```python
import pygame

winLen = 1280
winHeight = 680

pygame.init()
win = pygame.display.set_mode((winLen, winHeight))
pygame.display.set_caption("Pulley Simulation by Lynn Hu")

rad = 20 #radius of circle for button and pulley and motor
vel = 10 #velocity of the pulley when you move it from arrow keys
dt = 0.1 #delta time - time interval between each frame/each time it's
updated
pulley_colour = (255, 0, 0)
rope_colour = (0, 0, 0)
weight_colour = (100, 100, 100)
font = pygame.font.Font(None, 36) #font for slider and title
small_font = pygame.font.Font(None, 24) #font for instructions and
stats/values on the left
time_passed = 0
start_time = pygame.time.get_ticks() #number of milliseconds passed since
running
running = False #run simulation
velocity = 0 #velocity of weight
acceleration = 0 #acceleraiton of weight
show_run = [True] #it shows the 'run' button - not reset
instruction_font = pygame.font.Font(None, 24)
show_instructions = [False] #it doesn't show instructions right now - only
when ? button is clicked
current_step = [0] #list that holds index of current step in instructions.
Tracks which step is currently showing. stores current index of instruction
which refers to a specific instruction
instruction_texts = [
    "Welcome! Type f to place a pulley. use arrow keys to move it around,
type SPACE to set its position. It will create the weight",
    "This is a motor, it has force and pulls on a rope.",
    "Click on two objects to connect them with a rope. The motor must be
connected to the pulley, which connects to the weight to run.",
    "Use the slide bars to change values",
    "Here are live values which will show when running",
    "Click 'Run' to start the simulation, and then reset when you're done.
If it doesn't move, the weight is too heavy so change your values.",
] #list of the instructions for each step

def calculate_net_force(m_force, m_weight):
    return m_force - m_weight*g #W=mg - weight is m_weight*g (g is 9.81 on
Earth) net force is force of motor minus force of weight

def calculate_acceleration(net_force, mass):
    return net_force / mass #F=ma, so a (acceleration) is F (net
force)/m(mass)

def update_velocity(current_velocity, acceleration):
    return max(0, current_velocity + acceleration * dt) #v = u+at
max(0,...) to make sure value doesn't go below 0. this means that the
pulley won't go downwards

class Pulley:
    def __init__(self, x, y): #pulley's initial state
        self.x = x
        self.y = y
        self.is_fixed = False #you can move it
        self.colour = pulley_colour

    def move(self, keys):
        if not self.is_fixed: #if you can move it
            if keys[pygame.K_LEFT] and self.x > rad+ 200: # < key lets go
left until the edge of the white bit
                self.x -= vel #it moves left by velocity (10) in time dt -
this works s=d/t, and it is updated every t. or: new position = initial
position + velocity*time. dt is updated with each frame the rest are all
the same
            if keys[pygame.K_RIGHT] and self.x < winLen - rad - 200:
                self.x += vel
            if keys[pygame.K_UP] and self.y > 100 + rad:
                self.y -= vel
            if keys[pygame.K_DOWN] and self.y < winHeight - rad - 200:
                self.y += vel

    def centre(self):
        return (self.x, self.y)#this is the centre of the circle - used for
ropes - more efficient

    def draw(self, win):
        pygame.draw.circle(win, self.colour, (self.x, self.y), rad) #draws
the circle

class Weight:
    def __init__(self, pulley, winHeight):
        self.width = 40
        self.height = 40
        self.x = pulley.x - self.width // 2 #it aligns with the pulley
        self.y = winHeight - self.height - 205 #it is near the bottom
        self.y_velocity = 0
        self.can_lift = False #if the weight CAN be lifted or not - whether
or not it's connected to the pulley and the motor
        self.lifting = False #if its being lifted - like the variable
running

    def check_lift_condition(self, pulley, ropes): #see of the weight can
be lifted
        connected_to_pulley = False
