Moving barrel down ramps

line from 1.378,8.728 to 7.677,9.122 to 7.677,9.319 to 1.378,8.925 to 1.378,8.728 line from 1.377,4.804 to 7.676,5.197 to 7.676,5.394 to 1.377,5.000 to 1.377,4.804 line from 0.590,7.535 to 0.590,7.338 to 6.889,6.748 to 6.889,6.944 to 0.590,7.535 box with .sw at (2.362,7.154) width 0.787 height 1.772 box with .sw at (5.315,5.185) width 0.787 height 1.772 “r1r” at 7.480,9.549 ljust “r1l” at 1.181,9.155 ljust “r2l” at 0.591,7.777 ljust “r2r” at 6.890,7.187 ljust “r3r” at 7.677,5.612 ljust “r3l” at 1.181,5.218 ljust “l1t” at 2.559,8.565 ljust “l2t” at 5.512,6.596 ljust “l1b” at 2.559,7.581 ljust “l2b” at 5.512,5.415 ljust

Moving barrel down ramps

- create waypoints for the barrels
  - place waypoints into a dictionary (called `points`)
  - these will be screen coordinates - which we can extract from the ladder or ramp polygons

- then the barrels can use Bresenham's line algorithm to move between the waypoints

Moving barrel down ramps

- now we need a function to create a route

Moving barrel down ramps

kongroute.py

```
#!/usr/bin/env python3

import random

#
#  the points are:  l1t, l1b, l2t, l2b   (ladder no. top and bottom)
#                   r1l, r1r             (ramp 1 left and right)
#                   r2l, r2r             (ramp 2 left and right)
#                   r3l, r3r             (ramp 3 left and right)
#

#
#  a barrel might go to the end of a ramp or occasionally to
#  a ladder returns a list of way points
#
```

Moving barrel down ramps

```
def gen_route ():
    # moving along top ramp 1
    route = [``r1r'']
    if random.random () < 0.2:
        route += [``l1t'', ``l1b'']  # choose ladder
    else:
        route += [``r1l'', ``r2l'']  # fall off end
    # moving along top ramp 2
    if random.random () < 0.2:
        route += [``l2t'', ``l2b'']
    else:
        route += [``r2r'', ``r3r'']
    # and move along ramp 3
    route += [``r3l'']
    return route

for b in range (7):
    print gen_route ()
```

Moving barrel down ramps

```
$ python3 kongroute.py
['r1r', 'r1l', 'r2l', 'l2t', 'l2b', 'r3l']
['r1r', 'r1l', 'r2l', 'r2r', 'r3r', 'r3l']
['r1r', 'r1l', 'r2l', 'r2r', 'r3r', 'r3l']
['r1r', 'r1l', 'r2l', 'l2t', 'l2b', 'r3l']
['r1r', 'r1l', 'r2l', 'r2r', 'r3r', 'r3l']
['r1r', 'l1t', 'l1b', 'r2r', 'r3r', 'r3l']
['r1r', 'r1l', 'r2l', 'r2r', 'r3r', 'r3l']
```

- we can see random routes are chosen
- both ladder 1 and ladder 2 are rejected and chosen
- the function/method `random.random()` returns a floating point number in the range `0.0` to `1.0`

Main function

```
def main ():
    global screen
    pygame.init ()
    screen = pygame.display.set_mode ([width, height])
    draw_scene (gradient)
    play_game (screen)
    wait_for_event ()

main ()
```

play_game

```
def play_game (screen):
    o = -1
    while True:
        t = pygame.time.get_ticks()
        if o != t:
            activity_scheduler (t)
            o = t
        checkInput()
        screen.fill([0, 0, 0]) # blank the screen.
        draw_polygons ()
        for b in barrels:
            b.update (t, 0, width)
            screen.blit (b.image, b.rect)
        pygame.display.flip ()
```

