt der Klasse Pause angelegt (siehe Quelltext 3.43 auf Seite 97), damit es in `draw()` verwendet werden kann. In Zeile 175 wird das Flag abgefragt; ggf. wird dann das halbtransparente Bitmaps und der Schiftzug über das Bild gemalt.

#### Quelltext 3.46: Bubbles (Requirement 11) – `draw()` in Game

---

```
167
168      def draw(self) -> None:
169          self._background.draw(self._screen)
170          self._all_bubbles.draw(self._screen)
171          self._points.draw(self._screen)
172          # pygame.draw.rect(self._screen, (255, 0, 0), Settings.playground, 2)
173          # for b in self._all_bubbles:
174          #     pygame.draw.rect(self._screen, (255, 0, 0), b.rect, 2)
175          if self._pausing:                                         # Pausenbildschirm
176              self._pause.draw(self._screen)
```

---

Aber wie wird nun verhindert, dass das Spiel während der Pause weiterläuft? Wenn wir uns in Erinnerung rufen, dass alle Zuständsänderungen von `update()` ausgehen, muss dort die Anpassung vorgenommen werden. Die Lösung ist dann recht schnell gefunden. In Zeile 181 wird einfach abgefragt, ob das Spiel sich im Pausenmodus befindet. Falls *Ja*, werden die Aufrufe der `update()`-Methoden der Sprites übersprungen – das war's.

#### Quelltext 3.47: Bubbles (Requirement 11) – `update()` in Game

---

```
179
180      def update(self) -> None:
181          if not self._pausing and self._running:                  # Pausenbildschirm
182              if self.check_bubblecollision():
183                  self._running = False
184              else:
185                  self._all_bubbles.update()
186                  self._points.update()
187                  self.spawn_bubble()
```

---

### 3.1.12 Requirement 12: Neustart

---

#### Requirement 12 Neustart

---

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

---

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

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

Zunächst werden die Tastaturkonstanten für die Antwort importiert (Zeile 8).

---

#### Quelltext 3.48: Bubbles (Requirement 12) – Erweiterung des Imports

---

```
8 from pygame.constants import (K_ESCAPE, KEYDOWN, #  
9                         KEYUP, QUIT, K_j, K_n, K_p)
```

---

Die Aufgabe, eine Rückfrage in den Vordergrund zu schieben, ist eigentlich schon mit der Klasse `Pause` gelöst; ich kann daher die Klasse verallgemeinern, indem ich sie in `Foreground` umbenenne und den Text dem Konstruktor als String-Parameter übergebe (Zeile 57 und Zeile 64).

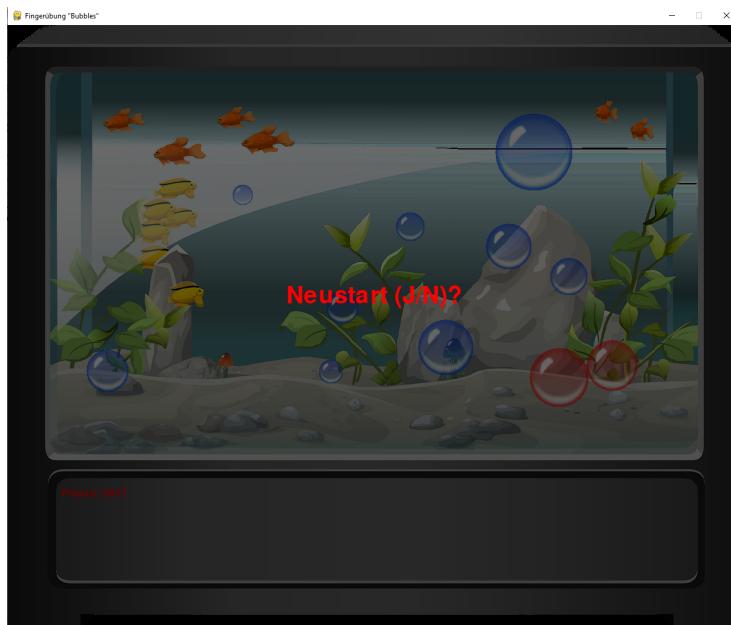


Abbildung 3.12: Bubbles: Neustartbildschirm

#### Quelltext 3.49: Bubbles (Requirement 12) – von Pause zu Foreground

---

```
56  class Foreground:
57      def __init__(self, message: str) -> None:                      # Nachrichtentext
58          super().__init__()
59          self._image = pygame.image.load(os.path.join(Settings.path["image"],
60                                              "hintergrund.png")).convert_alpha()
61          self._image = pygame.transform.scale(self._image, Settings.get_dim())
62          self._image.set_alpha(150)
63          self._rect_image = self._image.get_rect()
64          font_bigsize = pygame.font.Font(pygame.font.get_default_font(), 40)
65          self._text = font_bigsize.render(message, True, (255, 0, 0))  # rendern
66          self._rect_text = self._text.get_rect()
67          self._rect_text.centerx = Settings.window["width"] // 2
68          self._rect_text.centery = Settings.window["height"] // 2 - 50
69
70      def draw(self, screen: pygame.surface.Surface) -> None:
71          screen.blit(self._image, self._rect_image)
72          screen.blit(self._text, self._rect_text)
```

---

In Game werden im Konstruktor drei neue Attribute angelegt. Mit dem Flag `_restarting` merkt sich das Spiel, ob der Neustart-Vordergrund angezeigt werden soll und in `_do_start` wird abgelegt, ob der Anwender noch ein Spiel spielen möchte.

