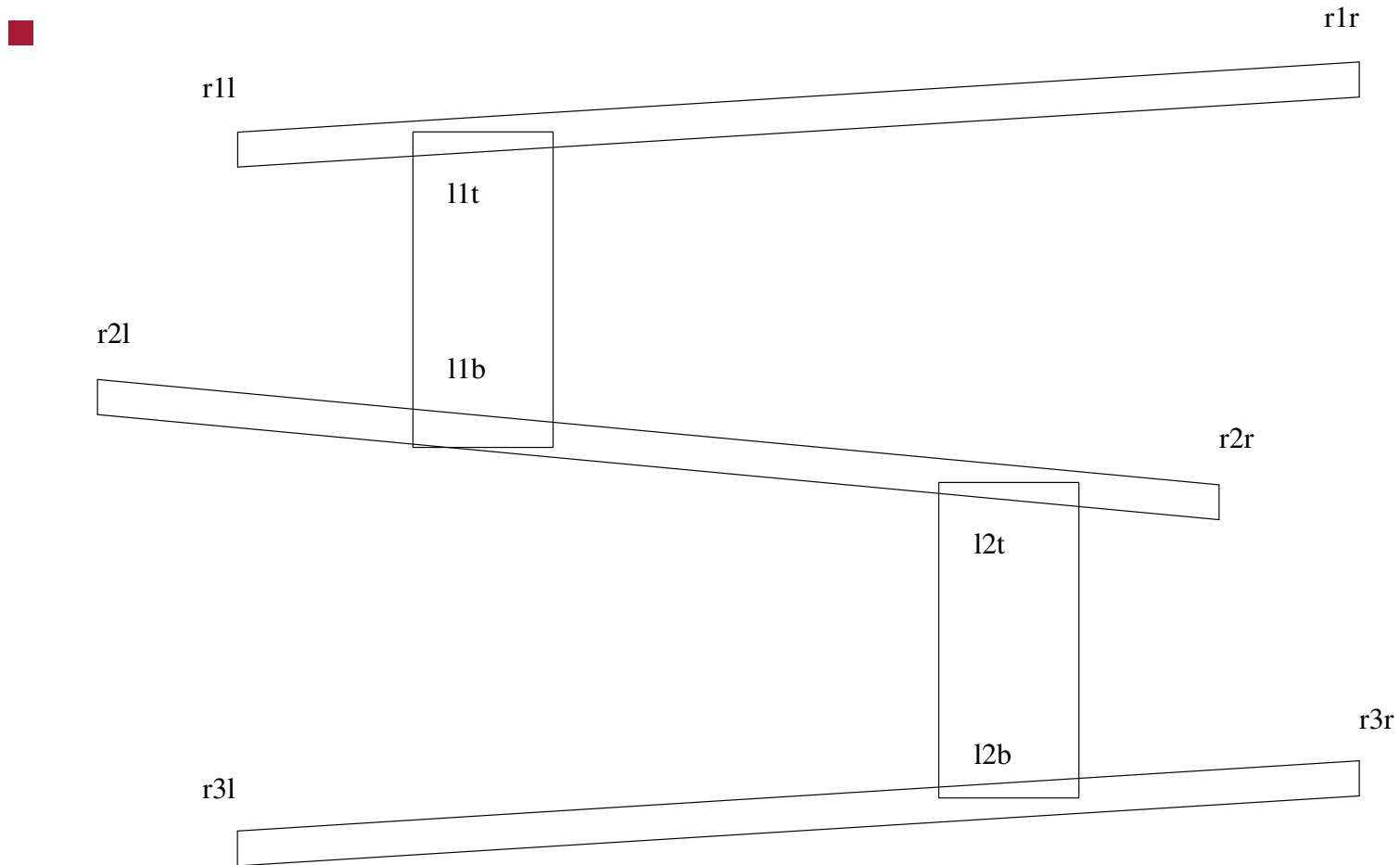


Python Pygame: Mario movement



Python Pygame: Mario movement

- Mario requires the movement
 - along ramps
 - up ladders
 - up to next ramp and down to lower ramp, when he reaches the end

