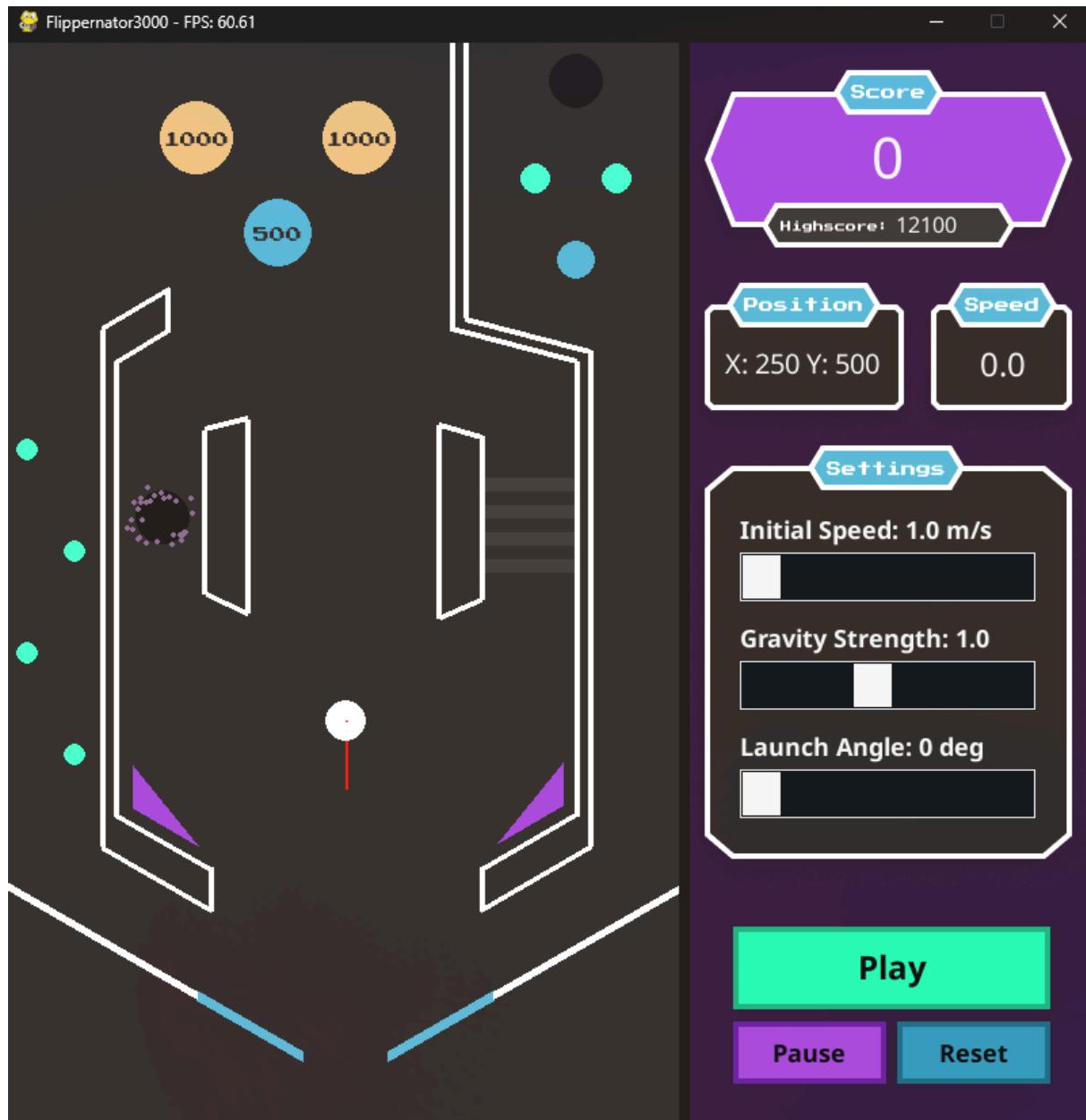


VC II // Flipper Dokumentation

Team: Flippernaut3000, Vanessa Frey, Julian Gutknecht

Finale Abgabe 3

Main Game Window mit GUI



Pause Menu



Game-Over-Screen



Ausführbare Datei

Die ausführbare Datei (.exe) finden Sie im Ordner "Flippernaut3000 Game", darin ist ein weiterer Ordner mit dem Namen "Game". In diesem Ordner sind einige Dateien zu finden, die ausführbare Datei ist die "main.exe".

Steuerung

A - Flipper Links aktivieren
D - Flipper Rechts aktivieren
R - Spiel zurücksetzen
F - Spiel einfrieren (muss ab und zu gedrückt gehalten werden)
ESC - Spiel pausieren / fortsetzen

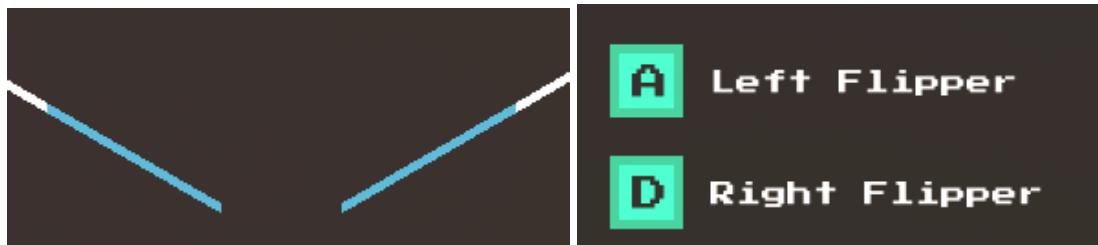
Information zum Spielen: Es muss auf dem Spielfeld eine Position der Kugel mit der Maus gewählt werden per Drag and Drop, dann lässt sich das Spiel erst starten.

Game Elements

Steuerung

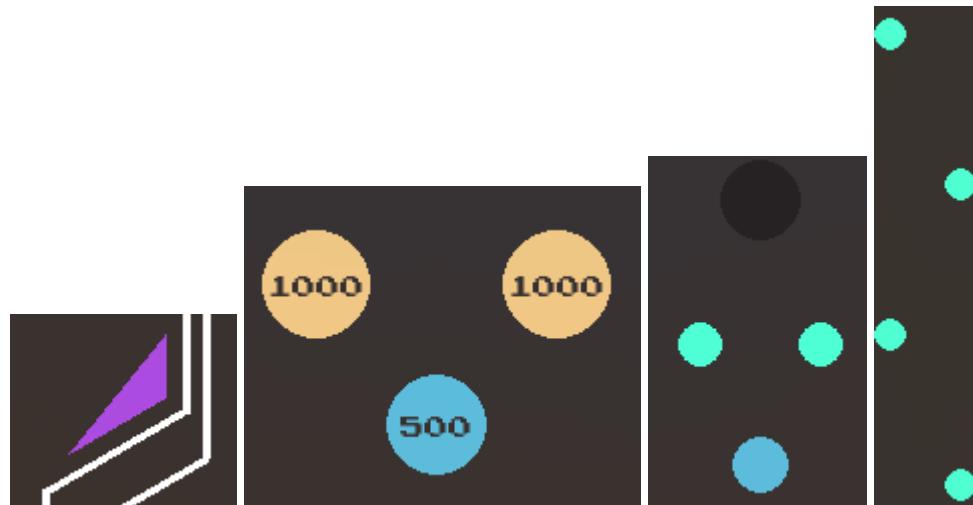


Flipper

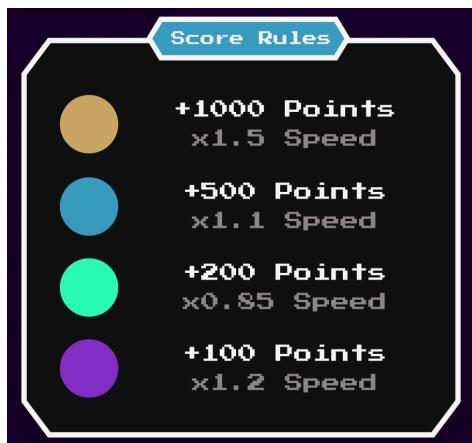


Die Flipper sind das einzige vom Spieler steuerbare Element der Pinball Machine. Sie können mit den Tasten "A" und "D" aktiviert werden. Wenn man die Tasten gedrückt hält, bleiben die Flipper aktiviert.

Bumper (rund und dreieckig)

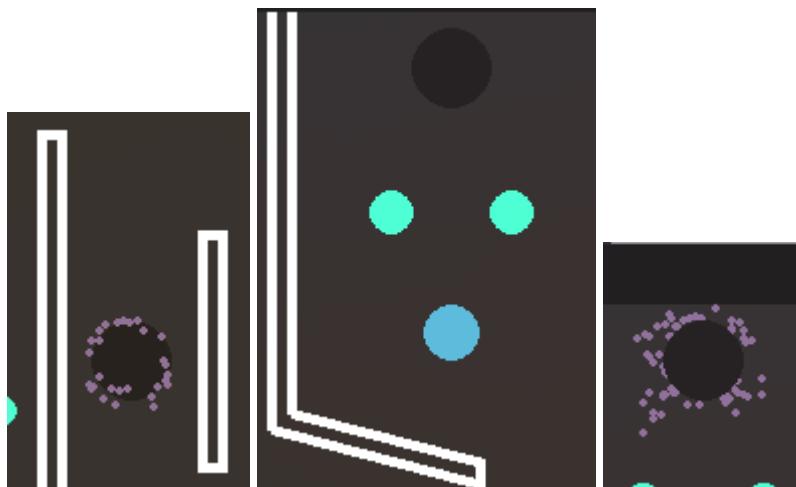


Die runden Bumper lassen die Kugel mit verschiedenen Impulsen stärker oder schwächer abprallen. Wenn die Bumper groß genug sind, zeigen sie die Punkte im Game-Window an. Diese Werte sind im Pause-Menü einzusehen:



Die Bumper nach dem Teleporter besitzen allerdings stärkere Impulse (x3). Die Bumper vergrößern sich temporär bei einer Kollision.

Teleporter



Wenn der Ball durch den Teleporter geschossen wird, teleportiert der Ball sich zu den Koordinaten des Exits. Dort verlässt die Kugel den Exit mit einem zufälligen Winkel und landet auf 3 Bumpern. Die Kugel wird außerdem für 500ms nicht gerendert, um die Teleportation zu verdeutlichen. Solange wird am Exit ein Partikeleffekt ausgeführt.

Scoring Lines



Wenn die Kugel über jede einzelne Linie rollt, erhält der Spieler Punkte. Je höher die Kugel kommt, desto mehr Punkte werden vergeben. (Linie 1 = 100, Linie 2 = 200, Linie 3 = 500, Linie 4 = 1000). Die Linien blinken kurz auf, um dem Spieler zu signalisieren, dass der Ball sie überquert hat. Punkte können nur erzielt werden, wenn die Kugel von unten nach oben geschossen wird.

Code Snippets

Bewegung der Kugel

Die `move_ball()`-Funktion aktualisiert dauerhaft die Position und Geschwindigkeit der Kugel. Dabei wirken verschiedene Einflüsse wie Gravitation, Luftwiderstand und der initiale Impuls. Außerdem wird sichergestellt, dass die Kugel für bessere Spielbarkeit nicht zu schnell werden kann.

```
● ● ●
1 # Bewegt die Kugel und aktualisiert ihre Position und Geschwindigkeit
2 def move_ball():
3     global GRAVITY, INITIAL_BALL_IMPULSE, BUMPER_BOUNCE, ball_angle, ball_angular_vel, ball_vel, ball_pos, ball_in_black_hole
4
5     if not GAME_STARTED:
6         return
7
8     # Initiale Bewegung der Kugel, wenn das Spiel beginnt
9     if ball_vel == [0, 0] and GAME_STARTED:
10         angle_rad = math.radians(BALL_ANGLE + 90)
11         ball_vel[0] = INITIAL_BALL_IMPULSE * math.cos(angle_rad)
12         ball_vel[1] = INITIAL_BALL_IMPULSE * math.sin(angle_rad)
13
14     # Überprüft Kollisionen mit dem schwarzen Loch, bevor Position und Geschwindigkeit aktualisiert werden
15     if not ball_in_black_hole:
16         check_hole_collision(ball_pos, ball_vel)
17
18     # Setzt die Geschwindigkeit zurück, wenn die Kugel im schwarzen Loch ist
19     if ball_in_black_hole:
20         ball_vel = [random.uniform(-2 * METER, 2 * METER), 1 * METER]
21         ball_angular_vel = 0
22         ball_in_black_hole = False
23
24     # Fügt die Schwerkraft zur Geschwindigkeit hinzu
25     ball_vel[1] += GRAVITY * dt
26
27     # Aktualisiert die Position der Kugel
28     ball_pos[0] += ball_vel[0] * dt + 0.5 * DAMPING_FACTOR * dt**2
29     ball_pos[1] += ball_vel[1] * dt + 0.5 * GRAVITY * DAMPING_FACTOR * dt**2
30
31     # Aktualisiert die Winkelposition der Kugel
32     ball_angle += ball_angular_vel * dt
33
34     # Wendet Dämpfung an
35     ball_vel[0] *= DAMPING_FACTOR
36     ball_vel[1] *= DAMPING_FACTOR
37     ball_angular_vel *= DAMPING_FACTOR
38
39     # Begrenze die Geschwindigkeit der Kugel
40     limit_velocity(ball_vel, MAX_VELOCITY)
41
42     # Überprüfe Kollisionen mit Bumpern
43     check_bumper_collision(ball_pos, ball_vel)
```

