Least Time to the Rescue

Suran Warnakulasooriya

June 21, 2023

Abstract

The most efficient route that a lifeguard must take to rescue a drowning person is described by the law of refraction. This law can be fittingly derived by finding the horizontal distance from the initial position of the lifeguard to the boundary between the two mediums that minimizes the time taken. Using realistic values for how far away the lifeguard is from their target and their speeds on sand and in water, we can find realistic jump points to run to and angles to head in any situation. From there, we find that the difference between the optimal path and straight path is only significant when the target is close to the waterfront but far from the lifeguard. In this scenario, the difference between times approaches a minute, making it crucial for a lifeguard to know to take the optimal path in this situation.

1 Introduction

In the second century CE, Ptolemy attempted to find an empirical relationship between the angles of refraction, but could only approximate for small angles. In 984, the Persian scientist Ibn Sahl studied Ptolemy's work and found a more concrete geometrical relationship between the angles. In 1621, the Dutch astronomer Willebrord Snellius used conservation of momentum to relate the angles using their sines, creating what we now know as the Snell's law. Pierre de Fermat later proved the same relationship using differential calculus and the principle of least time: that light travels along the most time-efficient path between two given points. In his lectures on physics[*], Richard Feynman analogized the behavior of light as a lifeguard taking the most efficient route from the beach to rescue a drowning person in the sea. It is from this analogy between light and the lifeguard that we begin to find the lifeguard's path.

2 Finding the Optimal Path

The lifeguard's path consists of two segments that span across three points. The lifeguard must run from their initial position (L) to the optimal jump point (J) at the waterfront; from there, they must swim to the target at position (D). The optimal path from L to D can be described by a single value: x, the horizontal distance between L and J. Since the optimal path is the path that takes the least time, we need to find an x that minimizes T, the time taken to traverse the path; x depends on five variables: the vertical distance between L and J (d_s), the vertical distance between J and D (d_w), the total horizontal distance between the L and D (l), the lifeguard's speed on sand (v_s), and the lifeguard's speed in water (v_w). Additionally, the angle between the lifeguard's trajectory to J and the vertical at J (θ_s) and the angle between the lifeguard's trajectory from J and the vertical at J (θ_w) can be calculated (see Figure 1).

[*]QED: The Strange Theory of Light and Matter, 1985.

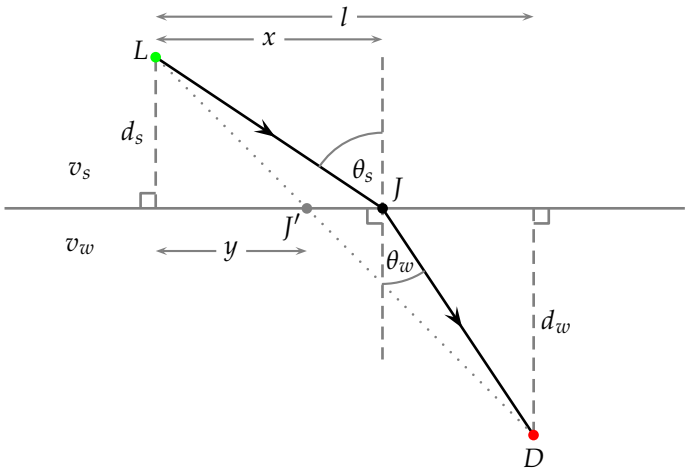

Figure 1: The lifeguard's path to the target along with the variables used in the computation. The path that minimizes time is the trajectory LJD. The straight path is $LJ'D$.

Some logical constraints can be applied in this situation. For example, $x \leq l$ since the lifeguard cannot run past the target. Also, x will only equal l when $x = l = 0$. Since humans always run faster than they swim, $v_s > v_w$. For a fit but average lifeguard, we will use $v_s = 2.7$ m/s[*] and $v_w = 0.9$ m/s[†]. Lifeguards are usually perched no farther than 10m from the waterfront[‡], so 0m $\leq$ d_s $\leq$ 10m. Lifeguards also must be able to swim up to 500m without assistance or stopping[‡], so 0m $\leq$ d_w $\leq$ 500m. A lifeguard generally covers 300m of the waterfront[‡]mass.gov, meaning 150m in one direction. Since our model is symmetrical, we only need to consider one side, so 0m $\leq l \leq$ 150m.