- ideally he should be able to jump off ladders!
 - left as an exercise for the reader

- Mario also needs the ability to jump
 - left as an exercise for the reader

Python Pygame: Mario movement

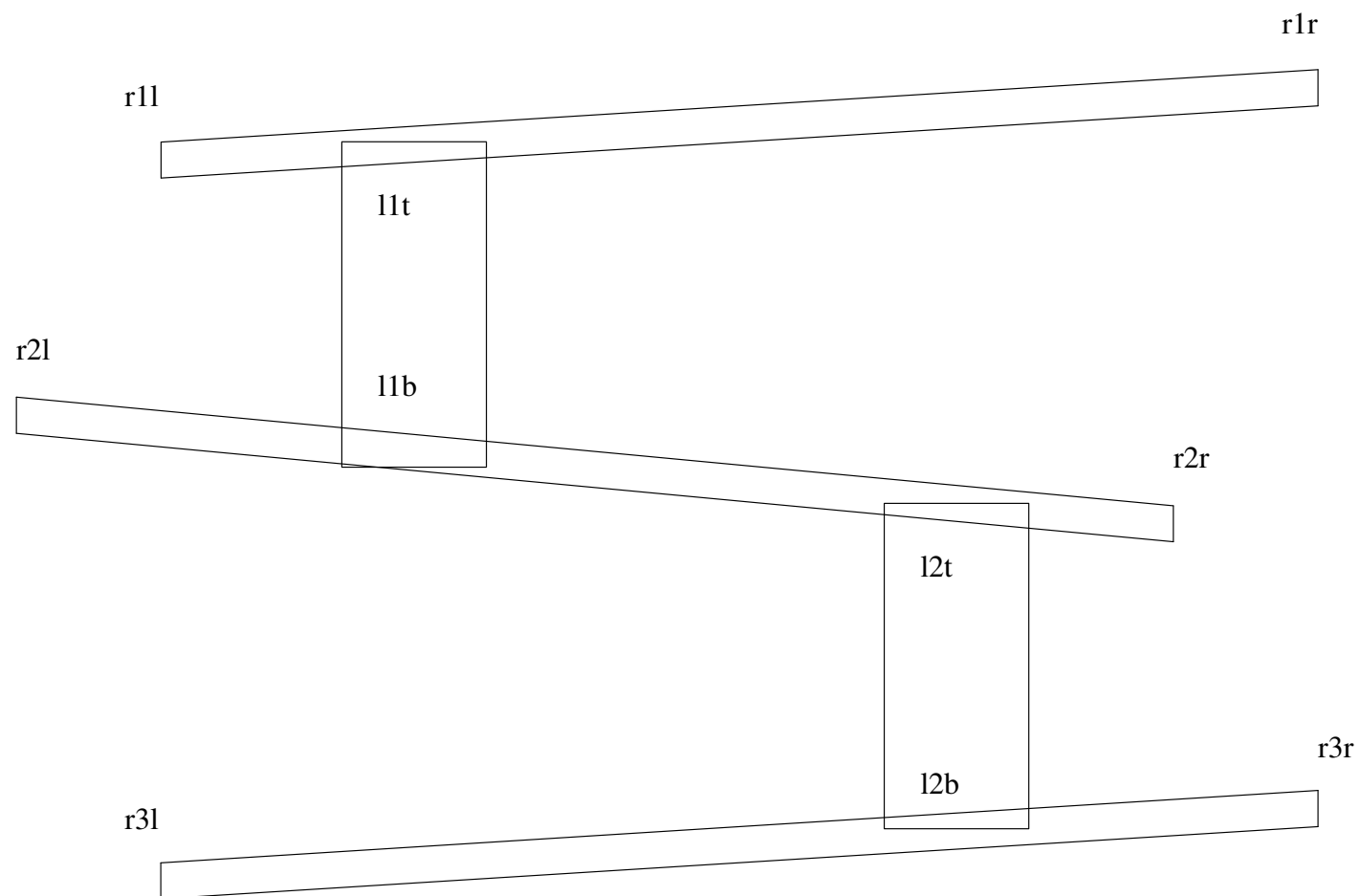
- one solution is to put Mario on rails
 - he can change direction (or path at the end of the current path)
 - he can reverse direction at any time
 - he needs the ability to choose a ladder