Kollisionserkennung und Kollisionshandling

Die `reflect_ball()`- und `check_collision()`-Funktionen sind am wichtigsten für die Kollisionserkennung der Pinball-Machine. Bei einer Kollision zwischen Kugel und Flippern, Bumpern oder Wänden wird die Geschwindigkeit der Kugel entsprechend reflektiert. Bei bestimmten Ereignissen werden zusätzlich Partikeleffekte ausgeführt.

```
● ○ ●
1 # Berechnet den Abprall der Kugel
2 def reflect_ball(start, end, is_flipper=False, flipper_angular_velocity=0, flipper_moving=False):
3     global ball_pos, ball_vel, ball_angular_vel
4
5     # Berechnet den Normalenvektor der Linie
6     normal = get_line_normal(start, end)
7     midpoint = ((start[0] + end[0]) / 2, (start[1] + end[1]) / 2)
8     ball_to_midpoint = (midpoint[0] - ball_pos[0], midpoint[1] - ball_pos[1])
9
10    # Wenn der Vektor von der Kugel zum Mittelpunkt und der Normalenvektor in die gleiche Richtung zeigen, Normalenvektor umkehren
11    if (ball_to_midpoint[0] * normal[0] + ball_to_midpoint[1] * normal[1]) > 0:
12        normal = (-normal[0], -normal[1])
13
14    # Reflektiert die Geschwindigkeit basierend auf dem Normalenvektor
15    dot_product = ball_vel[0] * normal[0] + ball_vel[1] * normal[1]
16    ball_vel[0] -= 2 * dot_product * normal[0]
17    ball_vel[1] -= 2 * dot_product * normal[1]
18
19    # Wendet den Rückprallkoeffizienten an
20    ball_vel[0] *= COEFFICIENT_OF_RESTITUTION
21    ball_vel[1] *= COEFFICIENT_OF_RESTITUTION
22
23    # Schwellenwert zur Bestimmung eines signifikanten Aufpralls
24    impact_threshold = BALL_RADIUS + 5
25
26    if is_flipper and flipper_moving:
27        # Berechnet die Entfernung vom Flipper-Drehpunkt
28        distance_from_pivot = math.sqrt((ball_pos[0] - start[0])**2 + (ball_pos[1] - start[1])**2)
29
30        # Berechnet die lineare Geschwindigkeit am Kollisionspunkt
31        linear_velocity = distance_from_pivot * flipper_angular_velocity
32
33        # Verringert den Geschwindigkeitsmultiplikator mit zunehmender Entfernung vom Drehpunkt
34        velocity_multiplier = max(0.5, 1.5 - (distance_from_pivot / FLIPPER_LENGTH))
35
36        # Wendet die Impulsübertragung vom Flipper auf die Kugel mit angepasstem Multiplikator an
37        ball_vel[0] += normal[0] * linear_velocity * velocity_multiplier
38        ball_vel[1] += normal[1] * linear_velocity * velocity_multiplier
39
40        # Fügt zusätzliche Geschwindigkeit hinzu, wenn die Kugel sehr nah am Flipper ist
41        if point_line_distance(ball_pos, start, end) <= impact_threshold:
42            extra_velocity = FLIPPER_MOTION_MOMENTUM * linear_velocity
43            ball_vel[0] += normal[0] * extra_velocity
44            ball_vel[1] += normal[1] * extra_velocity
45
46        # Fügt Flipper-Partikel hinzu, wenn es sich um einen Flipper handelt
47        if is_flipper:
48            add_flipper_particles(ball_pos)
49
50        # Wendet Drehmoment basierend auf der Kollision an
51        collision_vector = [ball_pos[0] - midpoint[0], ball_pos[1] - midpoint[1]]
52        torque = (collision_vector[0] * ball_vel[1] - collision_vector[1] * ball_vel[0]) / (BALL_RADIUS ** 2)
53        ball_angular_vel += torque
54
55        # Stellt sicher, dass die Kugel nicht zu tief in das Objekt eindringt
56        while point_line_distance(ball_pos, start, end) <= BALL_RADIUS:
57            ball_pos = [ball_pos[0] + normal[0] * 0.1, ball_pos[1] + normal[1] * 0.1]
```

Berechnung des Aufpralls

```

1 # Überprüft Kollisionen der Kugel mit den Flippern und Spielfeldgrenzen
2 def check_collision():
3     global ball_pos, ball_vel
4
5     # Überprüft Kollisionen mit den Flippern
6     for flipper_pos, angle, is_right, flipper_active, flipper_angular_velocity, flipper_moving in [
7         (left_flipper_pos, left_flipper_angle, False, left_flipper_active, FLIPPER_ANGLE_STEP if left_flipper_active else 0, left_flipper_moving),
8         (right_flipper_pos, right_flipper_angle, True, right_flipper_active, FLIPPER_ANGLE_STEP if right_flipper_active else 0, right_flipper_moving)]: 
9
10        # Berechnet die Start- und Endpunkte des Flippers basierend auf seiner Position, dem Winkel und der Ausrichtung
11        start_x, start_y = flipper_pos
12        end_x = start_x + FLIPPER_LENGTH * math.cos(math.radians(angle)) * (-1 if is_right else 1)
13        end_y = start_y - FLIPPER_LENGTH * math.sin(math.radians(angle))
14
15        flipper_start = (start_x, start_y)
16        flipper_end = (end_x, end_y)
17
18        # Überprüft kontinuierlich Kollisionen der Kugel mit dem Flippers
19        collision, collision_pos = check_continuous_collision(ball_pos, ball_vel, flipper_start, flipper_end)
20        if collision:
21            ball_pos = collision_pos # Update ball position to the collision point
22            reflect_ball(flipper_start, flipper_end, is_flipper=True, flipper_angular_velocity=flipper_angular_velocity, flipper_moving=flipper_moving)
23            break
24
25    # Überprüft Kollisionen mit Rampen/Wänden
26    for ramp in ramps:
27        ramp.check_collision(ball_pos, ball_vel)
28
29    # Überprüft Kollisionen mit den Spielfeldgrenzen
30    if ball_pos[0] <= BALL_RADIUS or ball_pos[0] >= GAME_WIDTH - BALL_RADIUS:
31        ball_vel[0] = -ball_vel[0] * COEFFICIENT_OF_RESTITUTION
32        ball_pos[0] = max(min(ball_pos[0], GAME_WIDTH - BALL_RADIUS), BALL_RADIUS)
33
34    if ball_pos[1] <= BALL_RADIUS:
35        ball_vel[1] = -ball_vel[1] * COEFFICIENT_OF_RESTITUTION
36        ball_pos[1] = BALL_RADIUS
37
38    if ball_pos[1] >= HEIGHT - BALL_RADIUS:
39        ball_vel[1] = -ball_vel[1] * COEFFICIENT_OF_RESTITUTION
40        ball_pos[1] = HEIGHT - BALL_RADIUS

```

Abfrage, ob Kollision stattgefunden hat

Es sind aber noch viele weitere Funktionen relevant für Kollisionen zwischen Kugel und Spielelementen.

Individuelle Elemente

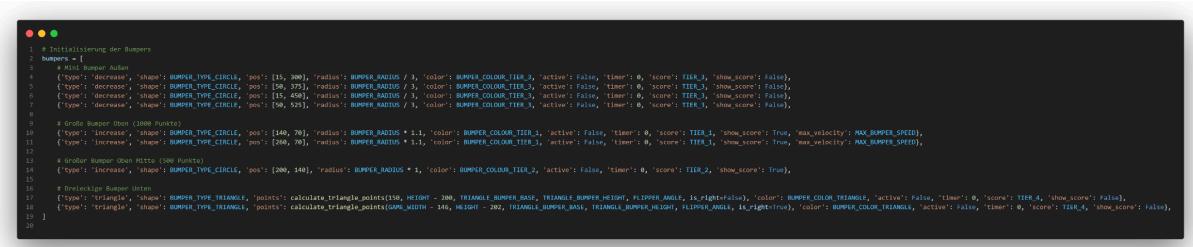
1) Bumper (aktiv)

Die Bumper gibt es in verschiedenen Formen, Größen und Farben. Zum einen gibt es runde sowie dreieckige Bumper, die mit verschiedenen Farben die Anzahl an Punkten, die sie zum Gesamtscore addieren, anzeigen. Sind die Bumper groß genug, hat dieser ein Label mit dem jeweiligen Score, der addiert wird. Die Bumper werden bei einer Kollision für einen kurzen Zeitraum visuell vergrößert, um dem Spieler klarzumachen, dass eine Kollision stattgefunden hat.



```
1 # Bumper
2 BUMPER_RADIUS = 25
3 BUMPER_BOUNCE = 1.25
4 BUMPER_SCALE = 1.25
5 BUMPER_TRIANGULAR_SCALE = 1
6 BUMPER_COLOUR = (134, 45, 198)
7 BUMPER_COLOUR_TIER_1 = (204, 169, 102)
8 BUMPER_COLOUR_TIER_2 = (56, 158, 191)
9 BUMPER_COLOUR_TIER_3 = (42, 254, 183)
10 TIER_1 = 1000
11 TIER_2 = 500
12 TIER_3 = 200
13 TIER_4 = 100
14
15 # Bumper Types
16 BUMPER_TYPE_CIRCLE = 'circle'
17 BUMPER_TYPE_TRIANGLE = 'triangle'
18
19 # Define the size for triangular bumpers
20 TRIANGLEBUMPER_BASE = 70
21 TRIANGLEBUMPER_HEIGHT = 45
22
23 # Bumper Colors
24 BUMPER_COLOR_DEFAULT = (134, 45, 198)
25 BUMPER_COLOR_VELOCITY_INCREASE = (56, 158, 191)
26 BUMPER_COLOR_VELOCITY_DECREASE = (42, 254, 183)
27 BUMPER_COLOR_TRIANGLE = (134, 45, 198)
28 MAXBUMPER_SPEED = 6 * METER
29
30 # Bumper Properties
31 BUMPER_PROPERTIES = {
32     'default': {'color': BUMPER_COLOR_DEFAULT, 'velocity_factor': 1.1},
33     'increase': {'color': BUMPER_COLOR_DEFAULT, 'velocity_factor': 1.5},
34     'teleport': {'color': BUMPER_COLOR_DEFAULT, 'velocity_factor': 3},
35     'decrease': {'color': BUMPER_COLOR_DEFAULT, 'velocity_factor': .85},
36     'triangle': {'color': BUMPER_COLOR_DEFAULT, 'velocity_factor': 1.2},
37 }
```

Kontanten der Bumper



```
1 # Initialisierung der Bumper
2 bumpers = [
3     # Rund Bumper Außen
4     {'type': 'decrease', 'shape': BUMPER_TYPE_CIRCLE, 'pos': [35, 200], 'radius': BUMPER_RADIUS / 3, 'color': BUMPER_COLOUR_TIER_3, 'active': False, 'timer': 0, 'score': TIER_3, 'show_score': False},
5     {'type': 'decrease', 'shape': BUMPER_TYPE_CIRCLE, 'pos': [150, 375], 'radius': BUMPER_RADIUS / 3, 'color': BUMPER_COLOUR_TIER_3, 'active': False, 'timer': 0, 'score': TIER_3, 'show_score': False},
6     {'type': 'decrease', 'shape': BUMPER_TYPE_CIRCLE, 'pos': [15, 450], 'radius': BUMPER_RADIUS / 3, 'color': BUMPER_COLOUR_TIER_3, 'active': False, 'timer': 0, 'score': TIER_3, 'show_score': False},
7     {'type': 'decrease', 'shape': BUMPER_TYPE_CIRCLE, 'pos': [150, 625], 'radius': BUMPER_RADIUS / 3, 'color': BUMPER_COLOUR_TIER_3, 'active': False, 'timer': 0, 'score': TIER_3, 'show_score': False},
8
9     # Große Bumper Oben (1000 Punkte)
10    {'type': 'increase', 'shape': BUMPER_TYPE_CIRCLE, 'pos': [146, 70], 'radius': BUMPER_RADIUS * 1.1, 'color': BUMPER_COLOUR_TIER_1, 'active': False, 'timer': 0, 'score': TIER_1, 'show_score': True, 'max_velocity': MAXBUMPER_SPEED},
11    {'type': 'increase', 'shape': BUMPER_TYPE_CIRCLE, 'pos': [206, 70], 'radius': BUMPER_RADIUS * 1.1, 'color': BUMPER_COLOUR_TIER_1, 'active': False, 'timer': 0, 'score': TIER_1, 'show_score': True, 'max_velocity': MAXBUMPER_SPEED},
12
13    # Großer Bumper Unten (800 Punkte)
14    {'type': 'decrease', 'shape': BUMPER_TYPE_CIRCLE, 'pos': [200, 140], 'radius': BUMPER_RADIUS * 1, 'color': BUMPER_COLOUR_TIER_2, 'active': False, 'timer': 0, 'score': TIER_2, 'show_score': True},
15
16    # Dreieckiger Bumper Unten
17    {'type': 'triangle', 'shape': BUMPER_TYPE_TRIANGLE, 'points': calculate_triangle_points(150, HEIGHT - 200, TRIANGLEBUMPER_BASE, TRIANGLEBUMPER_HEIGHT, FLIPPER_ANGLE, is_right=False), 'color': BUMPER_COLOUR_TRIANGLE, 'active': False, 'timer': 0, 'score': TIER_4, 'show_score': False},
18    {'type': 'triangle', 'shape': BUMPER_TYPE_TRIANGLE, 'points': calculate_triangle_points(GAME_WIDTH - 140, HEIGHT - 200, TRIANGLEBUMPER_BASE, TRIANGLEBUMPER_HEIGHT, FLIPPER_ANGLE, is_right=True), 'color': BUMPER_COLOUR_TRIANGLE, 'active': False, 'timer': 0, 'score': TIER_4, 'show_score': False},
19]
```