#### Quelltext 3.50: Bubbles (Requirement 12) – Neue Attribute im Konstruktor von Game

---

```
150      self._pause = Foreground("PAUSE")
151      self._pausing = False
152      self._restart = Foreground("Neustart (J/N)?")
153      self._restarting = False
154      self._do_start = False
```

---

Die Antwort muss in `watch_for_events()` abgefragt und in entsprechende Flaginhalte umgesetzt werden. 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 3.51: Bubbles (Requirement 12) – Erweiterung von `watch_for_events()`

---

```
156      def watch_for_events(self) -> None:
157          for event in pygame.event.get():
158              if event.type == QUIT:
159                  self._running = False
160              elif event.type == KEYDOWN:
161                  if event.key == K_ESCAPE:
162                      self._running = False
163              elif event.type == KEYUP:
164                  if event.key == K_p:
165                      self._pausing = not self._pausing
166                  elif event.key == K_j:                      #
167                      self._do_start = True
168                  elif event.key == K_n:                      #
169                      self._running = False
170              elif event.type == pygame.MOUSEBUTTONUP:
171                  if event.button == 1:
172                      self.sting(pygame.mouse.get_pos())
173                  elif event.button == 3:
174                      self._pausing = not self._pausing
```

---

In `update()` muss nun die Verarbeitung der Flags erfolgen. Gleich zu Beginn wird in Zeile 193 abgefragt, ob der Spieler neu starten will – also ein *J* eingegeben hat. Der Punktestand wird zurückgesetzt; das Gleiche passiert mit den die Spielschwierigkeit steuernden `Timer`-Objekten.

Danach werden die Flags wieder auf `False` gesetzt, damit die Vordergrundanzeige verschwindet und das Spiel beim nächsten Frame nicht wieder einen Neustart durchführt.

---

#### Quelltext 3.52: Bubbles (Requirement 12) – Erweiterung von `update()` in Game

---

```
193     if self._do_start:                                     # Neustart?
194         Settings.points = 0
195         self._all_bubbles.empty()
196         self._timer_bubble = Timer(1000, False)
197         self._timer_bubble_duration = Timer(10000, False)
198         self._do_start = False
199         self._restarting = False
200     if not self._pausing and self._running:
201         if self.check_bubblecollision():
202             self._restarting = True
203         else:
204             self._all_bubbles.update()
205             self._points.update()
206             self.spawn_bubble()
207             self.set_mousecursor()
```

---

In `draw()` wird nun ermittelt, ob die Vordergrundausgabe über das Spiel gelegt werden soll. Dazu wird in Zeile 185 das Flag `_restarting` abgefragt, um die entsprechende Methode in `Foreground` aufzurufen.

---

#### Quelltext 3.53: Bubbles (Requirement 12) – Erweiterung von `draw()` in Game

---

```
176     def draw(self) -> None:
177         self._background.draw(self._screen)
178         self._all_bubbles.draw(self._screen)
179         self._points.draw(self._screen)
180         # pygame.draw.rect(self._screen, (255, 0, 0), Settings.playground, 2)
181         # for b in self._all_bubbles:
182         #     pygame.draw.rect(self._screen, (255, 0, 0), b.rect, 2)
183         if self._pausing:
184             self._pause.draw(self._screen)
185         elif self._restarting:                                # Abfragebildschirm?
186             self._restart.draw(self._screen)
187
188         pygame.display.flip()
```

---

Da wir nun am Spielende eine halbtransparente Vordergrundausgabe haben, brauchen wir keine zweisekündige Pause, um die kollidierenden Blasen anzusehen (siehe Abbildung 3.12 auf Seite 100). Daher kann Zeile 269 auskommentiert werden.

---

#### Quelltext 3.54: Bubbles (Requirement 12) – `run()` in Game

---

```
262         self._running = True
263         while self._running:
264             self._clock.tick(Settings.fps)
265             self.watch_for_events()
```

```
266         self.update()
267         self.draw()
268
269         # pygame.time.wait(2000)                                #
270         pygame.quit()
```

---

#### 3.1.13 Requirement 13: Sound

##### Requirement 13 Sound

---

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

Zum Schluss noch eine kleine Sound-Untermalung. Ähnlich wie bei den Blasen-Sprites, möchte ich keine Performance durch permanentes Laden der Sound-Dateien verlieren. Daher werden die Sounds in einer Container-Klasse abgelegt.

Die Klasse für Soundeffekte ist `pygame.mixer.Sound`. Dem Konstruktor wird dabei der Dateiname der Sound-Datei übergeben.