        pulley_connected_to_motor = False
        for rope in ropes:
            if (rope.obj1 == self and rope.obj2 == pulley) or (rope.obj2 ==
self and rope.obj1 == pulley): #if the rope is connects the weight and the
pulley (obj1 or obj2 vice versa each way)
                connected_to_pulley = True
            if (rope.obj1 == pulley and rope.obj2 == motor) or (rope.obj2
== pulley and rope.obj1 == motor): #if rope connects pulley to motor - this
is required to lift the weight
                pulley_connected_to_motor = True
        self.can_lift = connected_to_pulley and pulley_connected_to_motor
#it can lift if it is connected to pulley and motor

    def lift(self): #if this function is called, then the weight lifts
        global running, start_time #global variables for simplicity
        if self.can_lift:
            self.lifting = True #it will lift if it is possible
            running = True #then the simulation runs
            start_time = pygame.time.get_ticks() #it finds what the current
time is (this is used in the live stats)
```

```
            show_run[0] = False #it no longer shows the run button - it 
shows reset button

    def update(self):
        global running, time_passed, velocity, acceleration
        if self.lifting:
            net_force = calculate_net_force(motorForce, weightForce)#finds 
net force from motorForce and weightForce - defined by slider values
            acceleration = calculate_acceleration(net_force, weightForce)
#finds acceleration
            self.y_velocity = update_velocity(self.y_velocity, 
acceleration) #finds velocity that it needs to be - updates velocity
            self.y -= self.y_velocity#changes  y value (moves weight up) 
based one velocity
            velocity = self.y_velocity #can be used globally
            if self.y < pulley.y + rad:#if the weight is not below the 
pulley
                self.y = pulley.y + rad#sets the y values of the weight - 
makes it so that it stops moving
                self.y_velocity = 0#stops moving
                self.lifting = False#is not lifting
                running = False#simulation is not running
                time_passed = (pygame.time.get_ticks() - start_time)/ 1000 
#the time is current time - start time. this is set, so that once the 
simulation stops, it still shows it

    def centre(self):
        return (self.x + self.width // 2, self.y + self.height // 2)#find 
centre of weight for efficiency

    def draw(self, win):
        pygame.draw.rect(win, weight_colour, (self.x, self.y, self.width, 
self.height))#draw the weight

class Motor:
    def __init__(self, x, y):
        self.x = x
        self.y = y
#defines where it is
    def centre(self):
        return (self.x, self.y) #this is its centre for the ropes

    def draw(self, win):
        pygame.draw.circle(win, (0, 0, 0), (self.x, self.y), rad) #draw it.

class Rope:
    def __init__(self, obj1, obj2):
        self.obj1 = obj1
        self.obj2 = obj2 #the rope connects 2 objects

    def draw(self, win):
        pos1 = self.obj1.centre() #one side is the centre of one object
        pos2 = self.obj2.centre() #other side is centre of other object
        pygame.draw.line(win, rope_colour, pos1, pos2, 5) #draw ropes

class Slider:
    def __init__(self, x, y, width, min_val, max_val, initial_val):
        self.x = x
        self.y = y
        self.width = width
        self.height = 10
        self.knob_radius = 8 #these are for what the slider looks like
        self.min_val = min_val #smallest value it can be
        self.max_val = max_val#largest value
        self.value = initial_val#starts with initial_val - that's what it
shows before you change anything
        self.knob_x = self.x + (self.value - self.min_val) / (self.max_val
- self.min_val) * self.width#calculates the position of the knob: range of
value - the size of the value, divided by the total range - the percentage
of the total to find the value of the knob, since it's multiplied by the
width
        self.dragging = False #when the user clicks on it and drags it,
then it will become True.