Initialisierung der Bumper

```

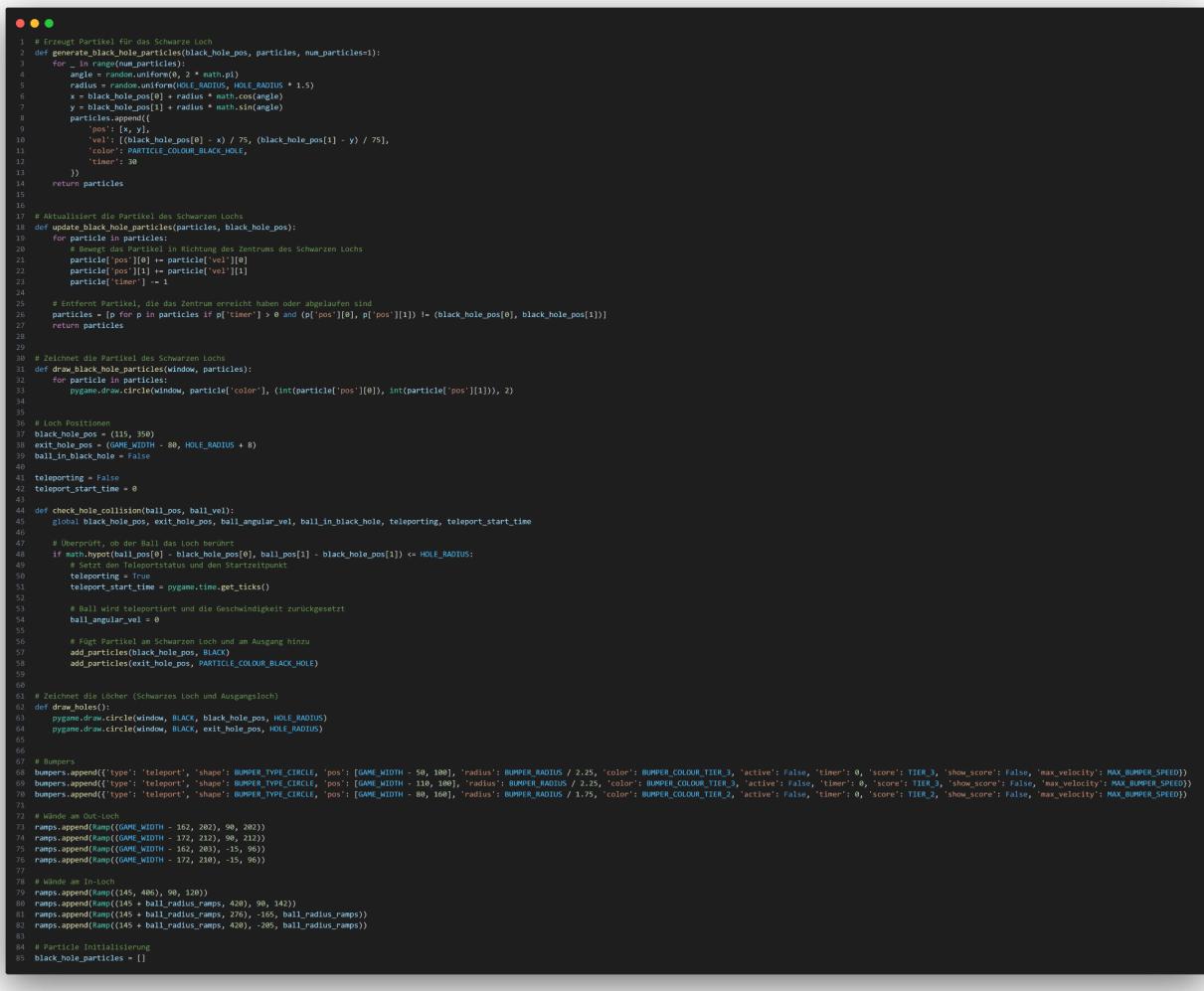
1 def reflect_ball_bumper(ball_pos, ball_vel, bumper):
2     global ball_angular_vel, score
3
4     if bumper['shape'] == BUMPER_TYPE_TRIANGLE:
5         reflect_ball_from_triangle(ball_pos, ball_vel, bumper)
6     else:
7         bumper_pos = bumper['pos']
8         angle_of_incidence = math.atan2(ball_pos[1] - bumper_pos[1], ball_pos[0] - bumper_pos[0])
9
10        # Normalenvektor und Skalarprodukt berechnen
11        normal = (math.cos(angle_of_incidence), math.sin(angle_of_incidence))
12        dot_product = ball_vel[0] * normal[0] + ball_vel[1] * normal[1]
13
14        # Geschwindigkeit reflektieren
15        ball_vel[0] -= 2 * dot_product * normal[0]
16        ball_vel[1] -= 2 * dot_product * normal[1]
17
18        # Geschwindigkeitsfaktor des jeweiligen Bumper anwenden
19        ball_vel[0] *= BUMPER_PROPERTIES[bumper['type']]['velocity_factor']
20        ball_vel[1] *= BUMPER_PROPERTIES[bumper['type']]['velocity_factor']
21
22        collision_vector = [ball_pos[0] - bumper_pos[0], ball_pos[1] - bumper_pos[1]]
23        torque = (collision_vector[0] * ball_vel[1] - collision_vector[1] * ball_vel[0]) / (BALL_RADIUS ** 2)
24        ball_angular_vel += torque
25
26        distance = math.hypot(ball_pos[0] - bumper_pos[0], ball_pos[1] - bumper_pos[1])
27        overlap = BALL_RADIUS + bumper['radius'] - distance
28        if overlap > 0:
29            ball_pos[0] += overlap * normal[0]
30            ball_pos[1] += overlap * normal[1]
31
32        add_particles(ball_pos)
33        score += bumper['score']
34
35        # Maximalgeschwindigkeit anwenden, falls angegeben
36        if 'max_velocity' in bumper:
37            limit_velocity(ball_vel, bumper['max_velocity'])
38        else:
39            limit_velocity(ball_vel, MAX_VELOCITY)
40
41        # Geschwindigkeit der Kugel begrenzen
42        limit_velocity(ball_vel, MAX_VELOCITY)
43        bumper['timer'] = 10
44
45
46    # Reflektiert die Kugelgeschwindigkeit bei Kollision mit einem dreieckigen Bumper
47    def reflect_ball_from_triangle(ball_pos, ball_vel, start, end):
48        global ball_angular_vel, score
49
50        # Normalenvektor der Linie berechnen
51        normal = get_line_normal(start, end)
52
53        # Mittelpunkt der Linie und Vektor von der Kugel zum Mittelpunkt berechnen
54        midpoint = ((start[0] + end[0]) / 2, (start[1] + end[1]) / 2)
55        ball_to_midpoint = (midpoint[0] - ball_pos[0], midpoint[1] - ball_pos[1])
56
57        if (ball_to_midpoint[0] * normal[0] + ball_to_midpoint[1] * normal[1]) > 0:
58            normal = (-normal[0], -normal[1])
59
60        # Skalarprodukt berechnen
61        dot_product = ball_vel[0] * normal[0] + ball_vel[1] * normal[1]
62
63        # Geschwindigkeit reflektieren
64        ball_vel[0] -= 2 * dot_product * normal[0]
65        ball_vel[1] -= 2 * dot_product * normal[1]
66
67        # Geschwindigkeitsfaktor des dreieckigen Bumpers anwenden
68        ball_vel[0] *= BUMPER_PROPERTIES['triangle']['velocity_factor']
69        ball_vel[1] *= BUMPER_PROPERTIES['triangle']['velocity_factor']
70
71        # Kollisionsvektor berechnen
72        collision_vector = [ball_pos[0] - midpoint[0], ball_pos[1] - midpoint[1]]
73        torque = (collision_vector[0] * ball_vel[1] - collision_vector[1] * ball_vel[0]) / (BALL_RADIUS ** 2)
74        ball_angular_vel += torque
75
76        distance = point_line_distance(ball_pos, start, end)
77        overlap = BALL_RADIUS - distance
78        if overlap > 0:
79            ball_pos[0] += overlap * normal[0]
80            ball_pos[1] += overlap * normal[1]
81
82        add_particles(ball_pos)
83        score += 100

```

Kollisionsabfrage der Bumper

2) Teleporter (aktiv)

Die Kugel wird bei einer Kollision mit dem Teleporter mit einer kurzen Verzögerung von 500 ms zum Exit-Loch teleportiert und mit einer zufälligen X-Velocity beschleunigt, damit es nicht zu repetitiven Ereignissen führt. Die Kugel fällt dann auf drei Bumper (2x 200 Punkte, 1x 500 Punkte), die jeweils einen dreifachen Geschwindigkeits-Impuls auf die Kugel auswirken. Um keine unrealistische Geschwindigkeit zu erreichen, ist ein maximaler Geschwindigkeits-Threshold definiert.



```
1 # Erzeugt Partikel für das Schwarze Loch
2 def generate_black_hole_particles(black_hole_pos, particles, num_particles=1):
3     for _ in range(num_particles):
4         angle = random.uniform(0, 2 * math.pi)
5         radius = random.uniform(HOLE_RADIUS, HOLE_RADIUS * 1.5)
6         x = black_hole_pos[0] + radius * math.cos(angle)
7         y = black_hole_pos[1] + radius * math.sin(angle)
8         particles.append({
9             'pos': [x, y],
10            'vel': [(black_hole_pos[0] - x) / 75, (black_hole_pos[1] - y) / 75],
11            'color': PARTICLE_COLOUR_BLACK_HOLE,
12            'timer': 30
13        })
14    return particles
15
16
17 # Aktualisiert die Partikel des Schwarzen Lochs
18 def update_black_hole_particles(particles, black_hole_pos):
19     for particle in particles:
20         # Bewegt das Partikel in Richtung des Zentrums des Schwarzen Lochs
21         particle['pos'][0] += particle['vel'][0]
22         particle['pos'][1] += particle['vel'][1]
23         particle['timer'] -= 1
24
25     # Entfernt Partikel, die das Zentrum erreicht haben oder abgelaufen sind
26     particles = [p for p in particles if p['timer'] > 0 and (p['pos'][0], p['pos'][1]) != (black_hole_pos[0], black_hole_pos[1])]
27     return particles
28
29
30 # Zeichnet die Partikel des Schwarzen Lochs
31 def draw_black_hole_particles(window, particles):
32     for particle in particles:
33         pygame.draw.circle(window, particle['color'], (int(particle['pos'][0]), int(particle['pos'][1])), 2)
34
35
36 # Loch Positionen
37 black_hole_pos = (115, 350)
38 exit_hole_pos = (GAME_WIDTH - 80, HOLE_RADIUS + 8)
39 ball_in_black_hole = False
40
41 teleporting = False
42 teleport_start_time = 0
43
44 def check_hole_collision(ball_pos, ball_vel):
45     global black_hole_pos, exit_hole_pos, ball_angular_vel, ball_in_black_hole, teleporting, teleport_start_time
46
47     # Überprüft, ob der Ball das Loch berührt
48     if math.hypot(ball_pos[0] - black_hole_pos[0], ball_pos[1] - black_hole_pos[1]) <= HOLE_RADIUS:
49         # Setzt den Teleportstatus und den Startzeitpunkt
50         teleporting = True
51         teleport_start_time = pygame.time.get_ticks()
52
53     # Ball wird teleportiert und die Geschwindigkeit zurückgesetzt
54     ball_angular_vel = 0
55
56     # Fügt Partikel am Schwarzen Loch und am Ausgang hinzu
57     add_particles(black_hole_pos, BLACK)
58     add_particles(exit_hole_pos, PARTICLE_COLOUR_BLACK_HOLE)
59
60
61 # Zeichnet die Löcher (Schwarzes Loch und Ausgangsloch)
62 def draw_holes():
63     pygame.draw.circle(window, BLACK, black_hole_pos, HOLE_RADIUS)
64     pygame.draw.circle(window, BLACK, exit_hole_pos, HOLE_RADIUS)
65
66
67 # Bumper
68 bumpers.append({'type': 'teleport', 'shape': BUMPER_TYPE_CIRCLE, 'pos': [GAME_WIDTH - 50, 300], 'radius': BUMPER_RADIUS / 2.25, 'color': BUMPER_COLOUR_TIER_3, 'active': False, 'timer': 0, 'score': TIER_3, 'show_score': False, 'max_velocity': MAX_BUMPER_SPEED})
69 bumpers.append({'type': 'teleport', 'shape': BUMPER_TYPE_CIRCLE, 'pos': [GAME_WIDTH - 110, 300], 'radius': BUMPER_RADIUS / 2.25, 'color': BUMPER_COLOUR_TIER_3, 'active': False, 'timer': 0, 'score': TIER_3, 'show_score': False, 'max_velocity': MAX_BUMPER_SPEED})
70 bumpers.append({'type': 'teleport', 'shape': BUMPER_TYPE_CIRCLE, 'pos': [GAME_WIDTH - 80, 300], 'radius': BUMPER_RADIUS / 2.25, 'color': BUMPER_COLOUR_TIER_3, 'active': False, 'timer': 0, 'score': TIER_3, 'show_score': False, 'max_velocity': MAX_BUMPER_SPEED})
71
72 # Wände am Loch
73 ramps.append(Ramp((GAME_WIDTH - 162, 202), 90, 202))
74 ramps.append(Ramp((GAME_WIDTH - 172, 212), 90, 212))
75 ramps.append(Ramp((GAME_WIDTH - 162, 203), -15, 96))
76 ramps.append(Ramp((GAME_WIDTH - 172, 210), -15, 96))
77
78 # Wände am In-Loch
79 ramps.append(Ramp((145, 406), 90, 120))
80 ramps.append(Ramp((145 + ball_radius_ramps, 420), 90, 142))
81 ramps.append(Ramp((145 + ball_radius_ramps, 276), -105, ball_radius_ramps))
82 ramps.append(Ramp((145 + ball_radius_ramps, 420), -105, ball_radius_ramps))
83
84 # Particle Initialisierung
85 black_hole_particles = []
```

Teleporter Funktionalitäten

3) Scoring Lines (aktiv)