Sound

##### Quelltext 3.55: Bubbles (Requirement 13) – SoundContainer

---

```
72 class SoundContainer:                                         #
73     def __init__(self) -> None:
74         self._sounds = {}
75         self._sounds["bubble"] = pygame.mixer.Sound(os.path.join(Settings.path["sound"],
76                         "plopp1.mp3"))
76         self._sounds["burst"] = pygame.mixer.Sound(os.path.join(Settings.path["sound"],
77                         "burst.mp3"))
77         self._sounds["clash"] = pygame.mixer.Sound(os.path.join(Settings.path["sound"],
78                         "glas.wav"))
78
79     def get(self, effect: str) -> pygame.mixer.Sound:
80         return self._sounds[effect]
```

---

Im Konstruktor von `Game` wird das Container-Objekt definiert (Zeile 148).

##### Quelltext 3.56: Bubbles (Requirement 13) – SoundContainer

---

```
145 class Game:
146     def __init__(self) -> None:
147         pygame.init()
148         self._sounds = SoundContainer()                           #
149         self._screen = pygame.display.set_mode(Settings.get_dim())
```

---

Nun müssen die Sounds nur noch an der geeigneten Stelle mit `pygame.mixer.Sound.play()` abgespielt werden: in `spawn_bubble()` in Zeile 235.

play()

#### Quelltext 3.57: Bubbles (Requirement 13.1) – `spawn_bubble()`

---

```
231         if collided:
232             tries -= 1
233         else:
234             self._all_bubbles.add(b)
235             self._sounds.get("bubble").play()      #
236             break
```

---

und in `sting()` in Zeile 253:

#### Quelltext 3.58: Bubbles (Requirement 13.2) – `sting()`

---

```
250     def sting(self, mousepos) -> None:
251         for bubble in self._all_bubbles:
252             if bubble.collidepoint(mousepos):
253                 self._sounds.get("burst").play()      #
254                 bubble.stung()
```

---

und in `update()`. Dabei muss noch berücksichtigt werden, ob das Spiel gerade den Vordergrund für den Neustart anzeigt. Wenn *Ja* (Zeile 210), darf der Sound nicht noch einmal abgespielt werden; ansonsten wird permanent der Berühren-Sound abgespielt (Zeile 211).

#### Quelltext 3.59: Bubbles (Requirement 13.3) – `update()`

---

```
204     self._timer_bubble = Timer(1000, False)
205     self._timer_bubble_duration = Timer(10000, False)
206     self._do_start = False
207     self._restarting = False
208     if not self._pausing and self._running:
209         if self.check_bubblecollision():
210             if not self._restarting:                      #
211                 self._sounds.get("clash").play()          #
212                 self._restarting = True
213             else:
214                 self._all_bubbles.update()
215                 self._points.update()
216                 self.spawn_bubble()
217                 self.set_mousecursor()
218
219     def spawn_bubble(self) -> None:
220         if self._timer_bubble_duration.is_next_stop_reached():
221             self._timer_bubble.duration = max(self._timer_bubble.duration - 100, 400)
```

---

Und Schluss :-)

# Abbildungsverzeichnis

2.1	Eine einfache grüne Spielfläche . . . . .	6
2.2	Ressourcenverbrauch ohne Taktung . . . . .	6
2.3	Ressourcenverbrauch mit Taktung . . . . .	7
2.4	Einige Grafikprimitive . . . . .	10
2.5	Bitmaps laden und ausgeben, Version 1.0 . . . . .	13
2.6	Größen OK . . . . .	14
2.7	Transparenz OK . . . . .	15
2.8	Bitmaps positionieren (Verteidiger) . . . . .	16
2.9	Bitmaps positionieren (Angreifer, Version 1) . . . . .	17
2.10	Bitmaps positionieren (Angreifer, Version 2) . . . . .	18
2.11	Bitmaps positionieren (Angreifer, Version 3) . . . . .	19
2.12	Elemente eines Rect-Objekts . . . . .	20
2.13	Bitmaps bewegen, Version 1.0 . . . . .	22
2.14	Der Verteidiger bewegt sich und prallt ab . . . . .	24
2.15	Ränder . . . . .	30
2.16	Textausgabe mit Fonts . . . . .	37
2.17	Fontliste . . . . .	40
2.18	Beispiel für eine Spritelib . . . . .	44
2.19	Bedeutung der Angaben in Spritelib . . . . .	46
2.20	Textausgabe mit Bitmaps . . . . .	49
2.21	Kollisionserkennung mit Rechtecken . . . . .	50
2.22	Kollisionserkennung mit Kreisen . . . . .	51
2.23	Kollisionsprüfung: 4 Sprites ohne Kollision . . . . .	52
2.24	Kollisionsprüfung durch Rechtecke (Montage) . . . . .	52
2.25	Kollisionsprüfung durch Kreise (Montage) . . . . .	53
2.26	Kollisionsprüfung durch Masken (Montage) . . . . .	53
2.27	Feuerball ohne Zeitsteuerung . . . . .	59
2.28	Feuerball mit Zeitsteuerung . . . . .	63
2.29	Animation einer Katze: Einzelsprites . . . . .	65
2.30	Animation einer Explosion: Einzelsprites . . . . .	71
2.31	Mausaktionen . . . . .	73
3.1	Bubbles: Hintergrundbild (aquarium.png) . . . . .	80
3.2	Blase . . . . .	81
3.3	Bubbles: Die Blasen haben beim Start einen Mindestabstand . . . . .	84
3.4	Bubbles: Die Blasen sind gewachsen/verwachsen . . . . .	87