Points of interest

- `pygame.time.get_ticks()` returns the time in the number of milliseconds
- `screen.fill([0, 0, 0])` blank out compete screen
  - then redraw everything
- `barrels` is a list of barrels
  - when a barrel is deleted it is removed from this list

activity_scheduler

```
# there are 1000 ticks per second in pygame
activity_list = [[2000, 0.5, create_new_barrel],
                 [1000, 1.0, display_time],
                 [120000, 1.0, finish_game]]

def activity_scheduler (ticks):
    global activity_list
    for e in activity_list:
        if (ticks % e[0] == 0) and (random.random () <= e[1]):
            e[2] (ticks)
```

activity_scheduler

- describes a way of encoding when a function should be executed

- in the example above we attempt to call `create_new_barrel` every 2 seconds
  - but the program only calls this function if `random.random()` is >= 0.5
  - giving a probability of $\frac{1}{2}$

activity_scheduler

- the `activity_list` specifies that `display_time` is called every second
- `finish_game` is called in 2 minutes
- notice that it is possible that `finish_game` might not be called!
  - the call to `pygame.time.get_ticks()` might miss this tick (due to the operating system running something else)

activity_scheduler

- this approach is very useful as it allows for easy experimentation
- it also allows the program to change the rate or probability depending upon circumstance

check_input

```
def checkInput():
    for event in pygame.event.get():
       if event.type == KEYDOWN:
          if event.key == K_ESCAPE:
              sys.exit(0)
          elif event.key == K_RIGHT:
              print ``right cursor pressed''
              # do_right
          elif event.key == K_LEFT:
              print ``left cursor pressed''
              # do left
          elif event.key == K_UP:
              print ``up cursor pressed''
              # do_jump
```

check_input

- the above are placeholders to make Mario jump or move

Barrel sprites

```
class barrel_sprite (pygame.sprite.Sprite):
    image = None

    def __init__ (self):
        pygame.sprite.Sprite.__init__(self)
        if barrel_sprite.image is None:
            barrel_sprite.image = pygame.image.load (``barrel.png'').convert ()
        self.image = barrel_sprite.image
        self.radius = barrel_sprite.image.get_height()
        self.rect = self.image.get_rect()
        self.route = gen_route ()
        self.rect.topleft = points[self.route[0]]
        self.next_update_time = 0 # update() hasn't been called yet.
        self.nav = None
        self.hop_goal = 0
```

Barrel sprites

- `points` is a dictionary of our way points
  - `points[``l1t''']` gives a coordinate (list) of an x and y value for the top of ladder 1

- `gen_route()` returns the random route list which we covered in the earlier slides

- `self.nav` will contain the Bresenham's object which is instantiated when we call `p2pnav.walk_along` (seen in the next side)

- `self.goal` determines which waypoint this barrel is moving towards
- `self.rect.topleft = points[self.route[0]]` assigns the initial position to this sprite

Barrel sprites

```
    def update (self, current_time, left, right):
        global barrels
        # Update every 10 milliseconds = 1/100th of a second.
        if self.next_update_time < current_time:
            if self.nav == None or self.nav.finished ():
                if self.hop_goal == len (self.route)-1:
                    # finished all routes, delete ourself
                    self.kill ()
                    barrels.remove (self)
                else:
                    # move onto next route
                    self.nav = p2pnav.walk_along (self.get_point (self.hop_goal),
                                                  self.get_point (self.hop_goal+1))
                    self.hop_goal += 1
            self.rect.topleft = self.nav.get_next ()
            self.next_update_time = current_time + 10
```

get_point

- the method `get_point` is needed to adjust the waypoints slightly to take into account the barrel image size
- left points need to be adjusted leftwards so that the barrels fall off the edge rather than drop though the floor
- the ladder bottom point needs adjusting updates so that the barrel rests on the floor
- the ramp height is adjusted so that the barrel appears to roll along the ramp
- it is better to adjust the values in this method as it takes into consideration the sprite image size

get_point

```
    def get_point (self, goal):
        if self.route[goal][-1] == 'b':
            # bottom of the ladder is adjusted upwards
            return [points[self.route[goal]][0],
                    points[self.route[goal]][1]-self.radius]
        elif self.route[goal][-1] == 'l':
            # left ramp way point is adjusted, so it falls off edge
            x = points[self.route[goal]][0]-self.radius/2
        elif self.route[goal][-1] == 'r':
            # right ramp way point is adjusted, so it falls off edge
            x = points[self.route[goal]][0]-self.radius/2
        else:
            x = points[self.route[goal]][0]
        # we do adjust the ramp height, to offset the circle height
        return x, points[self.route[goal]][1]-self.radius/2
```

Homework/tutorial work

- download this code and study it
- comment each function/method/class
- change the code so that you have
  - smaller barrels
  - more ramps and more ladders
- consider how you might introduce Mario as a sprite