Die Scoring Lines sind vier Linien, die mit gleichem Abstand auf dem Spielfeld zwischen zwei Wänden platziert sind. Wenn die Kugel über eine Linie geschossen wird, blinkt die Linie kurz auf, um dem Spieler zu signalisieren, dass eine Kollision stattgefunden hat. Die Kollision hat keine Auswirkung auf die Kugel. Je höher die Linie ist, desto mehr Punkte gibt es. Wenn alle Linien überquert wurden, kann es bis zu 1600 Punkte geben (100 + 200 + 500 + 1000). Punkte gibt es nur, wenn die Kugel die Linie von unten nach oben überquert.

```
1 # Definitionen der Scoring Lines
2 scoring_lines = [
3     {'start': (GAME_WIDTH - 150, 325), 'end': (GAME_WIDTH - 80, 325), 'idle_color': LIGHT_GREY, 'flash_color': BUMPER_COLOUR_TIER_1, 'score': 400},
4     {'start': (GAME_WIDTH - 150, 345), 'end': (GAME_WIDTH - 80, 345), 'idle_color': LIGHT_GREY, 'flash_color': BUMPER_COLOUR_TIER_2, 'score': 300},
5     {'start': (GAME_WIDTH - 150, 365), 'end': (GAME_WIDTH - 80, 365), 'idle_color': LIGHT_GREY, 'flash_color': BUMPER_COLOUR_TIER_3, 'score': 200},
6     {'start': (GAME_WIDTH - 150, 385), 'end': (GAME_WIDTH - 80, 385), 'idle_color': LIGHT_GREY, 'flash_color': BUMPER_COLOUR_TRIANGLE, 'score': 100},
7 ]
8
9 # Positionen der Wände
10 wall_left_start = (GAME_WIDTH - 150, 410)
11 wall_right_start = (GAME_WIDTH - 80, 510)
12
13
14 ramps.append(Ramp(wall_right_start, 90, 275))
15 ramps.append(Ramp((wall_right_start[0] + 10, wall_right_start[1]), 90, 283))
16
17 # Walls Innen
18 ramps.append(Ramp(wall_left_start, 90, 120))
19 ramps.append(Ramp((wall_left_start[0] - ball_radius_ramps, wall_left_start[1] + 14), 90, 142))
20 ramps.append(Ramp((GAME_WIDTH - 150, 290), 165, ball_radius_ramps))
21 ramps.append(Ramp((GAME_WIDTH - 150, 410), 205, ball_radius_ramps))
22
23 crossed_lines = {i: {'crossed': False, 'last_flash_time': 0} for i in range(len(scoring_lines))}
24
25
26 # Überprüft, ob die Kugel eine Linie überquert hat
27 def check_line_crossing(ball_pos, prev_ball_pos, line_start, line_end):
28     # Hilfsfunktion zur Bestimmung, auf welcher Seite der Linie ein Punkt liegt
29     def side_of_line(point, line_start, line_end):
30         return (line_end[0] - line_start[0]) * (point[1] - line_start[1]) - (line_end[1] - line_start[1]) * (point[0] - line_start[0])
31
32     # Bestimmt, auf welcher Seite der Linie die Kugel sich vorher befand
33     side_prev = side_of_line(prev_ball_pos, line_start, line_end)
34     # Bestimmt, auf welcher Seite der Linie die Kugel sich aktuell befindet
35     side_current = side_of_line(ball_pos, line_start, line_end)
36
37     # Überprüft, ob die Kugel sich nach oben bewegt
38     if ball_pos[1] < prev_ball_pos[1]:
39         # Überprüft, ob die Kugel die Linie überquert hat
40         if side_prev * side_current < 0:
41             # Überprüft, ob die Kugel sich innerhalb der x-Koordinaten der Linie befindet
42             if min(line_start[0], line_end[0]) <= ball_pos[0] <= max(line_start[0], line_end[0]):
43                 return True
44
45     return False
46
47 # Überprüft, ob die Kugel eine der Scoring-Linien überquert hat
48 def check_scoring_lines(ball_pos, prev_ball_pos):
49     global score
50
51     # Schleife durch alle Scoring-Linien
52     for i, line in enumerate(scoring_lines):
53         # Überprüft, ob die Kugel die aktuelle Linie überquert hat
54         if check_line_crossing(ball_pos, prev_ball_pos, line['start'], line['end']):
55             # Überprüft, ob die Linie zuvor nicht überquert wurde
56             if not crossed_lines[i]['crossed']:
57                 # Erhöht den Highscore um die Punktzahl der Linie
58                 score += line['score']
59                 # Markiert die Linie als überquert und speichert die Zeit des Überquerens
60                 crossed_lines[i] = {'crossed': True, 'last_flash_time': pygame.time.get_ticks()}
61
62
63     # Zeichnet die Scoring-Linien auf das Spielfeld
64     def draw_scoring_lines():
65         current_time = pygame.time.get_ticks()
66
67         # Schleife durch alle Scoring-Linien
68         for i, line in enumerate(scoring_lines):
69             color = line['idle_color']
70             # Überprüft, ob die Linie überquert wurde
71             if crossed_lines[i]['crossed']:
72                 # Überprüft, ob die Linie innerhalb der Blinkzeit überquert wurde
73                 if current_time - crossed_lines[i]['last_flash_time'] <= FLASHING_TIME:
74                     color = line['flash_color']
75                 else:
76                     crossed_lines[i]['crossed'] = False
77
78         # Zeichnet die Linie auf das Spielfeld
79         pygame.draw.line(window, color, line['start'], line['end'], 10)
```

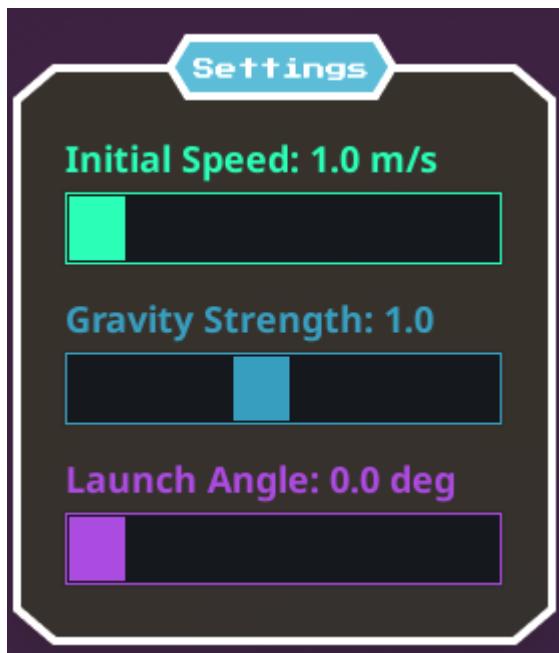
Scoring Lines Funktionalitäten

4) Seitenwände (passiv)

Die Seitenwände sind ein passives Element, an dem die Kugel nur abprallt.

```
● ● ●  
1 # Überprüft Kollisionen mit den Spielfeldgrenzen  
2 if ball_pos[0] <= BALL_RADIUS or ball_pos[0] >= GAME_WIDTH - BALL_RADIUS:  
3     ball_vel[0] = -ball_vel[0] * COEFFICIENT_OF_RESTITUTION  
4     ball_pos[0] = max(min(ball_pos[0], GAME_WIDTH - BALL_RADIUS), BALL_RADIUS)  
5  
6 if ball_pos[1] <= BALL_RADIUS:  
7     ball_vel[1] = -ball_vel[1] * COEFFICIENT_OF_RESTITUTION  
8     ball_pos[1] = BALL_RADIUS  
9  
10 if ball_pos[1] >= HEIGHT - BALL_RADIUS:  
11     ball_vel[1] = -ball_vel[1] * COEFFICIENT_OF_RESTITUTION  
12     ball_pos[1] = HEIGHT - BALL_RADIUS
```

Einstellbare Parameter



Vor dem Spiel sind folgende Parameter, die Auswirkungen auf das Spiel haben, einstellbar:

- **Initial Speed:** Legt die Anfangsgeschwindigkeit fest (1,0 – 10,0 m/s)
- **Gravity Strength:** Legt die Gravitationsstärke fest (0,5 – 2 m/s²)
- **Launch Angle:** Der Winkel, mit dem die Kugel gestartet wird (0 – 359°)

Platzieren der Kugel:

Die Kugel lässt sich per Drag and Drop beliebig auf dem Spielfeld platzieren. Somit kann eine individuelle Startposition gewählt werden.

Partikeleffekte

Die Partikeleffekte verbessern das Spielerlebnis und verdeutlichen Kollisionen zwischen Kugel und einigen Spielelementen.

```
● ○ ●
1 particles = []
2
3 # Fügt Partikel an einer gegebenen Position hinzu
4 def add_particles(pos, color=None):
5     for _ in range(20):
6         particles.append({
7             # Position des Partikels
8             'pos': list(pos),
9             # Zufällige Geschwindigkeit des Partikels
10            'vel': [random.uniform(-2, 2), random.uniform(-2, 2)],
11            # Lebensdauer des Partikels
12            'timer': random.randint(10, 20),
13            # Zufällige Farbe des Partikels
14            'color': color if color else random.choice([RED, GREEN, BLUE, PURPLE, CYAN, WHITE])
15        })
16
17 # Fügt spezielle Partikel für die Flipper hinzu
18 def add_flipper_particles(pos):
19     flipper_color = (42, 254, 183)
20     add_particles(pos, flipper_color)
21
22 # Aktualisiert die Position und den Timer der Partikel
23 def update_particles():
24     global particles
25     for particle in particles:
26         # Aktualisiert die x- und y-Positionen des Partikels
27         particle['pos'][0] += particle['vel'][0]
28         particle['pos'][1] += particle['vel'][1]
29         particle['timer'] -= 1
30     # Entfernt Partikel, deren Timer abgelaufen ist
31     particles = [p for p in particles if p['timer'] > 0]
32
33
34 # Zeichnet alle aktiven Partikel auf das Fenster
35 def draw_particles():
36     for particle in particles:
37         pygame.draw.circle(window, particle['color'], (int(particle['pos'][0]), int(particle['pos'][1])), 2)
```

Partikeleffekte

Herausforderungen

Realistische Flipper

Die größte Herausforderung beim Programmieren der Pinball-Maschine war die Umsetzung realistischer Flipper. Einige Trial-and-Error-Versuche waren notwendig, um ein zufriedenstellendes Ergebnis zu erhalten. Im Endeffekt haben wir die Flipper so umgesetzt, dass die Flipper zusätzliche Geschwindigkeit auf die Kugel addieren, während sie sich in der Flipper-Animation befinden. Zusätzlich wird überprüft, wie weit die Kugel vom Pivot-Punkt entfernt ist. Ist die Kugel am äußersten Punkt des Flippers (am weitesten entfernt vom Pivot), wird mehr Geschwindigkeit addiert. Dies ermöglicht es, die Kugel am Flipper zu halten, sie auf dem Flipper herunterrollen zu lassen und am äußersten Punkt "wegzuschießen".

```
● ● ●
1 if is_flipper and flipper_moving:
2     # Berechnet die Entfernung vom Flipper-Drehpunkt
3     distance_from_pivot = math.sqrt((ball_pos[0] - start[0])**2 + (ball_pos[1] - start[1])**2)
4
5     # Berechnet die lineare Geschwindigkeit am Kollisionspunkt
6     linear_velocity = distance_from_pivot * flipper_angular_velocity
7
8     # Verringert den Geschwindigkeitsmultiplikator mit zunehmender Entfernung vom Drehpunkt
9     velocity_multiplier = max(0.5, 1.5 - (distance_from_pivot / FLIPPER_LENGTH))
10
11    # Wendet die Impulsübertragung vom Flipper auf die Kugel mit angepasstem Multiplikator an
12    ball_vel[0] += normal[0] * linear_velocity * velocity_multiplier
13    ball_vel[1] += normal[1] * linear_velocity * velocity_multiplier
14
15    # Fügt zusätzliche Geschwindigkeit hinzu, wenn die Kugel sehr nah am Flipper ist
16    if point_line_distance(ball_pos, start, end) <= impact_threshold:
17        extra_velocity = FLIPPER_MOTION_MOMENTUM * linear_velocity
18        ball_vel[0] += normal[0] * extra_velocity
19        ball_vel[1] += normal[1] * extra_velocity
```