3.5	Kollisionserkennung: Punkt innerhalb des Kreises? . . . . .	87
3.6	Bubbles: Ausgabe Punktestand . . . . .	91
3.7	Bubbles – Kollision mit dem Rand . . . . .	92
3.8	Bubbles – Kollision der Blasen . . . . .	92
3.9	Blase 2 . . . . .	94
3.10	Bubbles: Kollision anzeigen . . . . .	97
3.11	Bubbles: Pausenbildschirm . . . . .	98
3.12	Bubbles: Neustartbildschirm . . . . .	100

# Quelltexte

2.1	Mein erstes <i>Spiel</i> , Version 1.0 . . . . .	4
2.2	Mein erstes <i>Spiel</i> , Version 1.1 . . . . .	7
2.3	Mein zweites <i>Spiel</i> , Version 1.0 . . . . .	8
2.4	Bitmaps laden und ausgeben, Version 1.0 . . . . .	12
2.5	Bitmaps laden und ausgeben, Version 1.1 . . . . .	13
2.6	Bitmaps laden und ausgeben, Version 1.2 . . . . .	14
2.7	Bitmap: Positionen, Version 1.4 . . . . .	15
2.8	Bitmaps bewegen, Version 1.0 . . . . .	21
2.9	Bitmaps bewegen, Version 1.2 . . . . .	22
2.10	Bitmaps bewegen, Version 1.3 . . . . .	23
2.11	Bitmaps bewegen, Version 1.4 . . . . .	23
2.12	Sprites (1), Version 1.0 . . . . .	25
2.13	Sprites (2), Version 1.0 . . . . .	26
2.14	Sprites (1), Version 1.1 . . . . .	26
2.15	Sprites (2), Version 1.1 . . . . .	26
2.16	Sprites (3), Version 1.1 . . . . .	27
2.17	Sprites (4), Version 1.1 . . . . .	27
2.18	Sprites (1), Version 1.2 . . . . .	27
2.19	Game-Klasse . . . . .	28
2.20	Bewegung durch Tastatur steuern (1), <code>Defender</code> . . . . .	30
2.21	Bewegung durch Tastatur steuern (2), <code>Border</code> . . . . .	31
2.22	Bewegung durch Tastatur steuern (3), <code>Game</code> -Konstruktor . . . . .	32
2.23	Bewegung durch Tastatur steuern (4), <code>Game.watch_for_events()</code> . . . . .	32
2.24	Text mit Fonts ausgeben (1), Präambel . . . . .	38
2.25	Text mit Fonts ausgeben (2), <code>TextSprite</code> . . . . .	38
2.26	Text mit Fonts ausgeben (3), Hauptprogramm . . . . .	39
2.27	Fontliste (1), Präambel, <code>Settings</code> und <code>Textsprite</code> . . . . .	40
2.28	Fontliste (2), <code>BigImage</code> . . . . .	41
2.29	Fontliste (3), Hauptprogramm (1) . . . . .	42
2.30	Fontliste, Hauptprogramm (2) . . . . .	43
2.31	Textbitmaps (1), Präambel und <code>Settings</code> . . . . .	44
2.32	Textbitmaps (2), <code>Spritelib</code> . . . . .	46
2.33	Textbitmaps (3): Konstruktor von <code>Latters</code> . . . . .	46
2.34	Textbitmaps (4): <code>create_letter_bitmap()</code> von <code>Latters</code> . . . . .	47
2.35	Textbitmaps (5): <code>get_letter()</code> und <code>get_text()</code> von <code>Latters</code> . . . . .	47
2.36	Textbitmaps (6): <code>TextBitmaps</code> . . . . .	48