The time T taken to traverse a given path is the sum of the time spent on sand and the time spent in water. Therefore,

$$T = \frac{LJ}{v_s} + \frac{JD}{v_w}. \tag{1}$$

LJ and JD can be expressed in terms of d_s, d_w, l, and x using the Pythagorean theorem, yielding

$$\boxed{T = \frac{\sqrt{d_s^2 + x^2}}{v_s} + \frac{\sqrt{d_w^2 + (l-x)^2}}{v_w}.} \tag{2}$$

Since we want to find an x that minimizes T, we need to take the derivative of T with respect to x and set the derivative to 0.

[*] proudtorun.org/average-human-running-speed
[†] ukfitnessevents.co.uk/swimming/what-is-the-average-swimming-speed
[‡] mass.gov/forms/pool-waterfront-safety-interest-form

$$\frac{dT}{dx} = \frac{x}{v_s\sqrt{d_s^2 + x^2}} - \frac{l-x}{v_w\sqrt{d_w^2 + (l-x)^2}} = 0. \tag{3}$$

From the Pythagoras theorem, we note that $\frac{x}{\sqrt{d_s^2+x^2}} = \sin\theta_s$ and $\frac{l-x}{\sqrt{d_w^2+(l-x)^2}} = \sin\theta_w$, so the derivative can be rewritten as

$$\frac{dT}{dx} = \frac{\sin\theta_s}{v_s} - \frac{\sin\theta_w}{v_w} = 0. \tag{4}$$

This substitution leads us to a proof of Snell's law (5), which relates the velocities and angles in different mediums when light refracts through them, since, like the lifeguard, light obeys the principle of least time.

$$\boxed{\frac{\sin\theta_s}{v_s} = \frac{\sin\theta_w}{v_w}.} \tag{5}$$

The fact that the x that satisfies (3) actually minimizes T can be validated by taking the derivative of (3) with respect to x. This results in,

$$\frac{d^2T}{dx^2} = \frac{1}{v_s\left(d_s^2 + x^2\right)^{\frac{1}{2}}} - \frac{x^2}{v_s\left(d_s^2 + x^2\right)^{\frac{3}{2}}} + \frac{1}{v_w\left(d_w^2 + (l-x)^2\right)^{\frac{1}{2}}} - \frac{(l-x)^2}{v_w\left(d_w^2 + (l-x)^2\right)^{\frac{3}{2}}}.$$

Since the first term on the right hand side is greater than the second term (the reason being $d_s^2 + x^2 > x^2$), and that the third term is greater than the fourth term (the reason being $d_w^2 + (l-x)^2 > (l-x)^2$), the sum of the terms on the right are positive. Therefore, $\frac{d^2T}{dx^2} > 0$, which validates that the x that satisfies (3) actually minimizes T.

Returning to our quest to find x, rearranging (3) gives us

$$\frac{x}{v_s\sqrt{d_s^2 + x^2}} = \frac{l-x}{v_w\sqrt{d_w^2 + (l-x)^2}}. \tag{6}$$

After squaring both sides and following through with many algebraic steps, we get the following quartic equation for x,

$$\boxed{\left(l - v^2\right)x^4 - 2l\left(1 - v^2\right)x^3 + \left(d_s^2 - v^2d_w^2 + l^2(1 - v^2)\right)x^2 - 2ld_s^2x + l^2d_s^2 = 0,} \tag{7}$$

where $v = \frac{v_w}{v_s}$ (this substitution is made since we find the ratio $\frac{v_w}{v_s}$ often in the computational steps). We note that $v < 1$ since $v_s > v_w$. Being a quartic equation, (7) has four solutions. To find the optimal x out of the four solutions, we need to consider only real positive x values. Since the lifeguard travels slower in water, they will always refract towards the normal when transitioning from sand to water, meaning they will cover less horizontal distance in water than

on sand. Since the lifeguard travels more horizontal distance on sand $x \geq \frac{l}{2}$. The lifeguard also cannot go past l, so $\frac{l}{2} \leq x \leq l$. If there are multiple valid x values, we need to run a candidate test to see which x produces the lowest T.

It helps to compare T with the time to go from L to D using a straight path (the dotted line in Figure (1)). Labeling the time it takes for the lifeguard along the straight path as T' and the horizontal distance to the jump point (J') as y, it follows that

$$\boxed{T' = \frac{\sqrt{d_s^2 + y^2}}{v_s} + \frac{\sqrt{d_w^2 + (l-y)^2}}{v_w}.} \tag{8}$$

From the geometry in Figure (1) we obtain that $d_s/y = d_w/(l-y)$, from which we find that

$$\boxed{y = l d_s/(d_s + d_w).} \tag{9}$$

This value for y can then be substituted in (8) to find T'.

