73430 Aalen
Anton-Huber-Straße 25
Hochschule Aalen

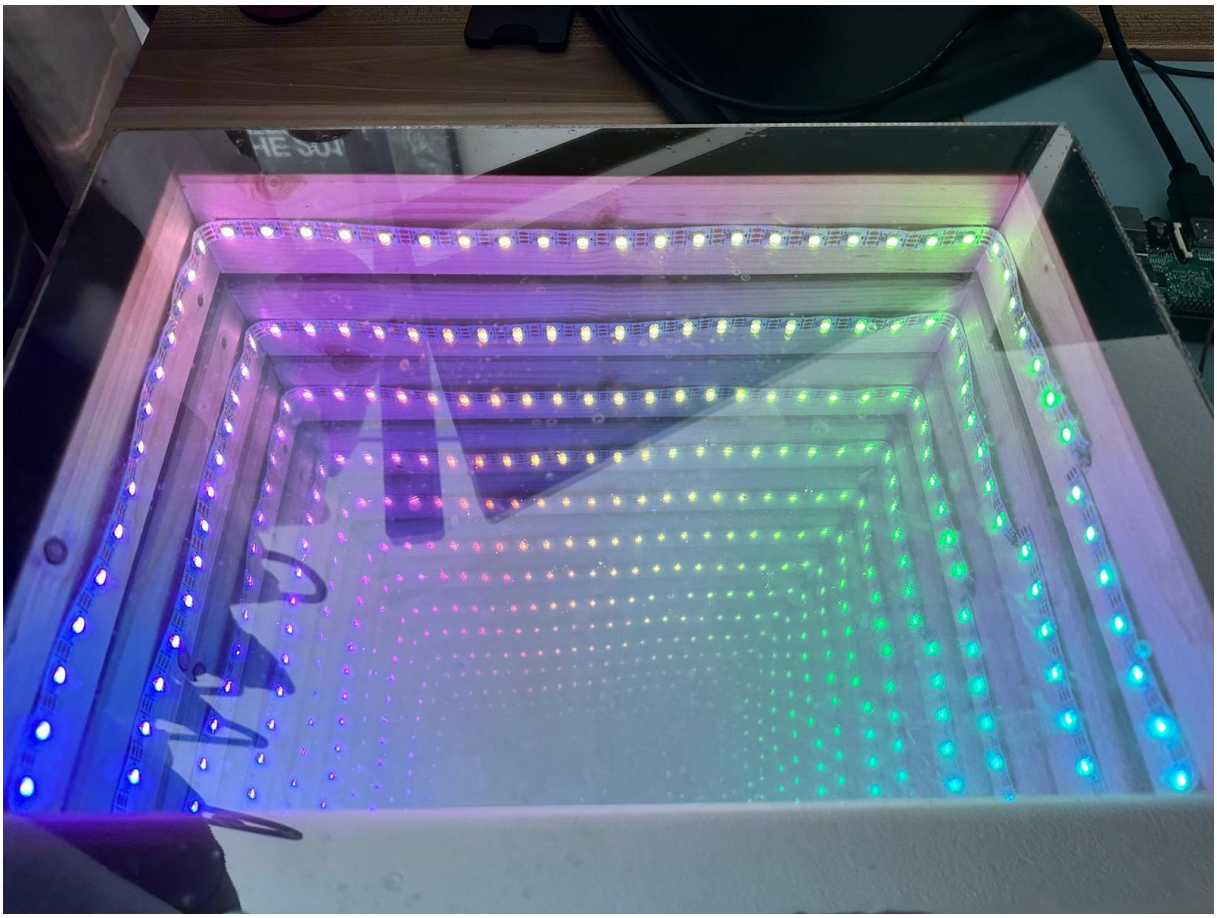
OOP-Projekt

Team:
Henrik Wörner
Leonard Eckert

Projektdokumentation

Projekt

Infinity Spiegel



73430 Aalen
Anton-Huber-Straße 25
Hochschule Aalen

Team:
Henrik Wörner
Leonard Eckert

Zusammenfassung

Infinity-Spiegel

Ein Infinity-Spiegel ist ein Spiegel, der die Illusion erzeugt in einen unendlich tiefen Raum zu schauen. Dieser Effekt wirkt sehr beeindruckend, sodass wir uns endschieden haben, ihn für unser Projekt zu nutzen. Der Spiegel wird mit steuerbaren LED-Lichtstreifen und einem Display ausgestattet, sodass wir unsere Software an diesen Geräten anwenden können. Das Display soll die Funktion besitzen, aktuelle Wetterdaten sowie den aktuellen Standort anzuzeigen. Die Wetterdaten enthalten ein Icon, den allgemeinen Wetterstatus, aktuelle Temperatur, Windrichtung und auch die Windgeschwindigkeit. Diese Daten und der aktuelle Standort sollen sich in einem definierten Zeitraum aktualisieren. Der LED-Lichtstreifen ist für den Infinity-Spiegel essenziel, da er zur Erzeugung der „Unendlichkeit Illusion“ beisteuert. Für diesen Effekt muss der Lichtstreifen das Licht zwischen den beiden Spiegeln erzeugen. Da das Licht den Effekt beeinflusst, sind sehr anschauliche „Spielereien“ mit dem Ansteuern des Lichtstreifens möglich.

Themenwahl

Das Thema wurde aufgrund der zwei unterschiedlichen Anforderungen gewählt. Der erste Aspekt betrifft die Hardwareansteuerung des LED-Lichtstreifens und der Displayausgabe. Der Zweite Aspekt, ist die softwareabhängige Abfrage der Wetterdaten. Dies schien ein perfektes Projekt für die objektorientierte Programmierung zu sein.