2.37	Textbitmaps (7): Hauptprogramm . . . . .	49
2.38	Kollisionsarten (1): Präambel und <code>Settings</code> . . . . .	53
2.39	Kollisionsarten (2): <code>Obstacle</code> . . . . .	54
2.40	Kollisionsarten (3): <code>Bullet</code> . . . . .	54
2.41	Kollisionsarten (4): Konstruktor von <code>Game</code> , Konstruktor . . . . .	55
2.42	Kollisionsarten (5): <code>run()</code> und <code>watch_for_events()</code> von <code>Game</code> . . . . .	55
2.43	Kollisionsarten (6): <code>update()</code> und <code>draw()</code> von <code>Game</code> . . . . .	56
2.44	Kollisionsarten (7): <code>resize()</code> von <code>Game</code> . . . . .	56
2.45	Kollisionsarten (8): <code>check_for_collision()</code> von <code>Game</code> . . . . .	57
2.46	Kollisionsarten (9): Variante von <code>check_for_collision()</code> von <code>Game</code> . . . . .	57
2.47	Kollisionsarten (10): Der Aufruf von <code>Game</code> . . . . .	58
2.48	Zeitsteuerung (1), Version 1.0: Präambel und <code>Settings</code> . . . . .	58
2.49	Zeitsteuerung (2), Version 1.0: <code>Enemy</code> . . . . .	59
2.50	Zeitsteuerung (3), Version 1.0: <code>Bullet</code> . . . . .	60
2.51	Zeitsteuerung (4), Version 1.0: Konstruktor und <code>run()</code> von <code>Game</code> . . . . .	60
2.52	Zeitsteuerung (5), Version 1.0: <code>watch_for_events()</code> und <code>draw()</code> von <code>Game</code> . . . . .	61
2.53	Zeitsteuerung (6), Version 1.0: <code>update()</code> und <code>new_bullet()</code> von <code>Game</code> . . . . .	61
2.54	Zeitsteuerung (7), Version 1.1: Konstruktor von <code>Game</code> . . . . .	62
2.55	Zeitsteuerung (8), Version 1.1: <code>new_bullet()</code> von <code>Game</code> . . . . .	62
2.56	Zeitsteuerung (9), Version 1.2: Konstruktor von <code>Game</code> . . . . .	63
2.57	Zeitsteuerung (10), Version 1.2: <code>new_bullet()</code> von <code>Game</code> . . . . .	63
2.58	Zeitsteuerung (11), Version 1.3: <code>Timer</code> . . . . .	64
2.59	Zeitsteuerung (12), Version 1.3: <code>Timer</code> -Objekt erzeugen . . . . .	64
2.60	Zeitsteuerung (13), Version 1.3: <code>Timer</code> -Objekt verwenden . . . . .	64
2.61	Animation einer Katze (1), Version 1.0: Präambel, <code>Timer</code> und <code>Settings</code>	65
2.62	Animation einer Katze (2), Version 1.0: <code>Cat</code> . . . . .	67
2.63	Animation einer Katze (3), Version 1.0: Konstruktor und <code>run()</code> . . . . .	67
2.64	Animation einer Katze (4), Version 1.0: <code>watch_for_events()</code> . . . . .	68
2.65	Animation einer Katze (5), Version 1.0: <code>update()</code> und <code>draw()</code> . . . . .	68
2.66	Animation (6), Version 1.1: <code>Animation</code> . . . . .	69
2.67	Animation einer Katze (7), Version 1.1: <code>Cat</code> . . . . .	70
2.68	Animation einer Explosion (1): <code>Rock</code> . . . . .	71
2.69	Animation einer Explosion (2): <code>ExplosionAnimation</code> . . . . .	71
2.70	Mausaktionen: <code>Statics</code> und Konstruktor von <code>Game</code> . . . . .	73
2.71	Mausaktionen: <code>Game.run()</code> . . . . .	74
2.72	Mausaktionen: <code>Game.watch_for_events()</code> . . . . .	74
2.73	Mausaktionen: <code>Game.update()</code> und <code>Game.draw()</code> . . . . .	75
2.74	Mausaktionen: <code>Ball</code> . . . . .	76
3.1	Bubbles (Requirement 1.1) – Präambel und <code>Settings</code> . . . . .	79
3.2	Bubbles (Requirement 1.2) – <code>Background</code> . . . . .	79
3.3	Bubbles (Requirement 1) – <code>Game</code> . . . . .	79
3.4	Bubbles (Requirement 1) – Aufruf . . . . .	81
3.5	Bubbles (Requirement 2) – Ergänzungen in <code>Settings</code> . . . . .	81