Homework/tutorial work

```
#!/usr/bin/env python3

import pygame, sys, time, random, bres
from pygame.locals import *

ramp_one, ramp_two, ramp_three = None, None, None

wood_light = (166, 124, 54)
wood_dark = (76, 47, 0)
blue = (0, 100, 255)
dark_red = (166, 25, 50)
dark_green = (25, 100, 50)
dark_blue = (25, 50, 150)
black = (0, 0, 0)
white = (255, 255, 255)
ladder_colour = (58, 112, 106)
```

Homework/tutorial work

```
width, height = 1024, 768
screen = None
ramp_height = 0.03
ramp_length = 0.85
ladder_height = 0.3
ladder_length = 0.07
gradient = 32
points = {}
debugging = False
barrels = []
```

Homework/tutorial work

```
#
#  the points are:  l1t, l1b, l2t, l2b   (ladder no. top and bottom)
#                   r1l, r1r             (ramp 1 left and right)
#                   r2l, r2r             (ramp 2 left and right)
#                   r3l, r3r             (ramp 3 left and right)
#


#
#  a barrel might go to the end of a ramp or occasionally to a ladder
#  returns a list of way points
#
```

Homework/tutorial work

```
def gen_route ():
    # moving along top ramp 1
    route = [``r1r'']
    if random.random () < 0.2:
        route += [``l1t'', ``l1b'']  # choose ladder
    else:
        route += [``r1l'', ``r2l'']  # fall off end
    # moving along top ramp 2
    if random.random () < 0.2:
        route += [``l2t'', ``l2b'']
    else:
        route += [``r2r'', ``r3r'']
    # and move along ramp 3
    route += [``r3l'']
    return route
```

Homework/tutorial work

```
class barrel_sprite (pygame.sprite.Sprite):
    image = None

    def __init__ (self):
        pygame.sprite.Sprite.__init__(self)
        if barrel_sprite.image is None:
            barrel_sprite.image = pygame.image.load (``barrel.png'').convert ()
        self.image = barrel_sprite.image
        self.radius = barrel_sprite.image.get_height()
        self.rect = self.image.get_rect()
        self.route = gen_route ()
        self.rect.topleft = points[self.route[0]]
        self.next_update_time = 0 # update() hasnt been called yet.
        self.nav = None
        self.hop_goal = 0
```

Homework/tutorial work

```
    def update (self, current_time, left, right):
        global barrels
        # Update every 10 milliseconds = 1/100th of a second.
        if self.next_update_time < current_time:
            if self.nav == None or self.nav.finished ():
                if self.hop_goal == len (self.route)-1:
                    # finished all routes, delete ourself
                    self.kill ()
                    barrels.remove (self)
                else:
                    # move onto next route
                    self.nav = bres.walk_along (self.get_point (self.hop_goal),
                                                self.get_point (self.hop_goal+1))
                    self.hop_goal += 1
            self.rect.topleft = self.nav.get_next ()
            self.next_update_time = current_time + 10
```

Homework/tutorial work

```
    def get_point (self, goal):
        if self.route[goal][-1] == ``b'':
            # bottom of the ladder is adjusted upwards
            return [points[self.route[goal]][0],
                    points[self.route[goal]][1]-self.radius]
        elif self.route[goal][-1] == ``l'':
            # left ramp way point is adjusted, so it falls off edge
            x = points[self.route[goal]][0]-self.radius/2
        elif self.route[goal][-1] == ``r'':
            # right ramp way point is adjusted, so it falls off edge
            x = points[self.route[goal]][0]-self.radius/2
        else:
            x = points[self.route[goal]][0]
        # we do adjust the ramp height, to offset the circle height
        return x, points[self.route[goal]][1]-self.radius/2
```

Homework/tutorial work

```
def xpos (v):
    global height
    return (int) (width*v)

def ypos (v):
    global width
    return (int) (height*v)

def draw_ramp (xoffset, yoffset, left_drop, right_drop):
    global ramp_length, ramp_height
    top_left = [xpos (xoffset), ypos (yoffset)+left_drop]
    top_right = [xpos (xoffset+ramp_length), ypos (yoffset)+right_drop]
    bot_right = [xpos (xoffset+ramp_length), ypos (yoffset+ramp_height)+right_drop]
    bot_left = [xpos (xoffset), ypos (yoffset+ramp_height)+left_drop]
    return pygame.draw.polygon (screen, wood_dark, [top_left, top_right, bot_right, bot
```