Problemstellung

Die Problemstellung dieses Projekts besteht darin, ein Display für den Infinity-Spiegel so zu programmieren, dass dieser periodisch Wetterdaten aktualisiert und sie auf eine ansprechende und leicht lesbare Weise darstellt. Des Weiteren muss der LED-Lichtstreifen so abgestimmt sein das der Infinity-Effekt hervorragend zur Geltung kommt. Da es mit einem Belichtungs-Modus zu monoton wird, werden hier mehrere Modi angestrebt. Zudem gibt es eine weitere, eigentlich nicht für das OOP-Projekt vorgesehene, weiter Aufgabe: Den Infinity-Spiegel zu bauen.

Lösungsweg

Erste Schritte

Als Erstes stellte sich die Frage, wo man einen Infinity-Spiegel herbekommt. Da wir schnell bemerkten, dass diese sehr teuer sind, entschlossen wir uns den Spiegel selbst zu bauen. Somit fuhren wir in den Baumarkt kauften alle benötigten Teile und bauten unseren eigenen Infinity-Spiegel.

Materialliste

Spiegel
Plexiglas
Spiegelfolie
Rahmenholz
Spiegel

Licht und Anzeige
HDMI-Display
LED-Lichterstreifen
Raspberry Pi

Bau des Spiegels

Holzrahmen

Zunächst sägten wir das Holz auf die Länge und Breite des gekauften Spiegels zurecht. Anschließend musste das Holz noch geschliffen werden, um eine glatte und gerade Oberfläche zu erzielen. Als dass gemacht war konnten wir die Holzbretter zu einem Rahmen verschrauben.

Halbdurchlässiger Spiegel

Um diesen selbst zu bauen, benötigten wir nur ein wenig Plexiglas und eine Spiegelfolie, die auf einer Seite sichtdurchlässig ist und auf der anderen nicht. Als erstes sägten wir das Plexiglas auf die Größe des gekauften Spiegels. Daraufhin musste das Plexiglas nur noch mit der Folie foliert werden. Somit war der halbdurchlässige Spiegel schon fertig.

Montage

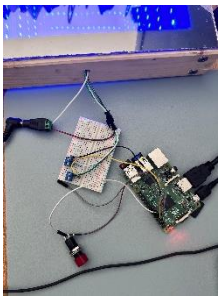
Nun mussten wir nur noch den Rahmen, den halbdurchlässigen Spiegel und den normalen Spiegel zusammenbauen. Dafür nutzten wir Montagekleber und Klebeband. Der Rahmen dient gleichzeitig auch als Abstandshalter der beiden Spiegel, sodass das Licht genug Platz hat, um sich zu reflektieren.

Implementierung

Display

Die Implementierung des Displays war sehr einfach da wir ein HDMI-Display verwendeten. Das bedeutete wir mussten einfach den Raspberry Pi und das Display via HDMI-Kabel verbinden.

LED-Lichtstreifen



Um den LED-Streifen zu betreiben, löteten wir zuerst ein Netzteil an, sodass dieser nicht über den Raspberry Pi versorgt wird, da der Raspberry Pi nur zur Ansteuerung dient. Um dies zu ermöglichen, verkabelten wir die GPIO-Pins des Raspberry Pi mit dem LED-Streifen. Zusätzlich schlossen wir einen Knopf an, um den Modus wechseln zu können.

Display

Für das Display haben wir eine Bibliothek entwickelt, die verschiedene Klassen umfasst. In diesen Klassen werden Einstellungen gespeichert und erstellt, der Standort in Koordinaten umgewandelt und Wetterdaten über eine API angefordert.

Einstellungen

Für die wichtigsten Einstellungen, die auch gespeichert werden sollen, haben wir die Klasse Settings erstellt.

```
class Settings:

    def __init__(self, username):

        self.user = username
        self.setting_file = 'setting_' + self.user + '.json'

        try:
            open(self.setting_file, 'x')
        except:
            try:
                with open(self.setting_file) as json_file:
                    data = json.load(json_file)
                len(data) == 3
            except:
                print('No or wrong locationdata found, set new one')
                self.set_location_setting()

        else:
            print('No location found, set new one')
            self.set_location_setting()
```

Bei der Initialisierung der Klasse über die init-Funktion wird das Attribut „user“ initialisiert. Anschließend wird überprüft, ob bereits eine Datei mit dem Benutzernamen und den dazugehörigen Einstellungsdaten existiert. Falls eine solche Datei nicht vorhanden ist, wird die set_location_settings-Funktion aufgerufen, die automatisch eine neue Datei mit dem Benutzernamen erstellt. In diesem Fall wird der Benutzer aufgefordert, die benötigten Daten über ein Eingabefeld anzugeben.

Für das Anlegen der neuen Daten haben wir die Funktion „set_location_settings“ erstellt. Wie oben beschreiben, frägt diese die Daten ab und speichert es in eine Json-Datei mit dem Namen des aktuellen Users.

```
def set_location_setting(self):

    country = input('Country: ')
    city = input('City: ')
    state = input('State: ')

    data = {
        'Country': country,
        'City': city,
        'State': state
    }
    setting = open(self.setting_file, 'w')
    setting.write(json.dumps(data))
    setting.close
```

Fazit Settings-Klasse:

Die Klasse ist in der Lage, Einstellungen zu speichern und dient somit als ein wichtiges Tool für viele Programme. Sie kann zudem erweitert werden, um weitere Daten zu speichern.

Standort

Für die Wetterdaten wird eine kostenlose API verwendet. Diese gibt die Daten der nächstgelegenen Wetterstation, die dem Server zur Verfügung steht, aus. Um die Wetterdaten zu erlangen, muss man eine get-request an die API schicken. Für diese get-request benötigt man jedoch noch die Breiten- und Längengrade, um den Standort festzulegen. Somit haben wir die Klasse „Location“ erstellt, um anhand den Einstellungsdaten genaue Koordinaten zu erhalten.