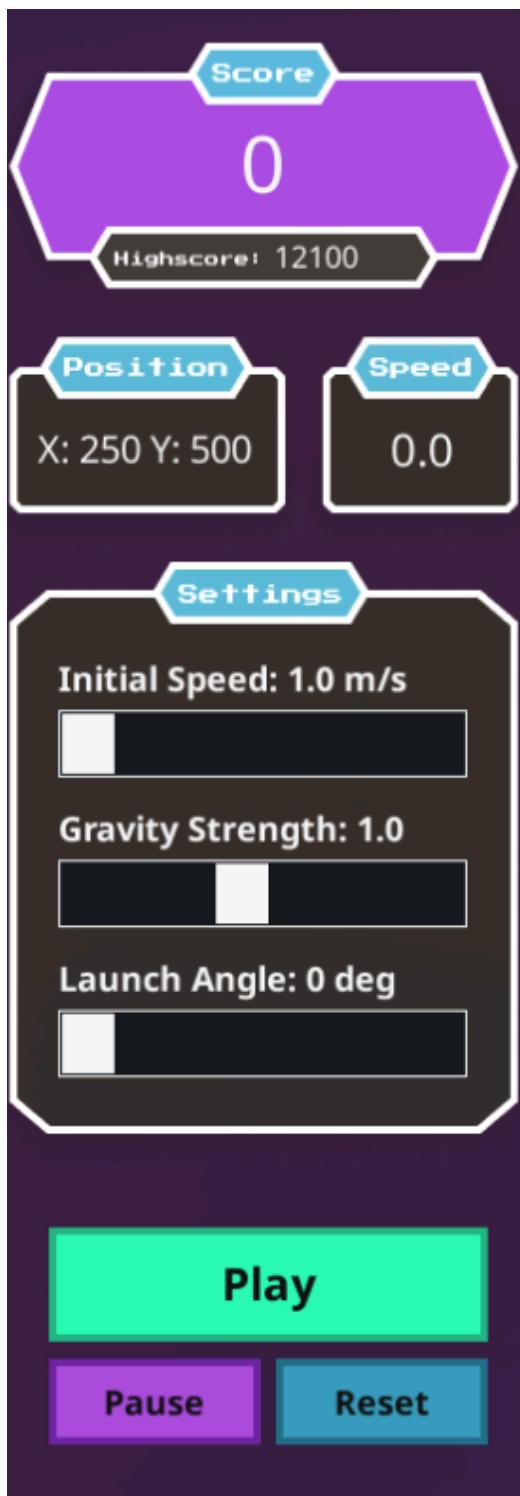
Wand Kollisionen

Da wir die Wände mit einzelnen Linien zeichnen, kommt es an einigen Eckpunkten zu Komplikationen mit der Kollisionsberechnung. Deshalb kann es in manchen Edge Cases zu fehlerhaften Kollisionen führen. Wir können die Fehler minimieren, indem wir die Wände, mit denen die Kugel kollidieren kann, mindestens so lang machen wie der Durchmesser der Kugel und die Dicke der Wände nur 2 Pixel beträgt.

```
● ○ ●
1 ball_radius_ramps = BALL_RADIUS * 2 + 2
2
3 # Initialisierung der Rampen im Spielfeld
4 ramps = [
5     # Flipper Rampen
6     Ramp(ramp_left_end, left_ramp_angle, RAMP_LENGTH, 6),
7     Ramp(ramp_right_end, right_ramp_angle, RAMP_LENGTH, 6),
8
9     # Spielfeld Rampen
10    # Schräge Rampen unter den dreieckigen Bumpern innen
11    Ramp((150, 609), left_ramp_angle, 80),
12    Ramp((GAME_WIDTH - 150, 609), right_ramp_angle, 80),
13
14    # Äußere Wände
15    Ramp((80, 570), 90, 335),
16    Ramp((GAME_WIDTH - 80, 570), 90, 100),
17
18    # Schräge Rampen unter den dreieckigen Bumpern außen
19    Ramp((150, 639), left_ramp_angle, 92),
20    Ramp((GAME_WIDTH - 150, 639), right_ramp_angle, 94),
21
22    # Innere Wände
23    Ramp((70, 592), 90, 380),
24    Ramp((GAME_WIDTH - 70, 592), 90, 105),
25
26    Ramp((70, 210), right_ramp_angle, 57),
27    Ramp((80, 235), right_ramp_angle, 45),
28    Ramp((118, 213), 90, ball_radius_ramps),
29
30    Ramp((GAME_WIDTH - 150, 640), 90, ball_radius_ramps),
31    Ramp((150, 640), 90, ball_radius_ramps),
32
33    # weitere Wände werden im weiteren Verlauf mit Ramps.append() hinzugefügt
34 ]
```

Initialisierung der Wände und Rampen

Graphical User Interface



Der aktuelle Score wird am größten dargestellt mit einem farbigen Hintergrund. Somit fällt es dem Spieler leichter, seinen Score während des spielen unkompliziert einzusehen. Darunter wird der globale Highscore angezeigt, um sich mit dem Bestwert zu vergleichen.

Die Position und Geschwindigkeit der Kugel wird darunter angezeigt. Dabei werden für bessere Lesbarkeit auf lange Nachkommazahlen verzichtet.

Die Einstellungen haben ein eigenes Fenster, mit sinnvollen Schiebereglern, die gewisse Einflüsse vor dem Spiel beeinflussen.

Mit dem Play-Button kann das Spiel gestartet werden.

Mit dem Pause-Button wird das Spiel pausiert und das Pause-Menü öffnet sich. Optional kann die ESC-Taste dafür verwendet werden.

Mit dem Reset-Button wird das Spiel zurückgesetzt und es kann neu gestartet werden.

Die GUI ist am rechten Bildschirmrand des Game-Windows platziert. Sie ist im Verhältnis zum Spielfeld deutlich kleiner und nimmt daher nicht zu viel Aufmerksamkeit auf sich. Der Score ist farblich hinterlegt, damit man einfacher

seinen aktuellen Punktestand wahrnehmen kann. Die Buttons sind ebenfalls eindeutig erkennbar, wie bei traditionellen Arcade-Maschinen.

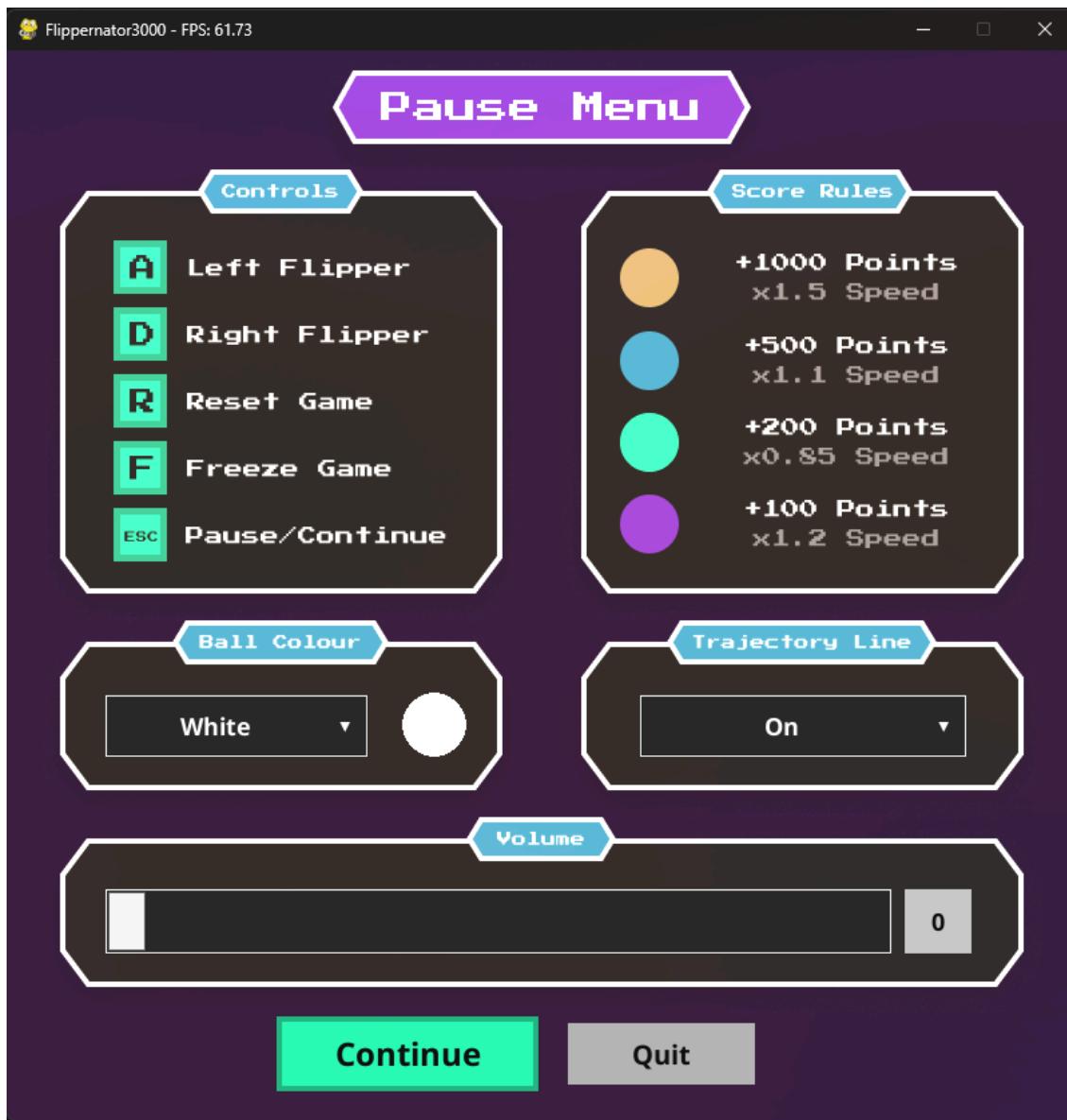


Vgl. SEGA Arcade Maschine

(Quelle:

https://coinoppartsetc.com/sites/default/files/SEGA%20Arcade%20Game%20VIEW%20CHANGE%20START%20BUTTON%20CONTROL%20PANEL%20ASSEMBLY%20%237961_2.jpg)

Pause-Menü



Im Pause-Menü gibt es einige Informationen hinsichtlich Steuerung und Spielregeln. Diese sind deutlich im oberen Bereich erkennbar. Darunter befinden sich zwei Einstellungen, die das Spielerlebnis beeinflussen:

- **Ball Colour:** Hier kann die Farbe der Kugel individuell angepasst werden
- **Trajectory Line:** Die Trajectory Line (Linie, die den Weg der Kugel hervorsagt) kann hier aktiviert/deaktiviert werden.
- **Volume:** Hier lässt sich anhand eines Schiebereglers die Lautstärke der Musik anpassen. In einem Textfeld daneben ist die Lautstärke auslesbar.

Mit dem Continue-Button oder der ESC-Taste kann das Pause-Menü wieder geschlossen werden. Während das Menü geöffnet ist, ist das Spiel im Hintergrund pausiert.

```

1  ##
2  ## Pause Menu ##
3  ##
4
5  # Erstellt ein Input-Fenster mit den Spielsteuerungen an
6  def pause_menu():
7      global is_pause_menu_open, pause_panel, continue_button, quit_button, volume_slider, volume_label, volume_value_label
8      is_pause_menu_open = True
9
10     # Kündigt die GUI Elemente des Spiels an
11     set_gui_visibility(False)
12
13     padding = 48
14
15     # Erstellt eine Oberfläche für das Pause-Menü
16     pause_surface = pygame.Surface((WIDTH, HEIGHT))
17     pause_surface.blit(pause_image, (0, 0))
18
19
20     # erstellt ein Panel, das das gesamte Fenster abdeckt
21     pause_panel = UIPanel(
22         relative_rect=pygame.Rect(0, 0, WIDTH, HEIGHT),
23         manager=manager,
24     )
25
26
27     #dropdown_menu (für die Farbauswahl der Kugel)
28     ball_preview_width = 48
29     dropdown_width = WIDTH / 2 - padding + padding - 56 - ball_preview_width
30     volume_width = WIDTH / padding + padding - 64 - ball_preview_width
31
32     dropdown = UDDropDownMenu(
33         options_list=['White', 'Red', 'Green', 'Blue', 'Purple', 'Orange', 'Yellow'],
34         starting_option=ball_color,
35         relative_rect=pygame.Rect(padding + 24, 480, dropdown_width, 60),
36         manager=manager,
37         container=pause_panel,
38         object_id=ObjectID(class_id='dropdown', object_id='Ball dropdown')
39     )
40
41     # Funktion zur Aktualisierung des Vorschau des Kugel
42     def update_ball_preview(color):
43         global ball_color
44         ball_color = color
45
46         # Neue Oberfläche für die Kugelverschmelzung erstellen
47         ball_preview_surface = pygame.Surface((ball_preview_width, ball_preview_width), pygame.SRCALPHA)
48         ball_preview_surface.fill((0, 0, 0, 0)) # transparenter Hintergrund
49         pygame.draw.circle(ball_preview_surface, pygame.Color(color.lower()), (ball_preview_width // 2, ball_preview_width // 2), ball_preview_width // 2)
50
51         ball_preview_label.set_img(ball_preview_surface)
52
53     # Vorschau der Kugel
54     ball_preview_label = UILabel(
55         relative_rect=pygame.Rect(padding + dropdown_width + padding, 480, ball_preview_width, ball_preview_width),
56         text="",
57         manager=manager,
58         container=pause_panel,
59         object_id=ObjectID(class_id='Ball preview', object_id='Ball preview')
60     )
61
62     # Initialisierung Kugelverschmelzung
63     update_ball_preview(ball_color)
64
65     # dropdown_menu für die Anzeige der Wallgeschwindigkeitslinie
66     line_dropdown = UDDropDownMenu(
67         options_list=['On/Off'],
68         starting_option='On' if draw_ball_direction_line else 'Off',
69         relative_rect=pygame.Rect(WIDTH / 2 + padding + 24, 480, dropdown_width + padding, 60),
70         manager=manager,
71         container=pause_panel,
72         object_id=ObjectID(class_id='dropdown', object_id='Wall dropdown')
73     )
74
75     # Funktion zur Aktualisierung oder Änderung der Linienanzeige
76     def update_ball_line_display(option):
77         global draw_ball_line
78         draw_ball_line = option == 'On'
79
80         # Event Handler für Änderungen im Dropdown Menü
81         def handle_line_dropdown(event):
82             if event.type == pygame.gui.UIT_DROP_DOWN_MENU_CHANGED:
83                 if event.ui_element == line_dropdown:
84                     update_ball_line_display(event.text)
85
86         volume_value_label = UIlabel(
87             relative_rect=pygame.Rect(padding + 48, 627, 56, SLIDER_HEIGHT + 12),
88             text=str(int(pygame.mixer.music.get_volume()) * 100),
89             manager=manager,
90             container=pause_panel,
91             object_id=ObjectID(class_id='label', object_id='volume_label')
92         )
93
94         volume_slider = UIHorizontalSlider(
95             relative_rect=pygame.Rect(padding + 24, 625, volume_width, SLIDER_HEIGHT + 12),
96             start_value=pygame.mixer.music.get_volume() * 100,
97             value_range=(0, 100),
98             manager=manager,
99             container=pause_panel,
100             object_id=ObjectID(class_id='horizontal_slider', object_id='volume_slider')
101         )
102
103     # Klickt den "Beenden"-Button hinzu
104     continue_button =UIButton(
105         relative_rect=pygame.Rect(WIDTH / 2 - 200, HEIGHT - 80), (200, 60),
106         text="Quit",
107         manager=manager,
108         container=pause_panel,
109         object_id=ObjectID(class_id='', object_id='quit_button')
110     )
111
112     # Fügt den "Beenden"-Button hinzu
113     quit_button =UIButton(
114         relative_rect=pygame.Rect(WIDTH / 2 + 20, HEIGHT - 70), (140, 40),
115         text="Quit",
116         manager=manager,
117         container=pause_panel,
118         object_id=ObjectID(class_id='', object_id='quit_button')
119     )
120
121     # Aktiviert das Fenster
122     is_running = True
123
124     while is_running:
125         time_delta = clock.tick(60) / 1000.0
126         for event in pygame.event.get():
127             if event.type == pygame.QUIT:
128                 pygame.quit()
129                 sys.exit()
130             elif event.type == pygame.KEYDOWN:
131                 if event.key == pygame.K_ESCAPE or event.key == pygame.K_RETURN:
132                     is_running = False
133                     is_pause_menu_open = False
134                     set_gui_visibility(True)
135                     if pause_panel:
136                         pause_panel.kill()
137                     pause_panel = None
138             elif event.type == pygame.gui.UIT_BUTTON_PRESSED:
139                 if event.ui_element == continue_button:
140                     is_running = True
141                     is_pause_menu_open = False
142                     set_gui_visibility(False)
143                     if pause_panel:
144                         pause_panel.kill()
145                     pause_panel = None
146             elif event.ui_element == quit_button:
147                 pygame.quit()
148                 sys.exit()
149             elif event.type == pygame.gui.UIT_DROP_DOWN_MENU_CHANGED:
150                 if event.ui_element == dropdown:
151                     update_ball_preview(event.text)
152             elif event.ui_element == line_dropdown:
153                 update_ball_line_display(event.text.lower())
154             elif event.type == pygame.gui.UH_HORIZONTAL_SLIDER_MOVED:
155                 if event.ui_element == volume_slider:
156                     volume = event.value * 100
157                     pygame.mixer.music.set_volume(volume)
158                     volume_value_label.set_text(str(int(volume * 100)))
159
160             manager_process_events(event)
161
162             if not GAME_STARTED:
163                 pregame_label.visible = False
164
165             manager_update_time_delta()
166             window.blit(mouse_surface, (0, 0))
167             manager_draw_ui(window)
168             pygame.display.flip()