3.6	Bubbles (Requirement 2) – Timer . . . . .	81
3.7	Bubbles (Requirement 2) – Bubble . . . . .	82
3.8	Bubbles (Requirement 2) – Konstruktor von Game . . . . .	82
3.9	Bubbles (Requirement 2) – draw() und update() von Game . . . . .	83
3.10	Bubbles (Requirement 2) – spawn_bubble() von Game . . . . .	83
3.11	Bubbles (Requirement 3) – Ergänzung von Settings . . . . .	84
3.12	Bubbles (Requirement 3) – Ergänzung von Game in spawn_bubbles() . . .	84
3.13	Bubbles (Requirement 4.1) – BubbleContainer . . . . .	85
3.14	Bubbles (Requirement 4.2) – Erweiterung von Settings . . . . .	85
3.15	Bubbles (Requirement 4) – Ergänzung von Bubble . . . . .	86
3.16	Bubbles (Requirement 4) – Ergänzung von update() in Game . . . . .	86
3.17	Bubbles (Requirement 4) – Ergänzung von spawn_bubble() in Game . . .	86
3.18	Bubbles (Requirement 5) – collidepoint() in Bubble . . . . .	88
3.19	Bubbles (Requirement 5) – set_mousecursor() in Game . . . . .	88
3.20	Bubbles (Requirement 6) – watch_for_event() in Game . . . . .	89
3.21	Bubbles (Requirement 6) – sting() in Game . . . . .	89
3.22	Bubbles (Requirement 7.1) – Erweiterung von Settings . . . . .	89
3.23	Bubbles (Requirement 7.2) – stung() in Bubble . . . . .	90
3.24	Bubbles (Requirement 7.2) – sting() in Game . . . . .	90
3.25	Bubbles (Requirement 7.3) – Erweiterung von Settings . . . . .	90
3.26	Bubbles (Requirement 7.3) – Points . . . . .	90
3.27	Bubbles (Requirement 7.3) – Erweiterung des Konstruktors von Game . .	90
3.28	Bubbles (Requirement 7.3) – Erweiterung von update(), draw() in Game	91
3.29	Bubbles (Requirement 8) – Erweiterung von update() in Game . . . . .	91
3.30	Bubbles (Requirement 8) – check_bubblecollision() in Game . . . . .	92
3.31	Bubbles ()Requirement 8) – Hilfslinien in Game . . . . .	93
3.32	Bubbles (Requirement 9.1) – Bubble . . . . .	93
3.33	Bubbles (Requirement 9.2) – Konstruktor von Game . . . . .	94
3.34	Bubbles (Requirement 9.2) – spawn_bubble() von Game . . . . .	94
3.35	Bubbles (Requirement 10) – Erweiterung von Settings . . . . .	95
3.36	Bubbles (Requirement 10) – Änderung Konstruktor von BubbleContainer	95
3.37	Bubbles (Requirement 10) – Änderung vom Konstruktor von Game . . .	95
3.38	Bubbles (Requirement 10) – Konstruktor von Bubble und update() . . .	95
3.39	Bubbles (Requirement 10) – set_mode() in Bubble . . . . .	96
3.40	Bubbles (Requirement 10) – check_bubblecollision() in Game . . . . .	96
3.41	Bubbles (Requirement 10) – Wartezeit in run() . . . . .	96
3.42	Bubbles (Requirement 11) – Zusätzliche Konstanten . . . . .	97
3.43	Bubbles (Requirement 11) – Konstruktor in Game . . . . .	97
3.44	Bubbles (Requirement 11) – watch_for_events() in Game . . . . .	98
3.45	Bubbles (Requirement 11) – Pause . . . . .	98
3.46	Bubbles (Requirement 11) – draw() in Game . . . . .	99
3.47	Bubbles (Requirement 11) – update() in Game . . . . .	99
3.48	Bubbles (Requirement 12) – Erweiterung des Imports . . . . .	100
3.49	Bubbles (Requirement 12) – von Pause zu Foreground . . . . .	101

3.50	Bubbles (Requirement 12) – Neue Attribute im Konstruktor von Game . . .	101
3.51	Bubbles (Requirement 12) – Erweiterung von <code>watch_for_events()</code> . . . .	101
3.52	Bubbles (Requirement 12) – Erweiterung von <code>update()</code> in Game . . . . .	102
3.53	Bubbles (Requirement 12) – Erweiterung von <code>draw()</code> in Game . . . . .	102
3.54	Bubbles (Requirement 12) – <code>run()</code> in Game . . . . .	102
3.55	Bubbles (Requirement 13) – SoundContainer . . . . .	103
3.56	Bubbles (Requirement 13) – SoundContainer . . . . .	103
3.57	Bubbles (Requirement 13.1) – <code>spawn_bubble()</code> . . . . .	104
3.58	Bubbles (Requirement 13.2) – <code>sting()</code> . . . . .	104
3.59	Bubbles (Requirement 13.3) – <code>update()</code> . . . . .	104

# 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. [15](#)

**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. [9](#)

**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. [48](#)

**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. [5](#)

**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, ... [30](#)

**Dictionary** Eine Datenstruktur, welche Werte unter einem einzigartigen Schlüssel ablegt. Andere Namen sind: Zuordnungstabelle, assoziatives Array, Hashtable. [46](#)

**Doublebuffer** Dies ist ein zweiter Speicherbereich, der genauso groß ist wie der Bildschirmspeicher. Wird jetzt etwas auf die Spielfläche gezeichnet, passiert dies zunächst auf diesem zweiten Speicher. Erst wenn alle Spielemente ihr neues Aussehen gemalt haben, wird mit einem Schlag der alte Bildschirmspeicher mit dem zweiten ausgetauscht. Bei bestimmten Hardware- oder Grafikkonfigurationen kann es passieren, dass der Bildschirmspeicher neu gemalt wird, obwohl das Spiel noch nicht alle neuen Zustände abgebildet hat. Dadurch können hässliche Artefakte entstehen. Durch das Doublebuffering wird dieser Effekt vermieden. [5](#)

**DTP-Punkt** Maßeinheit für Schriftgrößen. [116](#)

**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:`. [83](#)

**Flag** Eine meist boolsche Variable, die eine Operation/Schleife ein- und ausschaltet. [5](#)

**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. [38](#)

**frames per second** Maximale Anzahl der Bilder pro Sekunde. [7](#), [116](#)

**Framework** In der Informatik ist damit eine Arbeitsumgebung gemeint. Dies können einzelne Klassen, Funktionsbibliotheken oder ganze [Integrated Development Environment \(IDE\)](#) sein. [25](#)

**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. [4](#)

**Grad (°)** Maßeinheit für einen Winkel. Der Vollkreis hat dabei 360°. [76](#)

**Hauptprogrammschleife** Jedes nichttriviale 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. [5](#)

**Integrated Development Environment** Integrierte Entwicklungsumgebung. Diese heißen *integriert*, da sie nicht nur einen Compiler und Linker enthalten, sondern auch einen Editor, Debugger, Profiler etc. [112](#), [116](#)