Diese Klasse erbt von der Settings-Klasse, da diese immer initialisiert werden muss, wenn die Location-Klasse verwendet wird. Damit dies geschieht, wird der Befehl „super().__init__()“ in der init-Funktion von der Location-Klasse verwendet. Da das Attribut „user“ auch vererbt wird muss man diesen auch mit angeben.

```
def __init__(self, username):
    super().__init__(username)
```

Um an die Koordinaten zu kommen haben wir die Funktion „get_request“ geschrieben.

```python
def get_request(self):

    location_setting = eval(open(self.setting_file, 'r').read())
    country = location_setting['Country']
    city = location_setting['City']
    state = location_setting['State']

    for i in country_codes_list:
        if i['name'] == country:
            country_code = i['code']

    self.r_location = requests.get('http://api.openweathermap.org/geo/1.0/direct?q='
                                   + city + ','+ state + ',' + country_code +
                                   '&limit=5&appid=2fb76b8383ba35703d287350c62e568b')
```

Diese verwendet auch eine API. Als erstes muss eine Request gesendet werden. Um dies zu tun, benötigt man die Daten aus der Setting-Datei. Nach aufrufen dieser Datei werden die Informationen in den Attributen gespeichert. Die Attribute werden erst hier initialisiert, da sie bei jeder neuen Anfrage aus der aktuellen Settings-Datei abgefragt werden. Würden die Attribute in der init-Funktion initialisiert, blieben sie bei Änderungen in der Settings-Datei unverändert, da die Initialisierung nur einmal beim Erstellen der Instanz erfolgen würde. Diese Vorgehensweise stellt sicher, dass Änderungen in der Settings-Datei sofort wirksam werden und die Attribute bei jeder neuen Anfrage aktualisiert werden. Dann wird noch das benötigte Kürzel für das eingegebene Land anhand der for-Schleife gesucht und auch in ein Attribut gespeichert. Nun kann die Request gesendet werden. Von der Request bekommen wir eine Json-Datei als Antwort. Diese kann ähnlich wie ein Dictionary behandelt werden.

Um die Koordinaten aus der Request zu bekommen haben wir die Funktion „get_coordinates“ geschrieben.

```
def get_coordinates(self):

    self.get_request()
    lat = self.r_location.json()[0]['lat']
    lon = self.r_location.json()[0]['lon']

    return lat, lon
```

Diese nutz die get_request-Funktion und sucht anhand dem Key „lat“ und „lon“ die passende Values. Diese stehen für die Breiten- und Längengrade.

Eine weiter Funktion namens „get_city“ gibt die nächste Stadt der Koordinaten aus.

```
def get_city(self):

    self.get_request()
    return self.r_location.json()[0]['name']
```

Fazit Location-Klasse:

Diese Klasse kann für alle möglichen Anwendungen von gebrauch sein, in denen genaue Koordinaten für einen festen Ort benötigt werden. Somit ist diese Klasse sehr universell.

Wetterdaten

Für die Wetterabfrage wurde die Klasse Weather erstellt. Bei der Initialisierung wird eine Instanz der Location-Klasse übergeben, um deren Funktionen nutzen zu können. Anschließend wird getestet, ob wir eine Request an den Server mit den Wetterdaten stellen können.

```
def __init__(self, Location):

    self.Location = Location

    try:
        self.get_request()
    except:
        raise SystemError('No connection')
```

Um eine Request an die API zu senden haben wir die Funktion „get_request“ erstellt.

```
def get_request(self):

    coordinates = self.Location.get_coordinates()


    lat, lon = coordinates
    lat = str(lat)
    lon = str(lon)

    self.r_weather = requests.get('https://api.brightsky.dev/current_weather?lat='
                     + lat + '&lon=' + lon + '&units=dwd')
```

Die Funktion frägt die Koordinaten über get_coordinates-Funktion ab. Diese werden dann in einen String konvertiert, um sie in die request.get-Funktion einfügen zu können. Die request.get-Funktion gibt eine Art Dictionary mit dem Datentyp Json aus. Diese Json Datei wird dann in einem Attribut gespeichert.

Für die Temperatur haben wir die Funktion get_temperature erstellt.

```
def get_temperature(self):

    self.get_request()

    temperature = self.r_weather.json()['weather']['temperature']

    return str(temperature) + '°C'
```

Diese Funktion führt die get_request-Funktion aus. Wandelt dann die Json-Datei in ein Dictionary um und sucht nach dem Value für den Key „temperature“. Dieser wird dann mit return zurückgegeben, davor wird er aber noch in einen String konvertiert und ein „°C“ hinzugefügt.

Für die Winddaten haben wird die Funktion get_wind geschrieben.

```
def get_wind(self):


    self.get_request()


    wind_speed = self.r_weather.json()['weather']['wind_speed_30']
    wind_direction_degrees = int(self.r_weather.json()['weather']['wind_gust_direction_30'])

    #wind direction
    if 345 <= wind_direction_degrees and wind_direction_degrees <= 15:
        wind_direction = 'Nord'
    elif 15 < wind_direction_degrees and wind_direction_degrees <= 75:
        wind_direction = 'Nord-Ost'
    elif 75 < wind_direction_degrees and wind_direction_degrees <= 115:
        wind_direction = 'Ost'
    elif 115 < wind_direction_degrees and wind_direction_degrees <= 165:
        wind_direction = 'Süd-Ost'
    elif 165 < wind_direction_degrees and wind_direction_degrees <= 195:
        wind_direction = 'Süd'
    elif 195 < wind_direction_degrees and wind_direction_degrees <= 255:
        wind_direction = 'Süd-West'
    elif 255 < wind_direction_degrees and wind_direction_degrees <= 285:
        wind_direction = 'West'
    elif 285 < wind_direction_degrees and wind_direction_degrees < 345:
        wind_direction = 'Nord-West'
    else:
        print('error')

    return str(wind_direction) + '-Wind', str(wind_speed) + 'km/h'
```

Diese Funktion führt auch die get_request-Funktion aus. Wandelt diese in ein Dictionary um und sucht nach dem Value vom Key „wind_speed_30“ und den Key „wind_gust_direction_30“. Daraus erhalten wir die Windgeschwindigkeit in km/h der letzten 30 Minuten und die Windrichtung in Grad in den letzten 30 Minuten. Damit die Windrichtung nicht in Grad angezeigt wird, sondern die ungefähre Himmelsrichtung, durchläuft die Windrichtungsinformation in Grad noch die If-Schleifen. Diese werten aus in welchem Intervall die Windrichtung liegt und speichert die passende Himmelsrichtung in der Variable wind_direction ab. Diese wird mit return zurückgegeben zusammen mit dem String „Wind“. Desweiterem wird die Windgeschwindigkeit auch mit zurückgegeben.