A code in Python was written to solve for (7) and (9) and to find the values of T and T' using (2) and (8) under varying conditions.* For example, Figure 2 shows the difference in times between optimal and straight paths for varying l for fixed d_s and d_w. Figure 3 shows the variation in angles θ_s and θ_w for the same conditions as in Figure 2. Figure 4 shows a heatmap showing the time difference between the optimal and straight paths when l and d_w are varied for fixed d_s. As can be seen from the heatmap, taking the optimal path instead of the straight path matters most with a high l and low d_w, since the lifeguard can choose to run faster on the sand next to the waterfront and swim a short distance instead of swimming almost the entire way, saving nearly a minute of critical time.

*To access the code in GitHub, please visit suranw.net. The code for generating the heatmap and for creating an interactive simulation is also given in section 3 of this paper.

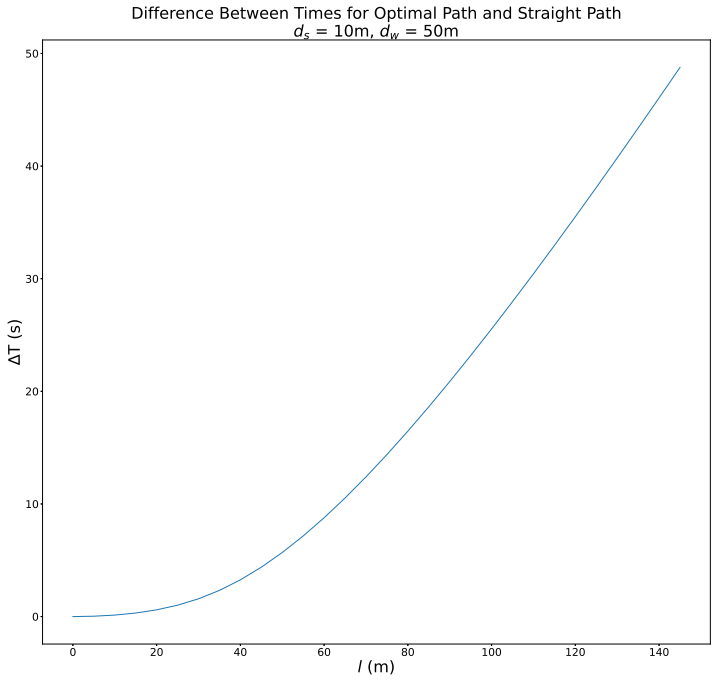

Figure 2: A graph showing the effect that increasing l has on ΔT. As l increases, the straight path covers more water while the optimal path runs nearly horizontally across the sand before covering a short distance in water. The difference in speeds on sand and in water leads to a seemingly exponential increasing difference in times with more horizontal distance.

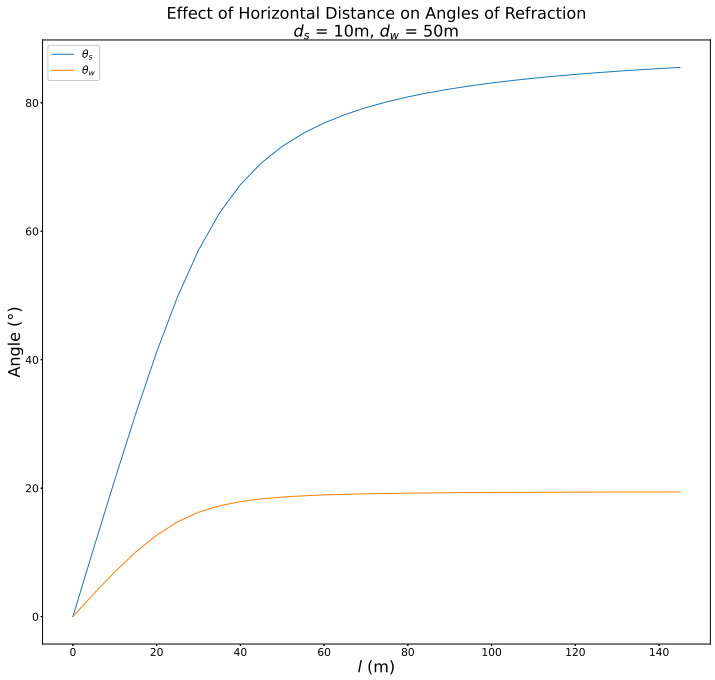

Figure 3: A graph showing the effect that increasing l has on θ_s and θ_w. As l increases, the part of the optimal path that is on sand (from L to J) approximates a horizontal line, so θ_s begins to approximate a right angle. Since d_s and d_w are kept constant, θ_w approximates a certain angle, 20° in this case, to cover the segment in water as θ_s approximates 90°.