```

End-Game-Screen



Wenn die Kugel den unteren Bildschirmrand berührt, ist das Spiel sofort beendet. Der End-Game-Screen wird daraufhin getriggert. Hier wird der erreichte Punktestand der gespielten Runde angezeigt. Man hat von hier aus die Option, ein neues Spiel zu starten oder die Anwendung zu schließen.

Erläuterung der GUI

Reduktion der kognitiven Belastung:

Das Menü-Design ist intuitiv und benutzerfreundlich, mit klaren Beschriftungen und visuellen Hinweisen, um die Benutzerführung zu erleichtern. Beispielsweise zeigt das Dropdown-Menü zur Farbauswahl der Kugel eine Vorschau der ausgewählten Farbe an und jedes Feld mit Informationen hat ein individuelles Label.

Strukturierung der Benutzungsschnittstelle:

Die GUI ist logisch und übersichtlich strukturiert. Die Elemente sind in einem klar strukturierten Grid angeordnet mit eindeutigen Rändern.

Kombination verschiedener Interaktionsmodalitäten:

Das Spiel unterstützt sowohl Tastatur- als auch Maussteuerung. Einstellungen wie Lautstärke und Kugelfarbe können bequem mit der Maus angepasst werden, während die Steuerung der Kugel über Tastenkombinationen möglich ist.

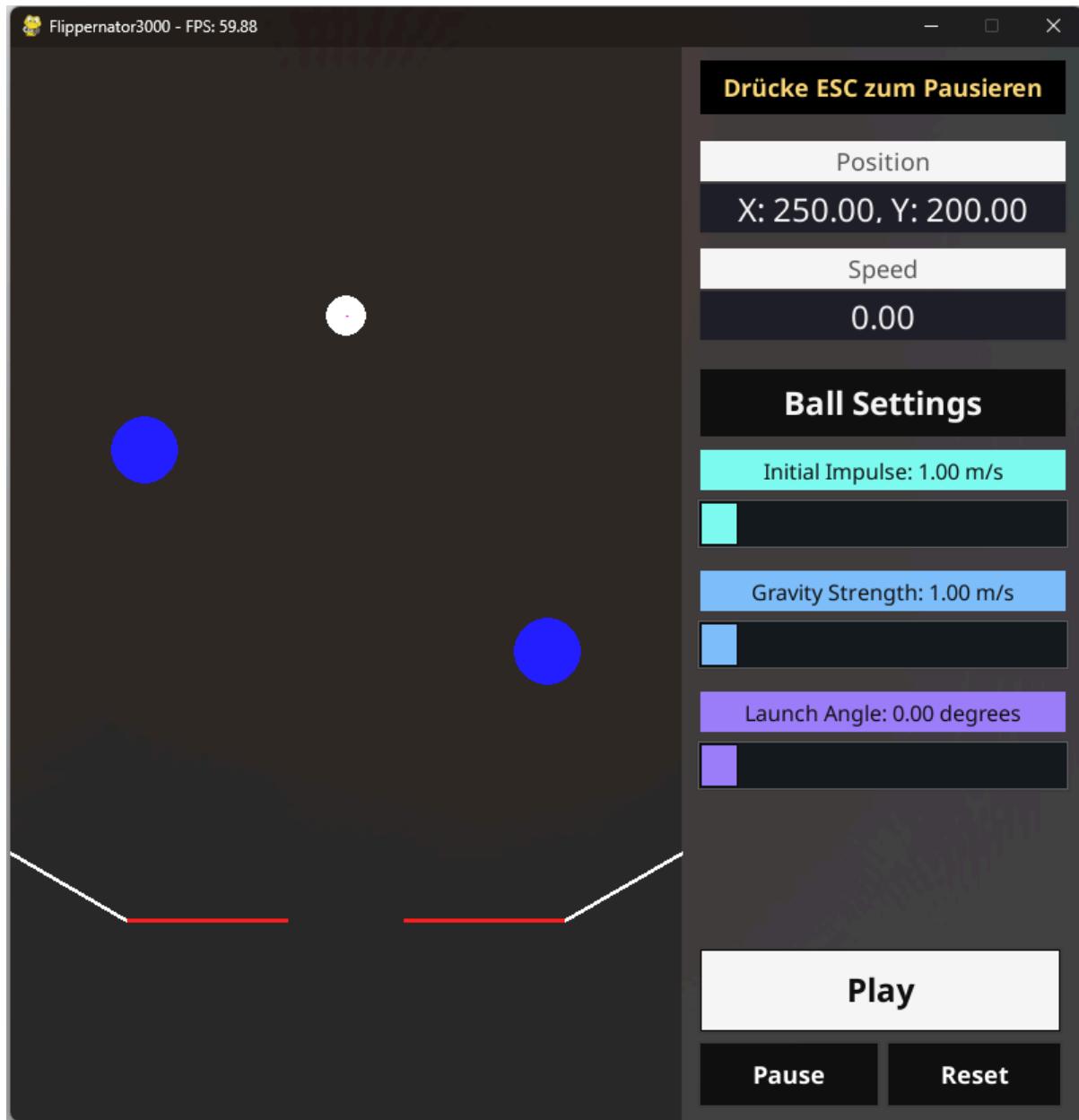
Barrierefreie Gestaltung:

Die Benutzeroberfläche ist barrierefrei gestaltet, mit hohem Kontrast und ausreichender Schriftgröße, um auch von Nutzern mit Sehschwächen gut lesbar zu sein.

Möglichkeit zum Abbrechen und Rückgängigmachen von Aktionen:

Das Spiel kann jederzeit pausiert oder abgebrochen werden. Es gibt zwei Varianten, das Spiel zu pausieren. Entweder kann man das Pause-Menü per ESC-Taste oder den "Pause"-Button öffnen. Optional kann das Spiel eingefroren werden mit der "F"-Taste. Mit der Taste "R" oder dem "Reset"-Button in der GUI kann das Spiel zurückgesetzt werden.

Abgabe 2



Code Snippets

Rollen auf der schiefen Ebene

Die "apply_flipper_physics"-Funktion simuliert das Rollen der Kugel auf einer schießen Ebene (in diesem Fall die Flipper). Sie berechnet die Komponente der Schwerkraft, die entlang des Flipperwinkels wirkt, und aktualisiert entsprechend die Geschwindigkeit der Kugel.

Die weiteren Funktionen ermöglichen es der Kugel, auf den schießen Ebenen (Rampen) zu rollen. Die "check_ramp_collision"-Funktion überprüft Kollisionen zwischen der Kugel und den Rampen, und die reflect_ball-Funktion reflektiert die Kugel basierend auf dem Normalenvektor der Rampe.

```
def apply_flipper_physics(ball_pos, ball_vel, flipper_start, flipper_end):
    flipper_angle = math.atan2(flipper_end[1] - flipper_start[1], flipper_end[0]
    - flipper_start[0])
    gravity_parallel = GRAVITY * math.sin(flipper_angle)

    # Aktualisiert die Geschwindigkeit der Kugel basierend auf der Schwerkraft
    # entlang des Flipperwinkels
    ball_vel[0] += gravity_parallel * dt * math.cos(flipper_angle)
    ball_vel[1] += gravity_parallel * dt * math.sin(flipper_angle)

    # Berechnet die projizierte Position der Kugel nach der Geschwindigkeit
    projected_pos = [ball_pos[0] + ball_vel[0] * dt, ball_pos[1] + ball_vel[1] *
    dt]

    # Berechnet den Abstand der projizierten Position zum Flipper
    dist_to_flipper = point_line_distance(projected_pos, flipper_start,
    flipper_end)

    # Wenn die Kugel den Flipper verlässt, korrigiert die Position, damit sie
    # darauf bleibt
    if dist_to_flipper > BALL_RADIUS:
        normal = get_line_normal(flipper_start, flipper_end)
        ball_pos[0] -= normal[0] * (dist_to_flipper - BALL_RADIUS)
        ball_pos[1] -= normal[1] * (dist_to_flipper - BALL_RADIUS)

    ...

def check_ramp_collision():
    global ball_pos, ball_vel

    # Durchläuft alle Rampen und überprüft, ob die Kugel eine Kollision hat
    for ramp_start, ramp_end in [(ramp_left_start, ramp_left_end),
    (ramp_right_start, ramp_right_end)]:
        if point_line_distance(ball_pos, ramp_start, ramp_end) <= BALL_RADIUS:
```

```

reflect_ball(ramp_start, ramp_end)
break

def reflect_ball(start, end):
    global ball_angular_vel

    # Berechnet den Normalenvektor der Linie
    normal = get_line_normal(start, end)
    midpoint = ((start[0] + end[0]) / 2, (start[1] + end[1]) / 2)
    ball_to_midpoint = (midpoint[0] - ball_pos[0], midpoint[1] - ball_pos[1])

    if (ball_to_midpoint[0] * normal[0] + ball_to_midpoint[1] * normal[1]) > 0:
        normal = (-normal[0], -normal[1])

    ball_vel[0], ball_vel[1] = reflect(ball_vel, normal)

    # Wendet Drehmoment basierend auf der Kollision an
    collision_vector = [ball_pos[0] - midpoint[0], ball_pos[1] - midpoint[1]]
    torque = (collision_vector[0] * ball_vel[1] - collision_vector[1] *
    ball_vel[0]) / (BALL_RADIUS ** 2)
    ball_angular_vel += torque

    # Stellt sicher, dass die Kugel nicht zu weit in das Objekt eindringt
    while point_line_distance(ball_pos, start, end) <= BALL_RADIUS:
        ball_pos[0] += normal[0] * 0.1
        ball_pos[1] += normal[1] * 0.1

```

Abhängigkeit von Reibung und Bewegung

Diese Funktion bewegt die Kugel unter Berücksichtigung von Schwerkraft und Dämpfung (Reibung). Wenn die Geschwindigkeit der Kugel unter einen bestimmten Schwellenwert fällt, wird die Kugel gestoppt.

```
def move_ball():
    global GRAVITY, INITIAL_BALL_IMPULSE, BUMPER_BOUNCE, ball_angle,
ball_angular_vel, ball_vel, ball_pos

    if not GAME_STARTED:
        return

    # Default
    if ball_vel == [0, 0]:
        angle_rad = math.radians(BALL_ANGLE + 90)
        ball_vel[0] = INITIAL_BALL_IMPULSE * math.cos(angle_rad)
        ball_vel[1] = INITIAL_BALL_IMPULSE * math.sin(angle_rad)

    # Fügt die Schwerkraft hinzu
    ball_vel[1] += GRAVITY * dt

    # Aktualisiert die Position der Kugel
    ball_pos[0] += ball_vel[0] * dt
    ball_pos[1] += ball_vel[1] * dt

    # Aktualisiert die Winkelposition
    ball_angle += ball_angular_vel * dt

    # Wendet Dämpfung an
    ball_vel[0] *= DAMPING_FACTOR
    ball_vel[1] *= DAMPING_FACTOR
    ball_angular_vel *= DAMPING_FACTOR

    # Überprüft, ob die Geschwindigkeit unterhalb des Schwellenwerts liegt, und
    # stoppt die Kugel, wenn ja
    if math.hypot(ball_vel[0], ball_vel[1]) < VELOCITY_THRESHOLD:
        ball_vel = [0, 0]
        ball_angular_vel = 0

    # Begrenzt die Position der Kugel auf die Spielfeldgrenzen
    ball_pos[0] = max(min(ball_pos[0], GAME_WIDTH - BALL_RADIUS), BALL_RADIUS)
    ball_pos[1] = max(min(ball_pos[1], HEIGHT - BALL_RADIUS), BALL_RADIUS)
```

Kollisionserkennung (Flipper & Spielfeld)