**Joint Photographic Experts Group** Verlustbehaftete komprimierte Bildinformationen. [111](#), [116](#)

**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*. [4](#)

**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. [19](#)

**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. [4](#)

**Linienzug** Eine Folge miteinander verbundener Linien. Wird meist durch eine Folge von Punkten definiert. Bei einem geschlossenen Linienzug spricht man von einem **Polygon**. [11](#), [113](#)

**List Comprehension** In Python kann man den Inhalte einer Liste, eines Tupels, eines Arrays oder eines Dictionarys nicht nur durch explizite Vorgaben festlegen, sondern auch indem man eine Generierungsvorschrift formuliert: `squares = [x**2 for x in range(10)]`. [59](#)

**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. [51](#)

**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. [5](#)

**Millisekunden** Der 1/1000 Teil einer Sekunde. [63](#), [116](#)

**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. [4](#)

**Objektorientiert** Die Analyse, das Design oder die Implementierung entspricht den allgemeinen Vorgaben der Objektorientierung. [116](#)

**Pixel** Die kleinste bei gegebener Auflösung ansteuerbare Bildschirmfläche. [5](#), [116](#)

**Polygon** Ein geschlossener **Linienzug**. Wird meist durch eine Folge von Punkten definiert, wobei der letzte Punkt mit dem ersten verbunden wird. [11](#), [113](#)

**Portable Network Graphics** Verlustfrei komprimierte Bildinformationen. [111](#), [116](#)

**Pygame** Pygame ist ein Verbund von Modulen, der die Entwicklung von Computerspielen in Python unterstützt. [3](#)

**Pylance** Pylance ist die Standard Python-Erweiterung von Visual Code zur Unterstützung der Python-Programmierung. Seine wesentlichen Features sind die Typüberwachung und Auto vervollständigung. [75](#)

**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. [3](#)

**Radiant (rad)** Maßeinheit für einen Winkel. Der Vollkreis hat dabei  $2\pi rad$ . [76](#)

**Red Green Blue** Additive Farbkodierung. [116](#)

**Rendern** Das Erzeugen eines Bildes – meist in Bitmap-Format – aus einer Bildbeschreibungsangabe. [38](#)

**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. [87](#)

**Semantik** Bedeutung einer Angabe. Wird meist in Abgrenzung zu **Syntax** einer Angabe verwendet. [9, 114](#)

**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. [28, 116](#)

**Slicing** Eine Technik, mit deren Hilfe man Teilmengen aus Strings oder Arrays bequem ausschneiden oder extrahieren kann. [48](#)

**Solid-State-Drive** Festspeicherplattentechnologie, welche nicht auf magnetische Prinzipien, sondern auf Halbleitertechnik basiert. [116](#)

**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)*. [13, 114](#)

**Spritelib** Meist eine Grafikdatei im Bitmap-Format, welches viele einzelne **Sprites** enthält. [43](#)

**Syntax** Form oder Grammatik einer Angabe. Wird meist in Abgrenzung zur **Semantik** einer Angabe verwendet. [114](#)

**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*. [98](#)

**True Type Font** Die Schriftinformation wird nicht im Bitmap-Format, sondern in einer Art Vektorgrafikformat abgespeichert. Dadurch lassen sich *beliebige* Schriftgrößen generieren. [116](#)

**Umgebungsvariable** Dies sind Variablen, die nicht vom Programm, sondern von der Programmumgebung verwaltet werden. Die Programmumgebung kann das Betriebssystem sein, aber auch eine Server. Über Umgebungsvariablen kann die Umgebung mit meinem Programm Informationen austauschen. In unserem Beispiel wird der Fensterverwaltung bzw. dem Betriebssystem mitgeteilt, an welcher Koordinate die linke obere Ecke des Fensters auf dem Bildschirm erscheinen soll. [4](#)

**Unicode** Ein Verfahren zur Kodierung von Zeichen und Symbolen. Gängige Umsetzungen sind UTF-8, UTF-16 und UTF-32. [48](#)