Homework/tutorial work

```
def draw_ramps (drop):
    return [draw_ramp (0.1, 0.16, drop, 0),
            draw_ramp (0.03, 0.48, 0, drop),
            draw_ramp (0.1, 0.80, drop, 0)]

def wait_for_event ():
    while True:
        event = pygame.event.wait()
        if event.type == pygame.QUIT:
            sys.exit(0)
        if event.type == KEYDOWN:
            if event.key == K_ESCAPE:
                sys.exit (0)
```

Homework/tutorial work

```
def add_points (ladders, ramps, gradient):
    global points
    for i, l in enumerate (ladders):
        top = ''l%dt'' % (i+1)
        bot = ''l%db'' % (i+1)
        print top, bot
        points[top] = [l.left, l.top-ypos (.045)]
        points[bot] = [l.left, l.bottom-ypos (.019)]
    for i, l in enumerate (ramps):
        left = ''r%dl'' % (i+1)
        right = ''r%dr'' % (i+1)
        print left, right
        if i % 2 == 0:
            points[left] = [l.left, l.top]
            points[right] = [l.right, l.top-gradient]
        else:
            points[left] = [l.left, l.top-gradient]
            points[right] = [l.right, l.top]
```

Homework/tutorial work

```
def draw_scene (gradient):
    global list_of_polygons
    for i in range (gradient):
        draw_ramps (i)
        pygame.display.flip ()
        screen.fill (black)
        if not debugging:
            time.sleep (.2)
    l = draw_ladders (gradient)
    r = draw_ramps (gradient)
    list_of_polygons = l + r
    pygame.display.flip ()
    add_points (l, r, gradient)
    print points
```

Homework/tutorial work

```
def draw_ladder (x, y, drop):
    global ladder_length, ladder_height
    top_left = [xpos (x), ypos (y)+drop]
    top_right = [xpos (x+ladder_length), ypos (y)+drop]
    bot_right = [xpos (x+ladder_length), ypos (y+ladder_height)+drop]
    bot_left = [xpos (x), ypos (y+ladder_height)+drop]
    return pygame.draw.polygon (screen, ladder_colour, [top_left, top_right, bot_right,

def draw_ladders (drop):
    return [draw_ladder (0.2, 0.16, drop),
            draw_ladder (0.7, 0.48, drop)]
```

Homework/tutorial work

```
def activity_scheduler (ticks):
    global activity_list
    for e in activity_list:
        if (ticks % e[0] == 0) and (random.random () <= e[1]):
            e[2] (ticks)

def create_new_barrel (ticks):
    global barrels
    barrels += [barrel_sprite ()]

def display_time (ticks):
    print ``time is'', ticks/100
    pass
```

Homework/tutorial work

```
def finish_game (ticks):
    print ``game over''
    sys.exit (0)

# there are 1000 ticks per second in pygame
activity_list = [[2000, 0.5, create_new_barrel],
                 [1000, 1.0, display_time],
                 [120000, 1.0, finish_game]]
```

Homework/tutorial work

```
def checkInput():
    for event in pygame.event.get():
       if event.type == KEYDOWN:
          if event.key == K_ESCAPE:
              sys.exit(0)
          elif event.key == K_RIGHT:
              print ``right cursor pressed''
              # do_right
          elif event.key == K_LEFT:
              print ``left cursor pressed''
              # do left
          elif event.key == K_UP:
              print ``up cursor pressed''
              # do_jump
```

Homework/tutorial work

```
def draw_polygons ():
    draw_ladders (gradient)
    draw_ramps (gradient)

def play_game (screen):
    o = -1
    while True:
        t = pygame.time.get_ticks()
        if o != t:
            activity_scheduler (t)
            o = t
        checkInput()
        screen.fill([0, 0, 0]) # blank the screen.
        draw_polygons ()
        for b in barrels:
            b.update (t, 0, width)
            screen.blit (b.image, b.rect)
        # pygame.display.update()
        pygame.display.flip ()
```

Homework/tutorial work

```
def main ():
    global screen
    pygame.init ()
    screen = pygame.display.set_mode ([width, height])
    draw_scene (gradient)
    play_game (screen)
    wait_for_event ()

main ()
```

barrel image

- here is the very simple image used to test the above code
  ⟨`barrel.png`⟩

- the module `bres.py` can be found at the end of the previous weeks notes
  - it needs to present in the same directory as the file above
  - together with the ⟨`barrel.png`⟩ file