    def handle_event(self, event): #for interaction
        if event.type == pygame.MOUSEBUTTONDOWN: #if clicked down
            if abs(self.knob_x - event.pos[0]) <= self.knob_radius and
abs(self.y - event.pos[1]) <= self.knob_radius:#if the position of the
mouse is in the horizontal distance of the knob while clicked down
                self.dragging = True
        elif event.type == pygame.MOUSEBUTTONUP:#if no longer clicked down
            self.dragging = False
        elif event.type == pygame.MOUSEMOTION and self.dragging:#if
dragging (mouse is clicked down and on the knob) and moving
            self.knob_x = max(self.x, min(event.pos[0], self.x +
self.width))#the knob changes position to where the mouse is
            self.value = int(self.min_val + (self.knob_x - self.x) /
self.width * (self.max_val - self.min_val))#finds value knob on slider
represents as an integer: find a percentage of how far along the knob is
and then multiple by number range to find the value

    def draw(self, win, val_name, unit):
        pygame.draw.rect(win, (255, 255, 255), (self.x, self.y, self.width,
self.height))#the bar to slide on
        pygame.draw.circle(win, (50, 100, 255), (int(self.knob_x), self.y +
self.height // 2), self.knob_radius)#the knob
        text_surface = font.render(f"{val_name}: {self.value} {unit}",
True, (0, 0, 0))#where the text is, next to the bar
        win.blit(text_surface, (self.x + self.width + 10, self.y -
10))#draw that text

class Button:
    def __init__(self, x, y, radius, text, colour, hover_colour, action):
        self.x = x
        self.y = y
        self.radius = radius
        self.text = text
        self.colour = colour
        self.hover_colour = hover_colour#colour when mouse it above it
        self.action = action#what it will do - what function it will call
        self.font = pygame.font.Font(None, 28)

    def draw(self, win):
        mouse_pos = pygame.mouse.get_pos()#where the mouse is
        dist = ((mouse_pos[0] - self.x) **2 + (mouse_pos[1] - self.y) **2)
**0.5#distance between mouse and centre of button using pythagoras' theorem
        current_colour = self.hover_colour if dist <=self.radius else
self.colour#the colour of the button is hover_colour if the mouse is on the
button/if the mouse is within the radius of the button's centre.
        pygame.draw.circle(win, current_colour, (self.x, self.y),
self.radius) #draws the button
```

```
        text_display = self.font.render(self.text, True, (255, 255, 
255))#the text in the button
        text_rect = text_display.get_rect(center=(self.x, self.y))#centres 
the text in the button by creating a rectangle
        win.blit(text_display, text_rect)#shows this

    def handle_event(self, event):
        if event.type == pygame.MOUSEBUTTONDOWN:#if mouse is clicked
            mouse_pos = event.pos
            dist = ((mouse_pos[0] - self.x) ** 2 + (mouse_pos[1] - self.y) 
** 2) ** 0.5#how far the mouse is from the centre of the button
            if dist <= self.radius:#if the mouse is in the button - while 
it is clicked
                self.action()#then call the function


def reset():
    global pulley, weight, ropes, time_passed, start_time, running, 
velocity, acceleration
    pulley = None
    weight = None
    ropes = []
    time_passed = 0
    start_time = pygame.time.get_ticks()
    running = False
    velocity = 0
    acceleration = 0
    show_run[0] = True #everything is set to nothing - reset back to 
original where there was nothing


def run_lift():
    if weight:
        weight.check_lift_condition(pulley, ropes)#see if the weight CAN be 
lifted
        weight.lift()#lifts it with function defined in class weight.