Diese Funktion überprüft Kollisionen der Kugel mit den Flippern und den Spielfeldgrenzen. Bei einer Kollision wird die Geschwindigkeit der Kugel entsprechend reflektiert.

```
def check_collision():
    global ball_pos, ball_vel

    # Überprüft Kollisionen mit den Flippern
    for flipper_pos, angle, is_right in [(left_flipper_pos, left_flipper_angle,
    False), (right_flipper_pos, right_flipper_angle, True)]:
        start_x, start_y = flipper_pos
        end_x = start_x + FLIPPER_LENGTH * math.cos(math.radians(angle)) * (-1
    if is_right else 1)
        end_y = start_y - FLIPPER_LENGTH * math.sin(math.radians(angle))

        flipper_start = (start_x, start_y)
        flipper_end = (end_x, end_y)

        if point_line_distance(ball_pos, flipper_start, flipper_end) <=
    BALL_RADIUS:
            reflect_ball(flipper_start, flipper_end)
            break

    # Überprüft Kollisionen mit den Spielfeldgrenzen
    if ball_pos[0] <= BALL_RADIUS or ball_pos[0] >= GAME_WIDTH - BALL_RADIUS:
        ball_vel[0] = -ball_vel[0]

    if ball_pos[1] <= BALL_RADIUS or ball_pos[1] >= HEIGHT - BALL_RADIUS:
        ball_vel[1] = -ball_vel[1]
```

Kollisionshandling an einem "passiven" Element

Diese Funktion behandelt Kollisionen der Kugel mit einem passiven Element (einem Bumper). Die Geschwindigkeit der Kugel wird reflektiert und modifiziert, um den Aufprall zu simulieren. Außerdem werden Particles bei einem Aufprall ausgelöst.

```
def reflect_ball_velocity(ball_pos, ball_vel, bumper_pos, bumper_radius):
    global ball_angular_vel

    # Berechnet den Einfallswinkel
    angle_of_incidence = math.atan2(ball_pos[1] - bumper_pos[1], ball_pos[0] - bumper_pos[0])

    # Reflektiert den Geschwindigkeitsvektor
    normal = (math.cos(angle_of_incidence), math.sin(angle_of_incidence))
    dot_product = ball_vel[0] * normal[0] + ball_vel[1] * normal[1]
    ball_vel[0] -= 2 * dot_product * normal[0]
    ball_vel[1] -= 2 * dot_product * normal[1]

    # Multipliziert die Geschwindigkeit mit dem Bumper-Bounce-Faktor
    ball_vel[0] *= BUMPER_BOUNCE
    ball_vel[1] *= BUMPER_BOUNCE

    # Wendet Drehmoment basierend auf der Kollision an
    collision_vector = [ball_pos[0] - bumper_pos[0], ball_pos[1] - bumper_pos[1]]
    torque = (collision_vector[0] * ball_vel[1] - collision_vector[1] * ball_vel[0]) / (BALL_RADIUS ** 2)
    ball_angular_vel += torque

    # Stellt sicher, dass die Kugel nicht zu weit in den Bumper eindringt
    distance = math.hypot(ball_pos[0] - bumper_pos[0], ball_pos[1] - bumper_pos[1])
    overlap = BALL_RADIUS + bumper_radius - distance
    if overlap > 0:
        ball_pos[0] += overlap * normal[0]
        ball_pos[1] += overlap * normal[1]

    # Fügt Partikel an der Kollisionsstelle hinzu
    add_particles(ball_pos)
```

Einstellbar: Position der Kugel

Diese Funktion ermöglicht es dem Benutzer, die Startposition der Kugel mit einem Mausklick festzulegen, solange das Spiel nicht gestartet ist. Das Spiel kann dann über die GUI gestartet werden.

```
def handle_mouse():
    global ball_pos
    if pygame.mouse.get_pressed()[0] and not GAME_STARTED:
        mouse_x, mouse_y = pygame.mouse.get_pos()

        if mouse_x < GAME_WIDTH and not
(initial_impulse_slider.get_abs_rect().collidepoint(mouse_x, mouse_y) or
gravity_strength_slider.get_abs_rect().collidepoint(mouse_x, mouse_y) or
launch_angle_slider.get_abs_rect().collidepoint(mouse_x, mouse_y)):
            ball_pos = list(pygame.mouse.get_pos())
```

Anzeige: Richtung und Geschwindigkeit der Kugel(n)

Anzeige: Richtung und Geschwindigkeit der Kugel(n)

Diese Funktionen zeichnen die Kugel und eine Linie, die die Richtung der Geschwindigkeit anzeigt. Außerdem wird die aktuelle Position und Geschwindigkeit der Kugel in der grafischen Benutzeroberfläche angezeigt.

```
def draw_ball():
    pygame.draw.circle(window, WHITE, (int(ball_pos[0]), int(ball_pos[1])), BALL_RADIUS)
    direction_length = 30 * math.sqrt(ball_vel[0]**2 + ball_vel[1]**2) / 100
    angle = math.atan2(ball_vel[1], ball_vel[0])
    end_pos = (ball_pos[0] + direction_length * math.cos(angle), ball_pos[1] + direction_length * math.sin(angle))
    pygame.draw.line(window, PURPLE, (ball_pos[0], ball_pos[1]), end_pos, 2)

def draw_gui():
    position_text = f'X: {ball_pos[0]:.2f}, Y: {ball_pos[1]:.2f}'
    speed = math.sqrt(ball_vel[0]**2 + ball_vel[1]**2) / 100
    speed_text = f'{speed:.2f}'

    position_value.set_text(position_text)
    speed_value.set_text(speed_text)

    position_label = UILabel(
        relative_rect=pygame.Rect((GAME_WIDTH + 14, 70), (UI_WIDTH - 28, 30)),
        text="Position",
        manager=manager,
        object_id=ObjectID(class_id="@label", object_id="#position_label")
    )

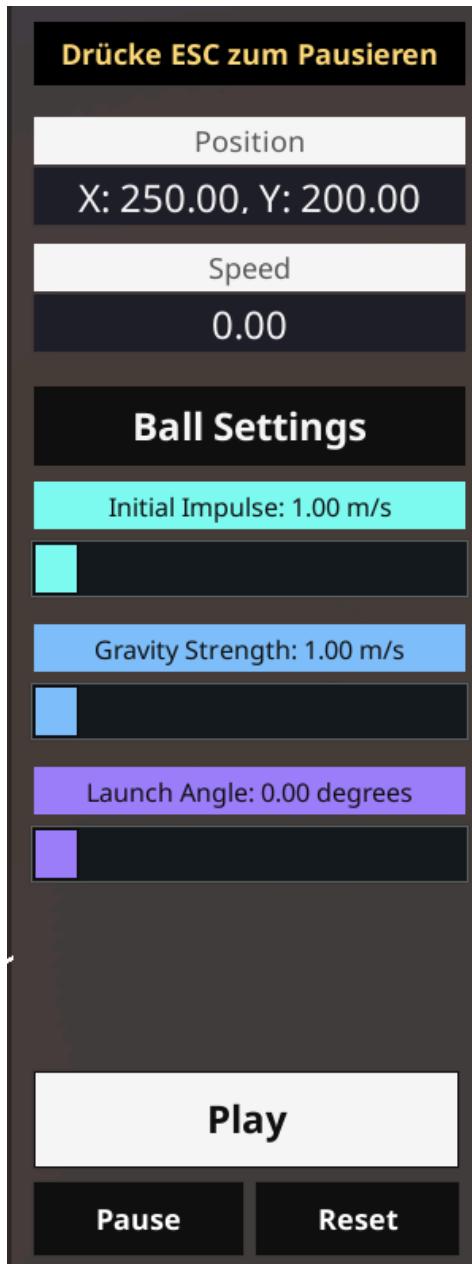
    position_value = UITextBox(
        relative_rect=pygame.Rect((GAME_WIDTH + 12, 100), (UI_WIDTH - 24, 40)),
        html_text="X: 0.00, Y: 0.00",
        manager=manager,
        object_id=ObjectID(class_id="@text_box", object_id="#position_value")
    )

    speed_label = UILabel(
        relative_rect=pygame.Rect((GAME_WIDTH + 14, 150), (UI_WIDTH - 28, 30)),
        text="Speed",
        manager=manager,
        object_id=ObjectID(class_id="@label", object_id="#speed_label")
    )

    speed_value = UITextBox(
        relative_rect=pygame.Rect((GAME_WIDTH + 12, 180), (UI_WIDTH - 24, 40)),
        html_text="0.00",
        manager=manager,
        object_id=ObjectID(class_id="@text_box", object_id="#speed_value")
    )
```

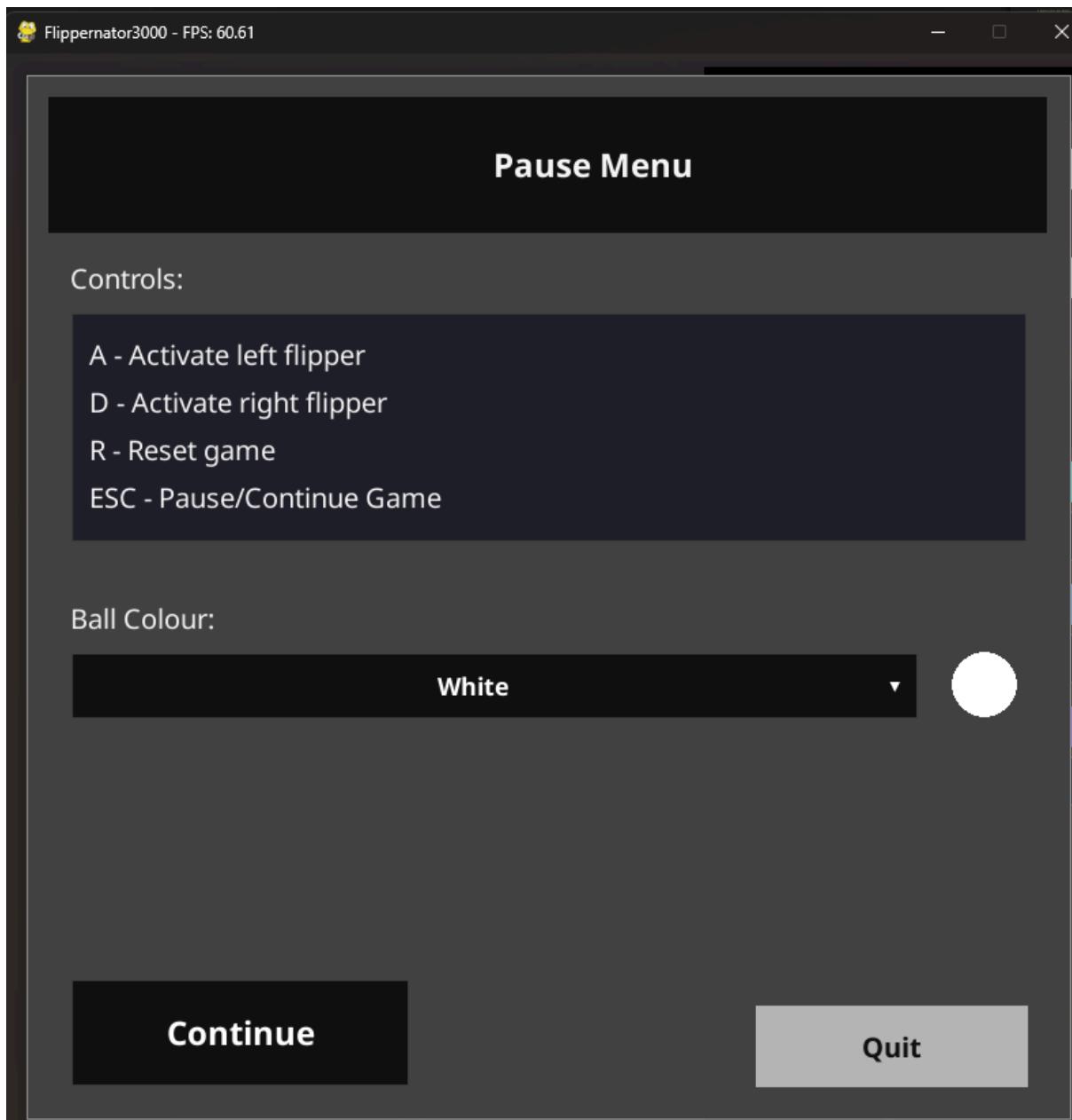
GUI

Game Settings:



- Anzeige der aktuellen Ball-Position
- Anzeige der aktuellen Ball-Geschwindigkeit
- Anpassbarer Slider für den Startimpuls
- Anpassbarer Slider für die Gravitationsstärke
- Anpassbarer Slider für den Startwinkel
- Buttons für Spielstart, Aufrufen des Pause Menüs, Spiel zurücksetzen

Pause Menu:



- Liste der Steuerung
- Anpassung der Farbe des Balles
- Buttons zum Fortsetzen und Anwendung schließen

Der relevante Code für das GUI ist ungefähr von Zeile 500 -83, sowie teilweise in der game_loop-Funktion. Design der UI-Elemente sind in data/theme.json zu finden.

Ausführbare Datei

Die ausführbare Datei (.exe) finden Sie im Ordner "Flippernator3000 Game", darin ist die Datei "Flippernator3000.exe".