**Universal Serial Bus** Bitserielles Datenübertragungsprotokoll. [116](#)

**Windows Bitmap Format** Bildinformationen im Windows Bitmap-Format. [111](#), [116](#)

# Akronyme

**bmp** Windows Bitmap Format. [111](#), [116](#), *Glossar: Windows Bitmap Format*

**fps** frames per second. [7](#), [116](#), *Glossar: frames per second*

**IDE** Integrated Development Environment. [112](#), [116](#), *Glossar: Integrated Development Environment*

**jpeg** Joint Photographic Experts Group. [111](#), [116](#), *Glossar: Joint Photographic Experts Group*

**ms** Millisekunden. [63](#), [116](#), *Glossar: Millisekunden*

**OO** Objektorientiert. [38](#), [116](#), *Glossar: Objektorientiert*

**png** Portable Network Graphics. [111](#), [116](#), *Glossar: Portable Network Graphics*

**pt** DTP-Punkt. [38](#), [116](#), *Glossar: DTP-Punkt*

**px** Pixel. [5](#), [116](#), *Glossar: Pixel*

**RGB** Red Green Blue. [5](#), [111](#), [116](#), *Glossar: Red Green Blue*

**SRP** Single Responsibility Principle. [28](#), [116](#), *Glossar: Single Responsibility Principle*

**SSD** Solid-State-Drive. [5](#), [116](#), *Glossar: Solid-State-Drive*

**ttf** True Type Font. [42](#), [116](#), *Glossar: True Type Font*

**USB** Universal Serial Bus. [5](#), [116](#), *Glossar: Universal Serial Bus*

# Index

- äquidistant, 15
- \_\_main\_\_, 29
- SDL\_VIDEO\_WINDOW\_POS, 4
- Alpha-Kanal, 9, 14
- Animation, 65
- assoziatives Array, 111
- Bitmap, 12
  - ausgeben, 12
  - bewegen, 20
  - laden, 12
- Blasen erscheinen, 81
- Blasen zerplatzen, 88
- Blasenzahl, 84
- Blasenwachstum, 85
- Bubbles, 78
- Dictionary, 46
- Doublebuffer, 5
- Flag, 5, 100
- Font, 37
- Geschwindigkeit, 22
- Grafikprimitive, 8
- Hashtable, 111
- Hauptprogrammschleife, 5
- Kollision, 50
- Kollision anzeigen, 94
- Kollisionserkennung
  - Kreis, 50
  - Pixel, 51
  - Rechteck, 50
- Maske, 51
- Maus, 72
- Mauscursor, 87
- Mausrad, 74
- main loop, 5
- Neustart, 100
- Pause, 97
- Punktestand, 89
- Pythagoras, Satz von, 87
- Rechteck, 81
- Rendern, 38
- Richtung, 22
- Richtungswechsel, 23
- Sound, 103
- Spielende, 91
- Sprite, 24
- Standardfunktionalität, 78
- self.image, 25, 79
- self.mask, 54
- self.radius, 54, 81
- self.rect, 25, 54, 81
- Tastatur, 30
- Timer, 63, 81, 93
- Transparenz, 14, 79
- Zeitanpassungen, 93
- Zeitsteuerung, 58
- Zuordnungstabelle, 111

# Index für den Namensraum pygame

Color, 9, 11  
KEYDOWN, 32, 33  
KEYUP, 32, 33, 97  
KEY, 33  
KMOD\_LSHIFT, 33  
K\_DOWN, 32  
K\_ESCAPE, 32  
K\_LEFT, 32  
K\_RIGHT, 32  
K\_SPACE, 33  
K\_UP, 32  
MOUSEBUTTONDOWN, 74, 77  
MOUSEBUTTONUP, 74, 77, 88  
QUIT, 5, 8  
Rect, 9, 11, 20, 24, 90  
    bottomright, 20  
    bottom, 20  
    centerx, 20  
    centery, 20  
    center, 20  
    collidepoint(), 75, 77  
    contains(), 92  
    height, 20, 81  
    left, 20, 81  
    move(), 23, 24  
    move\_ip(), 26, 30  
    right, 20  
    topleft, 20  
    top, 20, 81  
    width, 20, 81  
SYSTEM\_CURSOR\_CROSSHAIR, 88  
SYSTEM\_CURSOR\_HAND, 88  
Surface, 5  
    blit(), 13, 19, 21  
    convert(), 14, 19  
    convert\_alpha(), 14, 19  
    fill(), 5, 8  
    get\_rect(), 21, 24, 54  
    set\_at(), 11, 12  
    set\_colorkey(), 14, 19, 54  
    subsurface(), 41, 43, 47, 50  
display  
    flip(), 5, 8  
    set\_caption(), 5, 8, 80  
    set\_mode(), 4, 8, 80  
draw  
    circle(), 11, 12  
    line(), 11, 12  
    lines(), 11, 12  
    polygon(), 11, 12  
    rect(), 9, 12, 93  
event  
    Event  
        unicode, 48, 50  
        button, 74, 77  
        get(), 5, 8  
        key, 32  
        mod, 33  
        pos, 74  
        type, 5, 8  
font  
    Font, 38, 43  
        render(), 39, 43  
    get\_default\_font(), 38, 43  
    get\_fonts(), 42, 43  
    match\_font(), 42, 43  
gfxdraw  
    pixel(), 11  
image, 19  
    load(), 13, 19

init(), 4, 8, 80  
key, 32  
mask  
    from\_surface(), 54, 58  
mixer  
    Sound, 103  
    play(), 103  
    init(), 4  
mouse  
    get\_pos(), 75, 77, 88  
    set\_visible(), 75, 77  
quit(), 5, 8  
sprite  
    GroupSingle, 28, 30, 80, 82, 90  
    sprite, 28, 30  
    Group, 28, 30  
        sprites(), 92  
    Sprite, 25, 30  
        kill(), 60, 65, 71, 89  
        collide\_circle(), 57, 58, 83  
        collide\_mask(), 57, 58  
        collide\_rect(), 27, 30, 57, 58  
        spritecollide(), 28, 30, 57, 58, 83  
        Group, 78  
        Sprite, 78  
time  
    Clock, 7, 8, 80  
    tick(), 7, 8, 60  
    tick\_busy\_loop(), 7, 8  
    get\_ticks(), 63, 65  
    wait(), 96  
transform  
    rotate(), 76, 77  
    scale(), 14, 19, 85