def show_help():
    show_instructions[0] = True
    current_step[0] = 0#the instruction steps start from the beginning


def instructions_events(event):#event is what happens - the action that 
needs to be handled
    global current_step
    if show_instructions[0]:
        result = instruction_popups[current_step[0]].handle_event(event)
        if result == 'next':#if the next button is clicked
            current_step[0] += 1#goes to next step
            if current_step[0] >= len(instruction_popups):#if it's the last 
one
                show_instructions[0] = False#don't show instructions
        elif result == 'close':
            show_instructions[0] = False


class The_Instructions:
    def __init__(self, text, pos):
        self.text = text
        self.rect = pygame.Rect(pos[0], pos[1], 300, 140)#the rectangle
with the text in it
        self.next_button_rect = pygame.Rect(self.rect.x + 240, self.rect.y
+ 105, 50, 25)#next button rectangle
        self.close_button_rect = pygame.Rect(self.rect.x + self.rect.width
- 35, self.rect.y + 5, 25, 25)#close button rectangle

    def draw(self, win):
        pygame.draw.rect(win, (255,255,255), self.rect)#draw white
rectangle
        pygame.draw.rect(win, (0,0,0), self.rect, 2)#black rectangle border

        wrapped_lines = self.wrap_text(self.text, small_font,
self.rect.width - 20)#makes a margin around the text
        for i, line in enumerate(wrapped_lines):#goes through each line
            text_surface = small_font.render(line, True, (0,0,0))
            win.blit(text_surface, (self.rect.x + 10, self.rect.y + 35 + i
* 20))#show the text

        bottom_text = 'Next' if current_step[0] < len(instruction_popups) -
1 else 'Done'#if it is not the last instruction, it says 'Next', otherwise
'Done'
        pygame.draw.rect(win, (0, 255, 0), self.next_button_rect)#draws the
small button
        next_text = small_font.render(bottom_text, True, (255, 255, 255))
        win.blit(next_text, (self.next_button_rect.x + 5,
self.next_button_rect.y + 5))#shows the text

        pygame.draw.rect(win, (255, 0, 0), self.close_button_rect,
border_radius=5)#draws close button
        x_text = small_font.render("X", True, (255, 255, 255))
        win.blit(x_text, (self.close_button_rect.x + 5,
self.close_button_rect.y + 1))#draws the text for it

    def wrap_text(self, text, font, max_width):#turns long text string into
some lines so it fits
        words = text.split(' ')#turns the text into a list of words
        lines = []
        current_line = ''
        for word in words:
            test_line = current_line + word + ' '#add the next word to the
line
            if font.size(test_line)[0] <= max_width:#if the line as it is
now fits in max width (space available for the text to be in the box)
                current_line = test_line#if the line fits, the word is
added to the line
            else:
                lines.append(current_line)#if it doesn't fit, it is added
to the lines list -> next line
                current_line = word + ' '#new line is started with the
current word
        lines.append(current_line)#line is appended to lines
        return lines

    def handle_event(self, event):
        if event.type == pygame.MOUSEBUTTONDOWN:
            if self.next_button_rect.collidepoint(event.pos):#if the mouse
clicked on the next button
                return 'next'
            if self.close_button_rect.collidepoint(event.pos):#if it
clicked on the close button
```

```
                return 'close'
        return None

popup_positions = [
    (800, 180),
    (100, 300),
    (100, 450),
    (10, 400),
    (10, 300),
    (850, 450),
]#where the pop-up step-by-step instructions are
instruction_popups = [The_Instructions(text, pos) for text, pos in
zip(instruction_texts, popup_positions)]#puts pop_up positions (where the
instructions are) and the text for the instruction together in pairs in a
list as tuples

pulley = None
weight = None
motor = Motor(200+rad, 250)#makes the motor in this position
motorSlider = Slider(250, 640, 420, 1, 800, 12)
weightSlider = Slider(250, 600, 420, 1, 80, 1)
gravitySlider = Slider(250, 560, 420, 1, 30, 10) #defining the sliders
ropes = [] #list of ropes
selected_object = None
run_button = Button(1190, 600, 40, 'Run', (0,150,0), (0, 200, 0), run_lift)
reset_button = Button(1190, 600, 40, "Reset", (200, 0, 0), (255, 50, 50),
reset)
help_button = Button(1200, 60, 30, '?', (91, 167, 199), (2, 75, 107),
show_help) #defining the buttons

run = True
while run:
    pygame.time.delay(10)#delay at start of each iteration of loop
    keys = pygame.key.get_pressed()#checks state of keys

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            run = False#if clicked quit button, the run is False, so