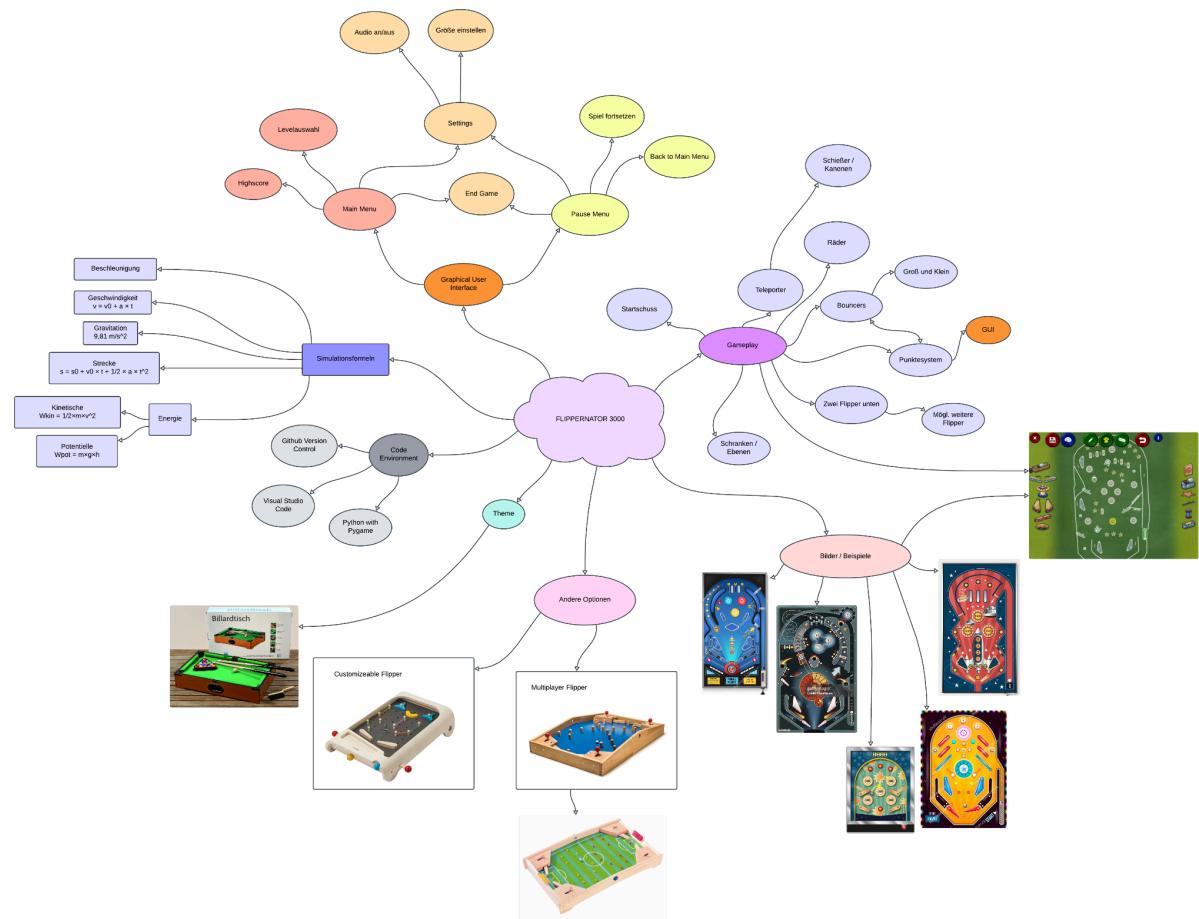
Steuerung

A – Flipper Links aktivieren
D – Flipper Rechts aktivieren
R – Spiel zurücksetzen
ESC – Spiel pausieren / fortsetzen

Information zum Spielen: Es muss auf dem Spielfeld eine Position der Kugel mit der Maus gewählt werden, dann lässt sich das Spiel erst starten.

Abgabe 1

Brainstorming (Lucidspark)



Code Snippets

1. Die Kugel bewegt sich in einer beliebigen Richtung:

Das folgende Snippet aus der Funktion move_ball zeigt, wie die Bewegung der Kugel basierend auf ihrer aktuellen Geschwindigkeit aktualisiert wird. Dies ermöglicht es der Kugel, sich in jede Richtung zu bewegen, abhängig von ihrer Geschwindigkeitsvektor-Komponenten ball_vel[0] und ball_vel[1]:

```
# main.py

# Aktualisiere die horizontale / vertikale Position der Kugel
ball_pos[0] += ball_vel[0]
ball_pos[1] += ball_vel[1]
```

2. Eine oder mehrere Beschleunigungen können wirken:

Das Snippet zeigt, wie die Schwerkraft als eine konstante Beschleunigung auf die Kugel wirkt, wodurch ihre vertikale Geschwindigkeit kontinuierlich angepasst wird. Der INITIAL_BALL_IMPULSE kann mit einem Slider zwischen 1 und 10 festgelegt werden (Default = 1) und gibt der Kugel eine Anfangsbeschleunigung:

```
# config.py

INITIAL_BALL_IMPULSE = 1
```

```
# main.py

if ball_vel == [0, 0]:
    # Setze den Anfangsimpuls nach unten
    ball_vel[1] = INITIAL_BALL_IMPULSE

# Schwerkraft anwenden, die die Kugel nach unten zieht
ball_vel[1] += GRAVITY
```

3. Startimpuls mit einer frei einstellbaren Richtung und Stärke &

4. Einstellbar: Startposition, Startimpuls, Beschleunigungseinflüsse:

Das folgende Snippet aus handle_mouse zeigt, wie das Spiel durch einen Mausklick gestartet wird. Dabei wird der Startpunkt der Kugel genau da hin

gesetzt, auf den man im Game-Window drückt. Man hat davor noch die Option, mit Hilfe von Slidern, die Richtung, Stärke und die Gravitationsstärke einzustellen.

```
# main.py

def handle_mouse():
    global ball_pos, ball_vel, GAME_STARTED, BALL_ANGLE, INITIAL_BALL_IMPULSE,
    GRAVITY_STRENGTH, GRAVITY
    if pygame.mouse.get_pressed()[0]:
        mouse_x, mouse_y = pygame.mouse.get_pos()
        # Überprüfe, ob die Maus auf einem der Slider ist
        if slider1_rect.collidepoint(mouse_x, mouse_y):
            INITIAL_BALL_IMPULSE = int((mouse_x - slider1_rect.left) /
    slider1_rect.width * (SLIDER_MAX_VALUE - SLIDER_MIN_VALUE) + SLIDER_MIN_VALUE)
        elif slider2_rect.collidepoint(mouse_x, mouse_y):
            GRAVITY_STRENGTH = int((mouse_x - slider2_rect.left) / slider2_rect.width
* (SLIDER_MAX_VALUE - SLIDER_MIN_VALUE) + SLIDER_MIN_VALUE)
            GRAVITY = 0.1 * GRAVITY_STRENGTH
        elif angle_slider_rect.collidepoint(mouse_x, mouse_y):
            BALL_ANGLE = int((mouse_x - angle_slider_rect.left) /
angle_slider_rect.width * (SLIDER_MAX_ANGLE - SLIDER_MIN_ANGLE) +
SLIDER_MIN_ANGLE)

    else:
        # Starte das Spiel nur, wenn außerhalb der Sliderbereiche geklickt wird
        if not GAME_STARTED:
            angle_rad = math.radians(BALL_ANGLE + 90)
            ball_vel = [
                INITIAL_BALL_IMPULSE * math.cos(angle_rad),
                INITIAL_BALL_IMPULSE * math.sin(angle_rad)
            ]
            ball_pos = list(pygame.mouse.get_pos())
            GAME_STARTED = True
```

Weitere Wichtige Methoden:

Bewegung der Kugel

In der Funktion move_ball werden alle Auswirkungen auf das Ball-Movement angewendet. Dabei werden Einflüsse wie GRAVITY, DAMPENING oder INITIAL_BALL_IMPULS übertragen und die Geschwindigkeit der Kugel aktualisiert. Auch Kollisionen mit Außenwänden und Bumpern werden hier berechnet.

```
def move_ball():
    global GRAVITY, INITIAL_BALL_IMPULSE, BUMPER_BOUNCE

    # Wenn das Spiel nicht gestartet ist, wird die Funktion vorzeitig verlassen.
    if not GAME_STARTED:
        return

    # Wenn die Kugel noch keine Anfangsgeschwindigkeit hat
    if ball_vel == [0, 0]:
        ball_vel[1] = INITIAL_BALL_IMPULSE # Setze den Anfangsimpuls nach unten

    # Schwerkraft anwenden, die die Kugel nach unten zieht
    ball_vel[1] += GRAVITY * DAMPENING

    # Aktualisiere die horizontale / vertikale Position der Kugel
    ball_pos[0] += ball_vel[0]
    ball_pos[1] += ball_vel[1]

    # Überprüfung auf Kollision mit den Seitenwänden des Spielfelds
    if ball_pos[0] <= BALL_RADIUS or ball_pos[0] >= WIDTH - BALL_RADIUS:
        # Kehre die horizontale Geschwindigkeit um
        ball_vel[0] = -ball_vel[0]

    if ball_pos[1] <= BALL_RADIUS or ball_pos[1] >= HEIGHT - BALL_RADIUS:
        # Kehre die vertikale Geschwindigkeit um
        ball_vel[1] = -ball_vel[1]

    # Kollision mit Bumpern
    for bumper in bumpers:
        if math.hypot(ball_pos[0] - bumper['pos'][0], ball_pos[1] - bumper['pos'][1]) < BALL_RADIUS + bumper['radius']:
            if not bumper['active']:
                bumper['active'] = True
                bumper['timer'] = 10 # Anzahl der Frames, die die Animation dauert
                angle = math.atan2(ball_pos[1] - bumper['pos'][1], ball_pos[0] - bumper['pos'][0])
                ball_vel[0] += BUMPER_BOUNCE * math.cos(angle)
                ball_vel[1] += BUMPER_BOUNCE * math.sin(angle)
```

Kollision der Kugel mit den Flippern

Damit das Spiel auch spielbar ist, haben die Flipper Kollisionen. Bei den Flippern wird außerdem ein FLIPPER_BOUNCE hinzugefügt, damit das Spiel nicht sofort zu Ende ist.

```
def reflect_ball(start, end):
    global FLIPPER_BOUNCE, DAMPENING

    normal = get_line_normal(start, end)
    midpoint = ((start[0] + end[0]) / 2, (start[1] + end[1]) / 2)
    ball_to_midpoint = (midpoint[0] - ball_pos[0], midpoint[1] - ball_pos[1])

    if (ball_to_midpoint[0] * normal[0] + ball_to_midpoint[1] * normal[1]) > 0:
        # Normalenvektor umkehren, wenn er zum Flipper zeigt
        normal = (-normal[0], -normal[1])

    new_velocity = reflect((ball_vel[0], ball_vel[1]), normal)

    ball_vel[0] = (new_velocity[0] + abs(normal[0])) *- FLIPPER_BOUNCE
    ball_vel[1] = (new_velocity[1] + abs(normal[1])) *- FLIPPER_BOUNCE

def check_collision():
    # Überprüft Kollisionen zwischen der Kugel und den Flippern und handhabt die
    # Folgen einer Kollision
    global ball_pos, ball_vel, collision_coldown

    if collision_coldown > 0:
        collision_coldown -= 1
        return

    for flipper_pos, angle, is_right in [(left_flipper_pos, left_flipper_angle,
                                           False), (right_flipper_pos, right_flipper_angle, True)]:
        # Berechne die Positionen der Außenwände des Flippers
        start_x, start_y = flipper_pos
        end_x = start_x + FLIPPER_LENGTH * math.cos(math.radians(angle)) * (-1 if
is_right else 1)
        end_y = start_y - FLIPPER_LENGTH * math.sin(math.radians(angle))
        normal = get_line_normal((start_x, start_y), (end_x, end_y))
        perpendicular = (-normal[1], normal[0])

        # Überprüfe Kollision mit den Außenwänden des Flippers
        wall_start = (start_x + perpendicular[0] * FLIPPER_WIDTH / 2, start_y +
perpendicular[1] * FLIPPER_WIDTH / 2)
        wall_end = (end_x + perpendicular[0] * FLIPPER_WIDTH / 2, end_y +
perpendicular[1] * FLIPPER_WIDTH / 2)
        if point_line_distance(ball_pos, wall_start, wall_end) <= BALL_RADIUS:
            reflect_ball(wall_start, wall_end)
            collision_coldown = COLLISION_COOLDOWN_MAX
            break
```

GUI

Im GUI werden folgende Informationen dargestellt:

- Hinweis zum Pause-Menü
- Aktuelle Position der Kugel auf 2 Nachkommastellen gerundet
- Aktuelle Geschwindigkeit der Kugel auf 2 Nachkommastellen gerundet
- Slider für Startgeschwindigkeit, Startwinkel und Stärke der Gravitation

```
def draw_gui():
    # Zeigt die GUI-Elemente auf dem Bildschirm an, einschließlich der aktuellen
    # Position und Geschwindigkeit der Kugel.
    position_text = f'X: {ball_pos[0]:.2f}, Y: {ball_pos[1]:.2f}'

    speed = math.sqrt(ball_vel[0]**2 + ball_vel[1]**2)
    speed_text = f'Speed: {speed:.2f}'

    pause_text = "Drücke ESC zum Pausieren"

    position_surf = font.render(position_text, True, pygame.Color('white'))
    speed_surf = font.render(speed_text, True, pygame.Color('white'))
    pause_surf = font.render(pause_text, True, pygame.Color('yellow'))

    window.blit(pause_surf, (10, 10))
    window.blit(position_surf, (10, 40))
    window.blit(speed_surf, (10, 70))

def draw_slider(slid_rect, slider_value, text, text_pos, min_value,
max_value):
    pygame.draw.rect(window, SLIDER_COLOR, slid_rect)
    # Berechne die Position des Handles basierend auf dem Slider-Wert
    normalized_value = (slider_value - min_value) / (max_value - min_value)
    handle_pos = slid_rect.left + normalized_value * (slid_rect.width -
SLIDER_HEIGHT)
    handle_rect = pygame.Rect(handle_pos, slid_rect.centery - SLIDER_HEIGHT // 2, SLIDER_HEIGHT, SLIDER_HEIGHT)
    pygame.draw.rect(window, SLIDER_HANDLE_COLOR, handle_rect)
    text_with_value = f'{text}: {slider_value}'
    text_surf = font.render(text_with_value, True, SLIDER_TEXT_COLOR)
    window.blit(text_surf, text_pos)

# Im Gameloop werden die Slider dann gezeichnet
draw_slider(slid1_rect, INITIAL_BALL_IMPULSE, "Initial Ball Impulse", (320, 20), SLIDER_MIN_VALUE, SLIDER_MAX_VALUE)

draw_slider(slid2_rect, GRAVITY_STRENGTH, "Gravity Strength", (320, 70),
SLIDER_MIN_VALUE, SLIDER_MAX_VALUE)

draw_slider(angle_slider_rect, BALL_ANGLE, "Launch Angle", (320, 120),
SLIDER_MIN_ANGLE, SLIDER_MAX_ANGLE)
```