Für den Status wird die Funktion get_status verwendet.

```
def get_status(self):
    self.get_request()

    status =  self.r_weather.json()['weather']['icon']

    return status
```

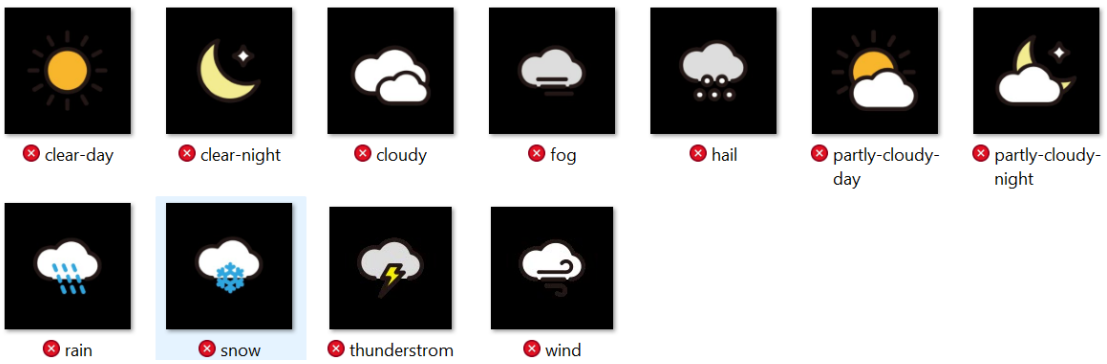
Diese Funktion führt die get_request-Funktion aus und sucht nach dem Value vom Key „Icon“. Dieses Value wird dann zurückgegeben.

Für das passende Icon würde die Funktion get_icon erstellt.

```
def get_icon(self):

    self.get_request()

    icon = self.r_weather.json()['weather']['icon']

    #icon path
    if icon == 'sleet' or icon == 'snow':
        icon_path = 'weather_icons/snow.png'
    else:
        icon_path = f'weather_icons/{icon}.png'

    return icon_path
```

Das Funktionsprinzip ist dasselbe wie bei der get_status-Funktion. Nur muss diese noch das Value in den Path einsetzen, so dass dieser übergeben werden kann. Mit öffnen der Datei über den Path wird das passende Icon mit dem Datentyp JPG aufgerufen.

Die Icons wurden in dem weather_icons-Ordner abgespeichert. Zudem wurden noch passende Dateinamen gewählt, die mit dem Status übereinstimmen, sodass es die Programmierung erleichterte.



https://www.iconfinder.com/search?q=weather

Fazit Weather-Klasse:

Die Klasse Weather kann somit für alle Anwendungen die Wetterdaten benötigt verwendet werden. Außerdem kann sie noch erweitert werden für weitere Wetterdaten, wie zum Beispiel den aktuellen Luftdruck.

Displayanzeige

Für die Displaydarstellung wurde eine bereits vorhandene Bibliothek verwendet namens Pygame. Diese ermöglicht Grafiken anzuzeigen und zu animieren. Diese wird in das Hauptprogramm importiert so wie unsere Bibliothek. Dann muss die Klasse pygame.init() und unser Weather- und Location-Klasse einmal initialisiert werden. Davor wird noch der Username abgefragt, um dies an die Location-Klasse zu übergeben.

```
import time
import pygame
from pygame.locals import *
import DisplayLib

user = input('username: ')
pygame.init()

L = DisplayLib.Location(user)
W = DisplayLib.Weather(L)
```

Danach werden die Skalierungen für das Icon festgelegt sowie den Skalierungsfaktor für die Animation. Als nächstes fragen wir den passenden Icon-Path zu den aktuellen Wetterdaten, anhand unserer get_icon-Funktion ab. Nun werden Text-Objekte eingefügt, den passenden Text laden wir über unsere Funktionen aus der Weather-Klasse. Als letztes legt man noch die Größe des Programmfensters fest, hier verwenden wir den Vollbildschirm.