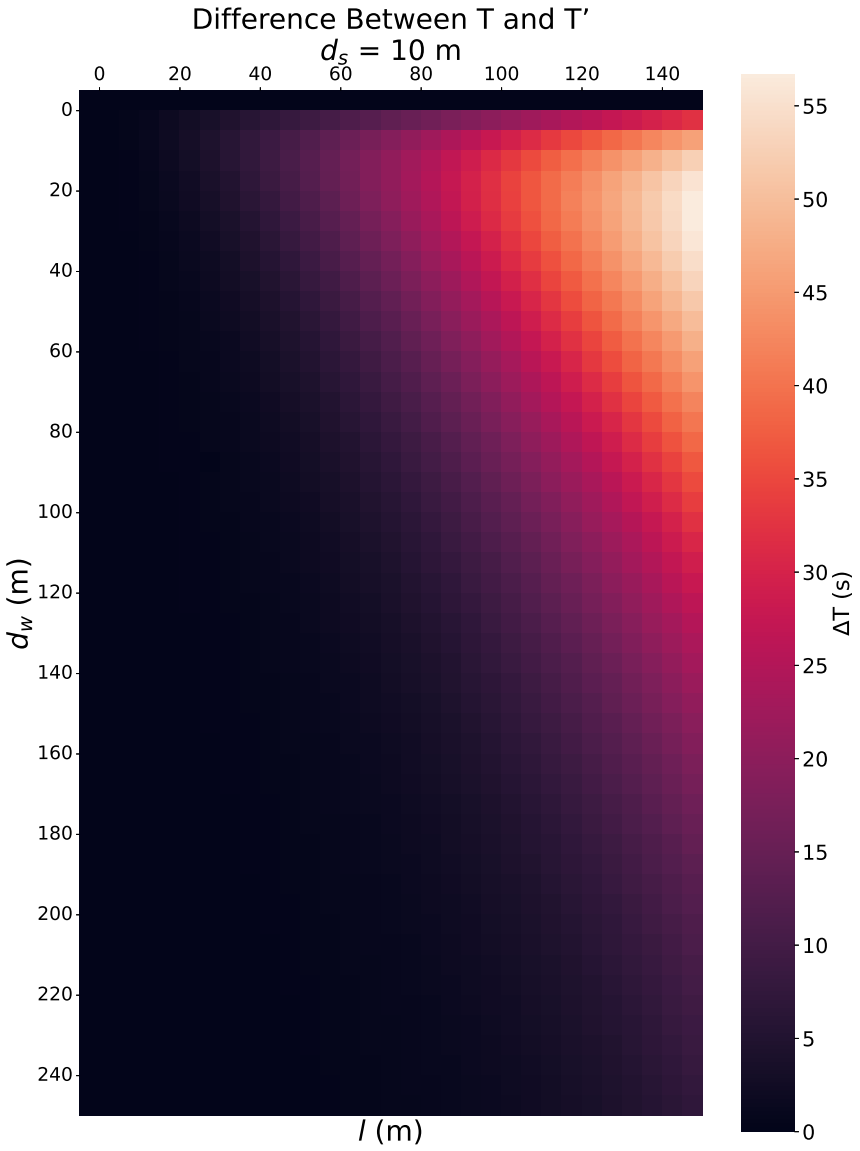

Figure 4: A heatmap showing the difference between the time (T) to traverse the optimal path and the time (T') to traverse the straight path. Here the lifeguard is at a distance of $d_s = 10$m from the waterfront. When $d_w = 0$m, the target is on the waterfront and the lifeguard's path is entirely on sand, making both times equal and their difference 0s (this is a non-drowning scenario). When $l = 0$m, the target is directly in front of the lifeguard so the optimal path is again the same as the straight path, making the difference 0s. Lifeguards are supposed to swim up to $d_w = 500$m, but the difference between times is no more than 5s when $d_w > 250$m. Taking the optimal path instead of the straight path matters most with a high l and low d_w, since the lifeguard can choose to run faster on the sand next to the waterfront and swim a short distance instead of swimming almost the entire way, saving nearly a minute of time.