everything here stops

        instructions_events(event)
        motorSlider.handle_event(event)
        weightSlider.handle_event(event)
        gravitySlider.handle_event(event)
        help_button.handle_event(event)#makes everything be able to have
interactions

        if show_run[0]:
            run_button.handle_event(event)#if it is show_run - if there is
the run button, then let the run_button handle event
        else:
            reset_button.handle_event(event)#otherwise the reset button
handles events

        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_f and pulley is None:#if there is no
pulley and 'f' is typed
                pulley = Pulley(500+rad, 200)#make a pulley here
            elif event.key == pygame.K_SPACE and pulley and not
pulley.is_fixed:#if SPACE is typed and pulley is not fixed yet
                pulley.is_fixed = True#then fix the pulley
                weight = Weight(pulley, winHeight)#make a weight

        if event.type == pygame.MOUSEBUTTONDOWN:
            mouse_x, mouse_y = pygame.mouse.get_pos()
            mouse_pos = event.pos#find where the mouse is where it clicked
            clicked_object = None

            if pulley and (pulley.x - mouse_x) ** 2 + (pulley.y - mouse_y)
** 2 <= rad ** 2:#if mouse is on pulley
                clicked_object = pulley
            elif weight and weight.x <= mouse_x <= weight.x + weight.width
and weight.y <= mouse_y <= weight.y + weight.height:#if mouse is on weight
                clicked_object = weight
            elif (motor.x - mouse_x) ** 2 + (motor.y - mouse_y) ** 2 <= rad
** 2:#if mouse is on motor
                clicked_object = motor

            if selected_object is None and clicked_object:#if nothing is
selected already, and something is clicked on
                selected_object = clicked_object#that becomes the selected
object
            elif selected_object and clicked_object and selected_object !=
clicked_object:#if 2 different objects have been clicked on
                ropes.append(Rope(selected_object, clicked_object))#make a
rope to join the objects
                selected_object = None#reset so a new object needs to be
selected again

    win.fill((255, 255, 255))#screen is white
    pygame.draw.rect(win, (200, 200, 200), (0, 0, winLen, 100))
    pygame.draw.rect(win, (200, 200, 200), (0, 0, 200, winHeight))
    pygame.draw.rect(win, (200, 200, 200), (0, 480, winLen, 200))
    pygame.draw.rect(win, (200, 200, 200), (1080, 0, 200, winHeight)) #this
is grey border for where UI is - separates from middle pulley simulation
bit

    text = font.render("Pulley Simulation", True, (0, 0, 0))
    win.blit(text, (winLen // 2 - text.get_width() // 2, 30))#the title


    if pulley:
        pulley.move(keys)
        pulley.draw(win) #if there is a pulley draw it, and make it so that
it can be moved

    if weight:
        weight.update()
        weight.draw(win)#if there is a weight then draw it and make sure it
moves

    motor.draw(win)#draw motor

    for rope in ropes:
        rope.draw(win)#draw all ropes
    motorForce = motorSlider.value
    weightForce = weightSlider.value
    g = gravitySlider.value #these values are all defined as whatever the
sliders say.

    motorSlider.draw(win, 'Motor Force', 'N')
    weightSlider.draw(win, 'Weight Mass', 'kg')
    gravitySlider.draw(win, 'Gravitational Acceleration', 'N') #draws the
sliders and what they show

    if running:
        time_passed = (pygame.time.get_ticks() - start_time) /1000#if the
simulation is running, show the current time passed - live stat of how much
time has passed since the simulation started (since run was clicked)
divided by 1000 for 3dp
    time_text = small_font.render(f'time: {time_passed: .3f} s', True,
(0,0,0))
    win.blit(time_text, (5, 150))
    velocity_text = small_font.render(f'velocity: {velocity: .3f}m/s',
True, (0,0,0))
    win.blit(velocity_text, (5, 200))
    acceleration_text = small_font.render(f'acceleration:
{acceleration: .3f} m/s²', True, (0,0,0))
    win.blit(acceleration_text, (5, 250)) #draws the stats on the left and
what they show

    if show_run[0]:
        run_button.draw(win) #if show_run[0] is True, then draw the run
button
    else:
        reset_button.draw(win)#otherwise draw the reset button

    help_button.draw(win)#draw help button

    if show_instructions[0] and current_step[0] <
len(instruction_popups):#if instructions are showing and the step they're
one is still within the number of instructions
        instruction_popups[current_step[0]].draw(win)#draw the current
instruction

    pygame.display.update()#displays whatever since the last frame

pygame.quit()
```