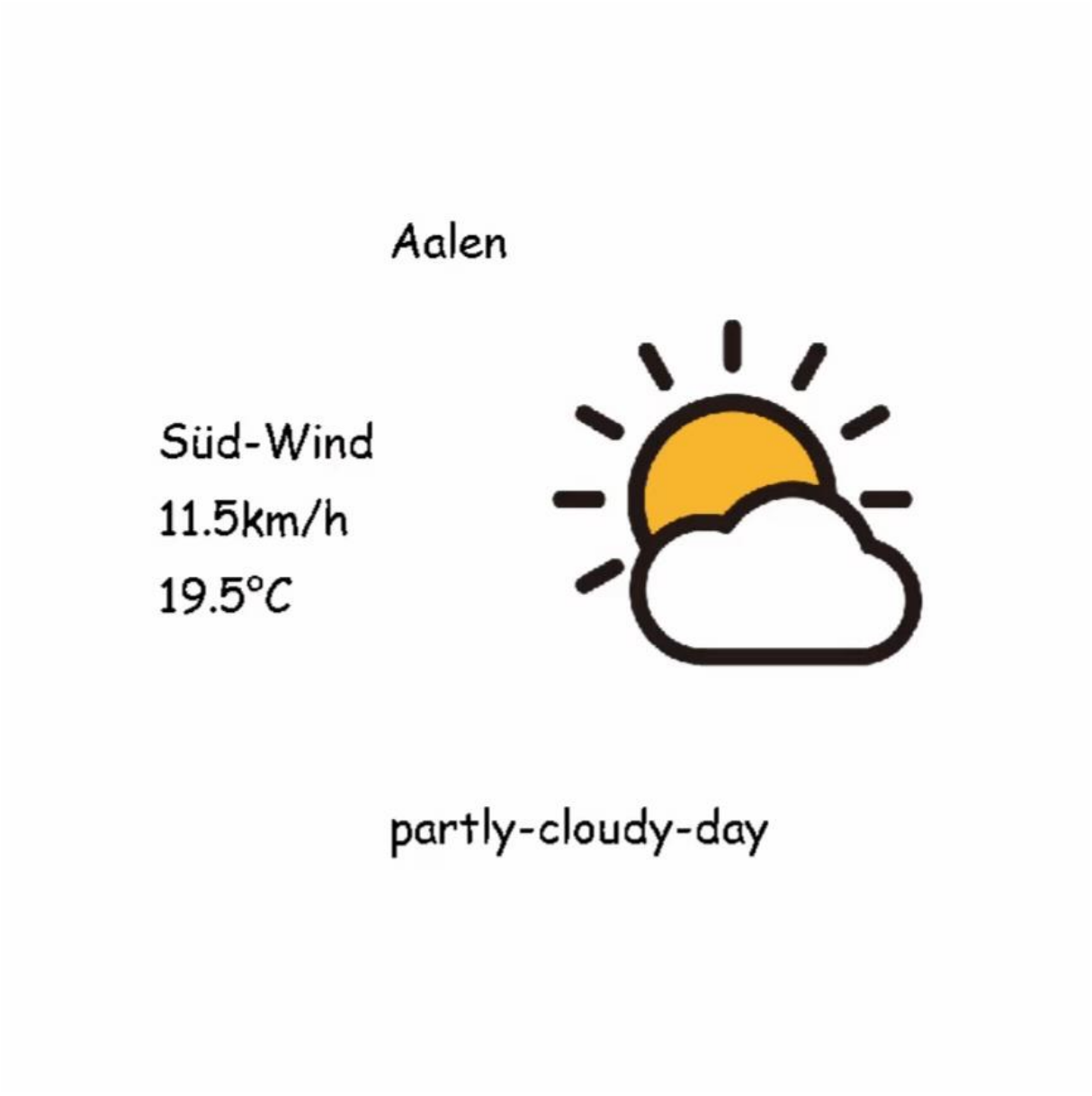
```
# Variablen/KONSTANTEN setzen
FPS  = 400
WEISS   = ( 255, 255, 255)
spielaktiv = True

#icon
skalierung      = 0
skalierungswert = 0.02

icon_path = pygame.image.load(W.get_icon())
bildgroessen = icon_path.get_rect()

#status text
my_font = pygame.font.SysFont('Comic Sans MS', 30)
status_text_surface = my_font.render(W.get_status(), False, (0, 0, 0))

#wind text
wind_direction, wind_speed = W.get_wind()
my_font = pygame.font.SysFont('Comic Sans MS', 30)
wind_direction_text_surface = my_font.render(wind_direction, False, (0, 0, 0))
wind_speed_text_surface = my_font.render(wind_speed, False, (0, 0, 0))

#temperature
temp_text_surface = my_font.render(W.get_temperature(), False, (0, 0, 0))

#City
city_text_surface = my_font.render(L.get_city(), False, (0, 0, 0))

# Definieren und Öffnen eines neuen Fensters
fenster =pygame.display.set_mode((0, 0), pygame.FULLSCREEN)
pygame.display.set_caption("Grafik skalieren")
clock = pygame.time.Clock()
```

Im Hauptprogramm gibt es eine Hauptschleife und eine Nebenschleife. Das Programm läuft eine festgelegte Zeit in der Hauptschleife, wenn diese unterbrochen wird, werden alle Daten anhand der Funktionen in der Weater-Klasse aktualisiert.

```
# Schleife Hauptprogramm
while spielaktiv:
    start = time.time()
    end = start + 10
    while end > start:
        start = time.time()
    # Überprüfen, ob Nutzer eine Aktion durchgeführt hat
        for event in pygame.event.get():
            # Beenden bei [ESC] oder [X]
            if event.type==QUIT or (event.type==KEYDOWN and event.key==K_ESCAPE):
                spielaktiv = False

        # Spiellogik
        skalierung += skalierungswert

        if skalierung > 5 or skalierung < -5:
            skalierungswert = -skalierungswert

        # Spielfeld Hintergrund
        fenster.fill(WEISS)

        # Spielfeld/figuren zeichnen
        icon_groesse = pygame.transform.scale(icon_path, (350+skalierung,350+skalierung))
        fenster.blit(icon_groesse, (150, 100))

        # Text zeichnen
        #status
        fenster.blit(status_text_surface, (30, 20))
        #wind_direction
        fenster.blit(wind_direction_text_surface, (100, 100))
```

```
    #wind_direction
    fenster.blit(wind_direction_text_surface, (100, 100))
    #wind_speed
    fenster.blit(wind_speed_text_surface, (100, 150))
    #temperature
    fenster.blit(temp_text_surface, (100, 300))
    #city
    fenster.blit(city_text_surface, (150, 400))

    # Fenster aktualisieren
    pygame.display.flip()
    clock.tick(FPS)

#Wetter daten aktualisieren
print('aktualisieren')

icon_path = pygame.image.load(W.get_icon())

status_text_surface = my_font.render(W.get_status(), False, (0, 0, 0))

wind_direction, wind_speed = W.get_wind()
wind_direction_text_surface = my_font.render(wind_direction, False, (0, 0, 0))
wind_speed_text_surface = my_font.render(wind_speed, False, (0, 0, 0))

temp_text_surface = my_font.render(W.get_temperature(), False, (0, 0, 0))

city_text_surface = my_font.render(L.get_city(), False, (0, 0, 0))
```

Ausgabe:



LED-Lichtstreifen

Für den LED-Streifen haben wir ebenfalls eine Bibliothek entwickelt, die verschiedene Klassen umfasst. In diesen Klassen werden Informationen über den LED-Streifen gespeichert und verarbeitet, Farben erzeugt und die Ansteuerung über einen Druckknopf gewährleistet.