3 Code

3.1 For Generating the Heatmap

```
from sympy.solvers import solve # solve quartic equation for x
from sympy import Symbol # symbol class for the library to recognise x
from math import atan, pi # arctan to find the angles and pi to convert to degrees
import matplotlib.pyplot as plt

x = Symbol('x') # horizontal distance the lifeguard must travel before entering the water
Vs, Vw = 2.7, 0.9 # speed in sand and speed in water (m/s)

Ds = 10 # distance (m) from the lifeguard to the beach (0 <= Ds <= 10)
Dw = 0 # distance (m) from the beach to the target (0 <= Dw <= 500)
l = 0 # total horizontal distance (m) between the lifeguard and target (0 <= l <= 150)

def find_T(x,Hs,Hw,l): # find the time required to traverse a given path specified by x, Hs, Hw, and l
    return (((Hs**2)+(x**2))**0.5)/Vs + (((Hw**2)+((l-x)**2))**0.5)/Vw

def calculate(l,Hs,Hw,Vs,Vw): # with given values, find x, the angles of refraction, time to traverse the refracted path, and time to traverse the straight path
    V = Vw/Vs # the ratio between the speeds occurs often within the quartic equation for x, so it is calculated once separately
    # use sympy.solvers to solve the quartic equation for x
    S = solve((x**4)*(1-V**2) - 2*l*(x**3)*(1-V**2) + (x**2)*(Hs**2-(V**2)*(Hw**2)+(l**2)*(1-V**2)) - 2*l*(Hs**2)*x + (Hs**2)*(l**2), x)
    # solve can yield multiple real positive solutions so a candidate test is needed
    best_T = 9e18 # set a ludicrously high best time so that it is guaranteed to be reduced
    best_x = 0 # set a throwaway best x candidate
    for s in S: # for each candidate solution s in the list of solutions ...
        try:
            float(s)
            if s >= 0: # check if s is a real positive number
                candidate_T = find_T(s,Hs,Hw,l) # find the time using the candidate s
```

```
                        if candidate_T < best_T: # set the best time and best x if the candidate s yields a faster time than the current best
                            best_T = candidate_T
                            best_x = s
            except: pass

      y = (Hs*l)/(Hs+Hw)
      if best_T != 9e18: # if there was a valid solution, best_T would be set to something other than 9e18, so if it is not 9e18, a valid solution was found
            try:
                  X = best_x; T = best_T; theta_s = atan(X/Hs)*180/pi; theta_w = atan((l-X)/Hw)*180/pi # find the angles that the lifeguard enters and leaves the beach from the normal in degrees
                  T_prime = find_T(y,Hs,Hw,l) # find the time to traverse the straight path
                  return round(X,3), round(theta_s,3), round(theta_w,3), round(T,3), round(T_prime,3)
            except: pass

      return None, None, None, None, None # if best_T is still 9e18, then no valid solutions were found

ind = []
dep1 = []
dep2 = []

dwr = 250
lr = 150

h = [0 for _ in range(0,lr+5,5)]
heatmap = [h]

from random import randint

for Dw in range(5,dwr+5,5): # loop through various total horizontal distances
      heatmap.append([0])
      for l in range(5,lr+5,5):
            _x, _theta_s, _theta_w, t, t_prime = calculate(l,Ds,Dw,Vs,Vw) # calculate values with the given l
            if _x != None and t != None and t_prime != None: # if the distance was valid
                  delta = round(float(t_prime-t),3)
```

```
61                  print(delta)
62                  heatmap[-1].append(delta)
63              else:
64                  heatmap[-1].append(0)
65
66  print()
67
68  # choose whether to display both the optimal and straight paths (
        ↪ comparison) or just their difference (delta)
69  display = 'delta' # comparison or delta
70
71  # set up the plot for the given setting
72  if display == 'comparison':
73     plt.ylabel('Time (s)')
74     plt.title('Times for Optimal Path and Straight-Line Path\n$d_{s}$ =
           ↪ {0} ft , Dw = {Dw} ft',fontsize=40)
75     plt.plot(ind,dep1,label='$T_{Optimal}$')
76     plt.plot(ind,dep2,label='$T_{Straight}$')
77  elif display == 'delta':
78        plt.title('Difference Between T and T_Prime\n$d_{s}$ = 10 m',
              ↪ fontsize=30)
79        import seaborn as sns
80        print(len(heatmap))
81        print(len(heatmap[0]))
82        print(len(heatmap[-1]))
83        sns.set_context("paper", font_scale=2.5)
84        ax = sns.heatmap(heatmap,square=True,cbar_kws={'label':'DeltaT (
              ↪ s)','ticks':[i for i in range(0,60,5)],'shrink':1})
85        ax.set_xlabel('$l$ (m)',fontsize=30)
86        ax.set_ylabel('$d_{w}$ (m)',fontsize=30)
87        ax.set_xticks([4*i+1 for i in range((lr+20)//20)])
88        ax.set_yticks([4*i+1 for i in range((dwr+20)//20)])
89        ax.set_xticklabels([i for i in range(0,lr+10,20)],fontsize=20)
90        ax.set_yticklabels([i for i in range(0,dwr+10,20)],fontsize=20)
91        ax.xaxis.tick_top()
92
93  plt.show()
94  plt.clf()
95  plt.close()
```