Initialisierung

Für die Initialisierung des LED-Streifens wird die Klasse PixelStrip im Programm „*rpi_ws281x*“ verwendet. Dieses stammt aus der Bibliothek „*rpi-ws281x-python*“. Die

Klasse PixelStrip war bereits objektorientiert gestaltet und wurde von uns übernommen.

```
40  class PixelStrip:
41      def __init__(self, num, pin, freq_hz=800000, dma=10, invert=False,
42              brightness=255, channel=0, strip_type=None, gamma=None):
43          """Class to represent a SK6812/WS281x LED display.  Num should be the
44          number of pixels in the display, and pin should be the GPIO pin connected
45          to the display signal line (must be a PWM pin like 18!).  Optional
46          parameters are freq, the frequency of the display signal in hertz (default
47          800khz), dma, the DMA channel to use (default 10), invert, a boolean
48          specifying if the signal line should be inverted (default False), and
49          channel, the PWM channel to use (defaults to 0).
50          """
51
52          if gamma is None:
53              # Support gamma in place of strip_type for back-compat with
54              # previous version of forked library
55              if type(strip_type) is list and len(strip_type) == 256:
56                  gamma = strip_type
57                  strip_type = None
58              else:
59                  gamma = list(range(256))
60
61          if strip_type is None:
62              strip_type = ws.WS2811_STRIP_GRB
63
64          # Create ws2811_t structure and fill in parameters.
65          self._leds = ws.new_ws2811_t()
66
67          # Initialize the channels to zero
68          for channum in range(2):
69              chan = ws.ws2811_channel_get(self._leds, channum)
70              ws.ws2811_channel_t_count_set(chan, 0)
71              ws.ws2811_channel_t_gpionum_set(chan, 0)
72              ws.ws2811_channel_t_invert_set(chan, 0)
73              ws.ws2811_channel_t_brightness_set(chan, 0)
```

```
75          # Initialize the channel in use
76          self._channel = ws.ws2811_channel_get(self._leds, channel)
77
78          ws.ws2811_channel_t_gamma_set(self._channel, gamma)
79          ws.ws2811_channel_t_count_set(self._channel, num)
80          ws.ws2811_channel_t_gpionum_set(self._channel, pin)
81          ws.ws2811_channel_t_invert_set(self._channel, 0 if not invert else 1)
82          ws.ws2811_channel_t_brightness_set(self._channel, brightness)
83          ws.ws2811_channel_t_strip_type_set(self._channel, strip_type)
84
85          # Initialize the controller
86          ws.ws2811_t_freq_set(self._leds, freq_hz)
87          ws.ws2811_t_dmanum_set(self._leds, dma)
88
89          self.size = num
90
91          # Substitute for __del__, traps an exit condition and cleans up properly
92          atexit.register(self._cleanup)
93
94      def __getitem__(self, pos):
95          """Return the 24-bit RGB color value at the provided position or slice
96          of positions.
97          """
98          # Handle if a slice of positions are passed in by grabbing all the values
99          # and returning them in a list.
100         if isinstance(pos, slice):
101             return [ws.ws2811_led_get(self._channel, n) for n in range(*pos.indices(self.size))]
102         # Else assume the passed in value is a number to the position.
103         else:
104             return ws.ws2811_led_get(self._channel, pos)
105
106     def __setitem__(self, pos, value):
107         """Set the 24-bit RGB color value at the provided position or slice of
108         positions.
109         """
```

```
109         """
110         # Handle if a slice of positions are passed in by setting the appropriate
111         # LED data values to the provided value.
112         if isinstance(pos, slice):
113             for n in range(*pos.indices(self.size)):
114                 ws.ws2811_led_set(self._channel, n, value)
115         # Else assume the passed in value is a number to the position.
116         else:
117             return ws.ws2811_led_set(self._channel, pos, value)
118
119     def __len__(self):
120         return ws.ws2811_channel_t_count_get(self._channel)
121
122     def _cleanup(self):
123         # Clean up memory used by the library when not needed anymore.
124         if self._leds is not None:
125             ws.ws2811_fini(self._leds)
126             ws.delete_ws2811_t(self._leds)
127             self._leds = None
128             self._channel = None
129
130     def setGamma(self, gamma):
131         if type(gamma) is list and len(gamma) == 256:
132             ws.ws2811_channel_t_gamma_set(self._channel, gamma)
133
134     def begin(self):
135         """Initialize library, must be called once before other functions are
136         called.
137         """
138
139         resp = ws.ws2811_init(self._leds)
140         if resp != 0:
141             str_resp = ws.ws2811_get_return_t_str(resp)
142             raise RuntimeError('ws2811_init failed with code {0} ({1})'.format(resp, str_resp))
143
```

```python
    def show(self):
        """Update the display with the data from the LED buffer."""
        resp = ws.ws2811_render(self._leds)
        if resp != 0:
            str_resp = ws.ws2811_get_return_t_str(resp)
            raise RuntimeError('ws2811_render failed with code {0} ({1})'.format(resp, str_resp))

    def setPixelColor(self, n, color):
        """Set LED at position n to the provided 24-bit color value (in RGB order).
        """
        self[n] = color

    def setPixelColorRGB(self, n, red, green, blue, white=0):
        """Set LED at position n to the provided red, green, and blue color.
        Each color component should be a value from 0 to 255 (where 0 is the
        lowest intensity and 255 is the highest intensity).
        """
        self.setPixelColor(n, Color.RGBW(red, green, blue, white))

    def getBrightness(self):
        return ws.ws2811_channel_t_brightness_get(self._channel)

    def setBrightness(self, brightness):
        """Scale each LED in the buffer by the provided brightness.  A brightness
        of 0 is the darkest and 255 is the brightest.
        """
        ws.ws2811_channel_t_brightness_set(self._channel, brightness)

    def getPixels(self):
        """Return an object which allows access to the LED display data as if
        it were a sequence of 24-bit RGB values.
        """
        return self[:]

    def numPixels(self):
        """Return the number of pixels in the display."""
        return len(self)