3.2 For the Simulation

```

import pygame
from sympy.solvers import solve # solve quartic equation for x
from sympy import Symbol # symbol class for the library to recognise x
from math import atan, pi # arctan to find the angles and pi to convert to degrees
from os import path

x = Symbol('x') # horizontal distance the lifeguard must travel before entering the water
X = 0
Vs, Vw = 2.7, 0.9 # speed in sand and speed in water (m/s)

Ds = 30 # distance (m) from the lifeguard to the beach (0 <= Ds <= 30)
Dw = 150 # distance (m) from the beach to the target (0 <= Dw <= 300)
l = 0 # total horizontal distance (m) between the lifeguard and target

def find_T(x,Hs,Hw,l): # find the time required to traverse a given path specified by x, Hs, Hw, and l
    return (((Hs**2)+(x**2))**0.5)/Vs + (((Hw**2)+((l-x)**2))**0.5)/Vw

def calculate(l,Hs,Hw,Vs,Vw): # with given values, find x, the angles of refraction, time to traverse the refracted path, and time to traverse the straight path
    V = Vw/Vs # the ratio between the speeds occurs often within the quartic equation for x, so it is calculated once separately
    # use sympy.solvers to solve the quartic equation for x
    S = solve((x**4)*(1-V**2) - 2*l*(x**3)*(1-V**2) + (x**2)*(Hs**2-(V**2)*(Hw**2)+(l**2)*(1-V**2)) - 2*l*(Hs**2)*x + (Hs**2)*(l**2), x)
    # solve can yield multiple real positive solutions so a candidate test is needed
    best_T = 9e18 # set a ludicrously high best time so that it is guaranteed to be reduced
    best_x = 0 # set a throwaway best x candidate
    for s in S: # for each candidate solution s in the list of solutions ...
        try:
            float(s)
            if s >= 0: # check if s is a real positive number
                candidate_T = find_T(s,Hs,Hw,l) # find the time using the candidate s
                if candidate_T < best_T: # set the best time and best x if the candidate s yields a faster time than the
```

```
                ↪ current best
32              best_T = candidate_T
33              best_x = s
34      except: pass
35
36  y = (Hs*l)/(Hs+Hw) # find horizontal distance to the beach if the
        ↪ path was straight
37  if best_T != 9e18: # if there was a valid solution, best_T would be
        ↪  set to something other than 9e18, so if it is not 9e18, a
        ↪ valid solution was found
38      try:
39
40          X = best_x; T = best_T; theta_s = atan(X/Hs)*180/pi; theta_w
                ↪ = atan((l-X)/Hw)*180/pi # find the angles that the
                ↪ lifeguard enters and leaves the beach from the normal
                ↪ in degrees
41          T_prime = find_T(y,Hs,Hw,l) # find the time to traverse the
                ↪ straight path
42          return round(X,3), round(y,3), round(theta_s,3), round(
                ↪ theta_w,3), round(T,3), round(T_prime,3)
43      except: pass
44
45  return None, y, None, None, None, None # if best_T is still 9e18,
        ↪ then no valid solutions were found
46
47 color_lookup = [
48     '#282c34', # background
49     '#c678dd', # purple
50     '#98c379', # green
51     '#e06c75', # red
52     '#61afef', # blue
53     '#d19a66', # orange
54     '#56b6c2', # teal
55     '#e5c07b', # yellow
56     '#abb2bf', # white