```

```python
    def getPixelColor(self, n):
        """Get the 24-bit RGB color value for the LED at position n."""
        return self[n]

    def getPixelColorRGB(self, n):
        return Color.RGBW(self[n])

    def getPixelColorRGBW(self, n):
        return Color.RGBW(self[n])

# Shim for back-compatibility
class Adafruit_NeoPixel(PixelStrip):
    pass

```

Zusammenfassend dient die Klasse *PixelStrip* als Treiber für WS281x LED-Streifen in Python. Sie ermöglicht die Initialisierung und Steuerung der LEDs über einen GPIO-Pin. Parameter wie die Anzahl der LEDs, die Frequenz des Anzeigesignals und die Helligkeit sind durch die Klasse konfigurierbar. Sie definiert Methoden zum Setzen und Lesen von RGB-Farbwerten der LEDs an bestimmten Positionen, zur Steuerung der Helligkeit und zur Aktualisierung des LED-Streifens.

Farben

Die Klasse *PixelStrip* ruft bei der Steuerung der Farben die Klasse *Color* auf, die sich ebenfalls in dem Programm *rpi_ws281x* befindet.

```
6   class Color(int):
7       def __new__(cls, r, g=None, b=None, w=None):
8           if (g, b, w) == (None, None, None):
9               return int.__new__(cls, r)
10          else:
11              if w is None:
12                  w = 0
13              return int.__new__(cls, (w << 24) | (r << 16) | (g << 8) | b)
14
15      @property
16      def r(self):
17          return (self >> 16) & 0xff
18
19      @property
20      def g(self):
21          return (self >> 8) & 0xff
22
23      @property
24      def b(self):
25          return self & 0xff
26
27      @property
28      def w(self):
29          return (self >> 24) & 0xff
30
31      @staticmethod
32      def RGBW(red, green, blue, white=0):
33          """Convert the provided red, green, blue color to a 24-bit color value.
34          Each color component should be a value 0-255 where 0 is the lowest intensity
35          and 255 is the highest intensity.
36          """
37          return Color(red, green, blue, white)
38
```

In dem ursprünglichen Programm war die Methode *RGBW* eine Klasse und die Klasse *Color* eine Methode. Dies haben wir getauscht, um das Programm übersichtlicher zu machen und damit beim Aufrufen klarer wird, welche Methode genau durchlaufen wird und wo diese zu finden ist.

Die Klasse Color dient also dazu, die gewünschte Farbe, die beim Aufrufen der Animationen im RGB-Format (0, 0, 0) angegeben wird, für die Ansteuerung entsprechend auszugeben.

Animationen

Für die Animationen ist die Klasse *LEDAnimation* im Programm *led_strip_animations* zuständig. Dieses Programm stellt auch unser main-Programm dar, das aufgerufen wird, um den LED-Streifen anzusteuern. Hier werden also alle Klassen zusammengeführt und schließlich ausgeführt.

```python
##led_strip_animations.py
#to control the led strip via push button

#imports necessary for the button
from button import Button

#imports necessary for the led strip
from rpi_ws281x import PixelStrip, Color
import argparse
import time

class LEDAnimation(PixelStrip, Color):
    def __init__(self, led_count, led_pin, led_freq_hz, led_dma, led_brightness, led_invert, led_channel):
        super().__init__(led_count, led_pin, led_freq_hz, led_dma, led_invert, led_brightness, led_channel)
        self.begin()
```

Die Klasse *LEDAnimation* erbt von den Klassen *PixelStrip* und *Color*. Die Parameter, die in der Klasse *PixelStrip* verarbeitet werden, werden bei der Initialisierung eines Objekts der Klasse *LEDAnimation* gespeichert.

Darüber hinaus werden in der Klasse *LEDAnimation,* wie der Name schon sagt, die Animationen, die auf dem LED-Streifen ablaufen sollen, gespeichert. Die Animationen haben wir auch von der Bibliothek auf Github übernommen.

```python
    def colorWipe(self, color, wait_ms=50):
        """Wipe color across display a pixel at a time."""
        for i in range(self.strip.numPixels()):
            self.setPixelColor(i, color)
            self.show()
            time.sleep(wait_ms / 1000.0)

    def theaterChase(self, color, wait_ms=50, iterations=10):
        """Movie theater light style chaser animation."""
        for j in range(iterations):
            for q in range(3):
                for i in range(0, self.numPixels(), 3):
                    self.setPixelColor(i + q, color)
                self.show()
                time.sleep(wait_ms / 1000.0)
                for i in range(0, self.numPixels(), 3):
                    self.setPixelColor(i + q, 0)

    def wheel(self, pos):
        """Generate rainbow colors across 0-255 positions."""
        if pos < 85:
            return Color.RGBW(pos * 3, 255 - pos * 3, 0)
        elif pos < 170:
            pos -= 85
            return Color.RGBW(255 - pos * 3, 0, pos * 3)
        else:
            pos -= 170
            return Color.RGBW(0, pos * 3, 255 - pos * 3)

    def rainbow(self, wait_ms=20, iterations=1):
        """Draw rainbow that fades across all pixels at once."""
        for j in range(256 * iterations):
            for i in range(self.strip.numPixels()):
                self.setPixelColor(i, self.wheel((i + j) & 255))
            self.show()
            time.sleep(wait_ms / 1000.0)
```

Hier wird auch nochmal die Verarbeitung der Farben über die Klasse *Color* mit der Methode *RGBW* deutlich.

```
    def rainbowCycle(self, wait_ms=20, iterations=5):
        """Draw rainbow that uniformly distributes itself across all pixels."""
        for j in range(256 * iterations):
            for i in range(self.numPixels()):
                self.setPixelColor(i, self.wheel((int(i * 256 / self.numPixels()) + j) & 255))
            self.show()
            time.sleep(wait_ms / 1000.0)

    def theaterChaseRainbow(self, wait_ms=50):
        """Rainbow movie theater light style chaser animation."""
        for j in range(256):
            for q in range(3):
                for i in range(0, self.numPixels(), 3):
                    self.setPixelColor(i + q, self.wheel((i + j) % 255))
                self.show()
                time.sleep(wait_ms / 1000.0)
                for i in range(0, self.numPixels(), 3):
                    self.setPixelColor(i + q, 0)
```

Insgesamt gibt es 6 verschiedene Animationen.

```
if __name__ == '__main__':
    # Process arguments
    parser = argparse.ArgumentParser()
    parser.add_argument('-c', '--clear', action='store_true', help='clear the display on exit')
    args = parser.parse_args()

    # LED strip configuration:
    LED_COUNT = 69        # Number of LED pixels.
    LED_PIN = 18          # GPIO pin connected to the strip
    LED_FREQ_HZ = 800000  # LED signal frequency in hertz (usually 800khz)
    LED_DMA = 10          # DMA channel to use for generating signal (try 10)
    LED_BRIGHTNESS = 255  # Set to 0 for darkest and 255 for brightest
    LED_INVERT = False    # True to invert the signal (when using NPN transistor level shift)
    LED_CHANNEL = 0       # set to '1' for GPIOs 13, 19, 41, 45 or 53

    strip = LEDAnimation(LED_COUNT, LED_PIN, LED_FREQ_HZ, LED_DMA, LED_INVERT, LED_BRIGHTNESS, LED_CHANNEL)
    strip.begin()
```

Im main-Programm wird mit den entsprechend erwarteten Parametern ein Objekt der Klasse *LEDAnimation* erzeugt und aufgerufen.

```
93      # Animation
94      animation = 0
95
96      def change_animation():
97          global animation
98          animation = (animation + 1) % 6 # to keep animation between 0-5
99
100     # Button configuration
101     BUTTON_PIN = 17  # GPIO pin connected to the button
102     button = Button(BUTTON_PIN, callback=change_animation)
103
104
105     print('Press Ctrl-C to quit.')
106     if not args.clear:
107         print('Use "-c" argument to clear LEDs on exit')
108
109     try:
110         while True:
111             if animation == 0:
112                 strip.colorWipe(Color.RGBW(255, 0, 0))  # Red wipe
113             elif animation == 1:
114                 strip.colorWipe(Color.RGBW(0, 255, 0))  # Green wipe
115             elif animation == 2:
116                 strip.colorWipe(Color.RGBW(0, 0, 255))  # Blue wipe
117             elif animation == 3:
118                 strip.theaterChase(Color.RGBW(127, 127, 127))  # White theater chase
119             elif animation == 4:
120                 strip.rainbow()
121             elif animation == 5:
122                 strip.rainbowCycle()
123
124     except KeyboardInterrupt:
125         if args.clear:
126             strip.colorWipe(Color.RGBW(0, 0, 0), 10)
127         button.cleanup()
```

Zum Durchschalten der verschiedenen Animationen wird eine globale Variable *animation* erzeugt und durch Zeile 98 zwischen den Werten 0 und 5 gehalten. Jeder Animation wird jetzt ein Wert zwischen 0 und 5 zugeordnet. Bei Betätigung des Druckknopfs soll sich der Wert der Variable *animation* um Eins erhöhen.

Steuerung

Hier kommt dann die Klasse *Button* im Programm *button* zum Einsatz.

```python
##led_strip_animations.py
#to control the led strip via push button

#imports necessary for the button
from button import Button
```

Diese wird im Hauptprogramm *led_strip_animations* importiert.

```python
##button_init.py
#initialization and processing of signal sent by button

import RPi.GPIO as GPIO

class Button:
    def __init__(self, pin, callback=None, bouncetime=300):
        self.pin = pin
        self.callback = callback

        GPIO.setmode(GPIO.BCM)
        GPIO.setup(self.pin, GPIO.IN, pull_up_down=GPIO.PUD_UP)
        GPIO.add_event_detect(self.pin, GPIO.FALLING, callback=self._internal_callback, bouncetime=bouncetime)

    def _internal_callback(self):
        if self.callback is not None:
            self.callback()

    def cleanup(self):
        GPIO.cleanup(self.pin)
```

Das Programm dient dazu, Signale von einem Knopf zu verarbeiten. Das *RPi.GPIO*-Modul wird importiert, um die GPIO-Pins des Raspberry Pi anzusteuern. Die Klasse *Button* beinhaltet die benötigten Funktionen. Wenn die Klasse erstellt wird, wird der GPIO-Pin (pin), an dem der Knopf angeschlossen ist, festgelegt. Zudem gibt es eine Rückruffunktion (callback), die aufgerufen wird, wenn der Knopf gedrückt wird. Um Störungen zu vermeiden, wird noch eine Entprellzeit (bouncetime) definiert.

Der GPIO-Pin wird als Eingang festgelegt. Beim Drücken des Knopfs wird die interne Methode _internal_callback ausgelöst. Diese Methode ruft die Rückruffunktion auf. Die cleanup-Methode setzt den GPIO-Pin zurück, damit keine Ressourcen verschwendet werden, wenn der Knopf nicht mehr gebraucht wird.

Quellen

Bauanleitung Infinity Spiegel:

https://www.obi.de/magazin/mach-mal-mit-obi/infinity-mirror-selber-bauen

Icons:

https://www.iconfinder.com/

Display Bibliothek Pygame:

https://www.pygame.org/

LED-Lichtstreifen:

https://github.com/rpi-ws281x/rpi-ws281x-python

Knopf:

https://tutorials-raspberrypi.de/raspberry-pi-kamera-per-gpio-knopfdruck-ausloesen/