57     '#20242d'] # hud
58
59
60 pygame.init()
61 pygame.font.init()
62 p = 4 # unit pixel size
63
64 dir = path.dirname(path.realpath(__file__))
65 font = pygame.font.Font(f'{dir}/font.ttf',p*2)
66
67 ms = 10
```

```
68  mw = 500
69
70  B = 4 # horizontal pad on left
71  C = 4 # vertical pad
72  h = ms*p + mw*p + (2*C+1)*p # 30 m sand + 150 m water + 2 m padding
73  w = 150*p + (2*B+1)*p # 150m + Bm left padding + 1m right padding
74  screen = pygame.display.set_mode((w,h))
75  pygame.display.set_caption(f'Ds : {Ds}, Dw : {Dw}')
76
77  X, Y, _theta_s, _theta_w, T, T_prime = calculate(l,Ds,Dw,Vs,Vw)
78
79  while __name__ == '__main__':
80      screen.fill(color_lookup[9])
81      pygame.draw.rect(screen, color_lookup[7], (p,p,(150+2*B-1)*p,(ms+C)
            ↪ *p))
82      pygame.draw.rect(screen, color_lookup[4], (p,(ms+C)*p,(150+2*B-1)*p
            ↪ ,(mw+C)*p))
83      change = False
84
85      for event in pygame.event.get():
86          if event.type == pygame.QUIT: exit()
87      keys = pygame.key.get_pressed()
88      if keys[pygame.K_ESCAPE]: exit()
89      elif keys[pygame.K_w]: Ds += 1; change = True
90      elif keys[pygame.K_s]: Ds -= 1; change = True
91      elif keys[pygame.K_UP]: Dw -= 1; change = True
92      elif keys[pygame.K_DOWN]: Dw += 1; change = True
93      elif keys[pygame.K_LEFT]: l -= 1; change = True
94      elif keys[pygame.K_RIGHT]: l += 1; change = True
95
96      pos = pygame.mouse.get_pos()
97      if pygame.mouse.get_pressed()[0]:
98          pygame.time.delay(60)
99          l = (pos[0]- p*(B + 1))//p
100         Dw = (pos[1]- p*(ms + 1 + C))//p
101         change = True
102
103     Ds = max(min(ms,Ds),0)
104     Dw = max(min(mw,Dw),0)
105     l = max(min(150,l),0)
106     pygame.draw.circle(screen, color_lookup[0], (p*B,p*(ms+C-Ds)), 10)
            ↪ # lifegaurd
107     pygame.draw.circle(screen, color_lookup[0], (p*(B+l),p*(ms+C+Dw)),
            ↪ 10) # target
108
109     jp_color = color_lookup[0]
```

```
    if change:
        _x, Y, _theta_s, _theta_w, t, t_prime = calculate(l,Ds,Dw,Vs,Vw) # calculate values with the given l
        # somtimes solve does not return valid solutions for l values that should have a solution, those l values are skipped
        if _x != None: # if the distance was valid
            X = _x
            T = t
            T_prime = t_prime
        elif _x == None or not float(_x): jp_color = color_lookup[3]


    pygame.draw.circle(screen, color_lookup[0], (p*(B+X), p*(ms+C)), 10) # jump point
    pygame.draw.circle(screen, color_lookup[0], (p*(B+Y), p*(ms+C)), 10) # straight point

    pygame.draw.line(screen, color_lookup[0], (p*B,p*(ms+C-Ds)), (p*(B+X),p*(ms+C))) # lifeguard to jump point
    pygame.draw.line(screen, color_lookup[0], (p*(B+X),p*(ms+C)), (p*(B+l),p*(ms+C+Dw))) # jump point to target

    pygame.draw.line(screen, color_lookup[3], (p*B,p*(ms+C-Ds)), (p*(B+Y),p*(ms+C))) # lifeguard to straight point
    pygame.draw.line(screen, color_lookup[3], (p*(B+Y),p*(ms+C)), (p*(B+l),p*(ms+C+Dw))) # straight point to target

    pygame.display.set_caption(f'Ds : {Ds}, Dw : {Dw}, l : {l}, optimal time : {T}, straight time : {T_prime}, ?T : {round (T_prime-T,2)}')
    pygame.display.update()
```