- placing Mario on rails is just one solution
 - another might be to use sprites for ramps and ladders and detect collisions

Mario on rails

- in Computer Science we often have the tradeoff between complex data structures or complex code
- adding a little complexity to the data structures will reduce the complexity of the code
- define a map for Mario, map is a dictionary of paths
 - at each end point in the Mario diagram we have a path for any chosen direction

Mario on rails



Mario on rails

- starting at r3l we note:

- he cannot move up
- he can move right towards r3r he will pass ladder 12b
- he cannot move down
- if he moves left he dies

- ```
map = { ``r3l-0``: None, # up
 ``r3l-1``: [``r3r``, [``12b``]], # right
 ``r3l-2``: None, # down
 ``r3l-3``: [``d3``, []], # left
 ...
```

- where

- pointname-0 is up, pointname-1 is right, etc
- if the path exists it is a list

## Mario on rails

- when he reaches `r3r` his choices are:
  - up to ramp 2
  - back to `r3l`
- he cannot go down and he cannot go right

```
``r3r-0``: [``r2r``, []], # up
``r3r-1``: None, # right
``r3r-2``: None, # down
``r3r-3``: [``r3l``, [``l2b``]], # left
```

## Path list

- all path lists must be entered into the dictionary map
  - however if a path is not an option for Mario then its value in the dictionary is None
  
- any non None path will consist of the following entries:
  - first element is the furthest destination way point
  - the second element is also a list of optional ladders



## Consider paths for ramp 2

- ```
``r2r-0``: None,                # up
``r2r-1``: None,                # right
``r2r-2``: [``r3r``, []],        # down
``r2r-3``: [``r2l``, [``l2t``, ``l1b``]], # left
```
- he cannot go up or right from point r2r
 - he can go down to r3r
 - and he can move left to r2l and optionally chose ladders l2t or l1b

Consider paths for ramp 2

- and if he reaches point r2l

- | |
|---|
| <pre>``r2l-0``: [``r1l``, []], # up ``r2l-1``: [``r2r``, [``l2t``, ``l1b``]], # right ``r2l-2``: None, # down ``r2l-3``: None, # left</pre> |
|---|

- here at point r2l he can move
 - up to r1l
 - right (and return) to r2r possibly choosing ladders l2t and l1b
- he cannot go left or down

Mario sprite class

```
class mario (pygame.sprite.Sprite):
    image = None
    def __init__ (self, o, d, startpos, path):
        pygame.sprite.Sprite.__init__(self)
        mario.image = pygame.image.load (barrel_colour()).convert_alpha ()
        self.images = []
        self.orientation = o
        for i in mario_actions:
            self.images += [pygame.image.load (action_image_names[i]).convert_alpha ()]
        self.image_height = 0
        self.image_width = 0
        self._change (d)
        self.rect = self.image.get_rect()
        self.newpath = path
        startpos = self.adjust (startpos)
        self.route = bres.walk_along (startpos, startpos)
        self.curpos = self.route.get_next ()
        self.rect.topleft = self.curpos
        self.next_update_time = 0
        self.Xspeed = 0
        self.direction = None
        self.path = None
        self.pathname = None
```

Mario sprite class

```
def new_goal (self, d):
    print ``new_goal says our newpath is'', self.newpath
    self.pathname = ``%s-%d'' % (self.newpath, d)
    print ``Mario is using path'', self.pathname,
    path = map[self.pathname]
    print `` ='', path
    if path == None:
        print ``no path to walk along''
        self.route = bres.walk_along (self.curpos, self.curpos)
    else:
        print ``newpath ='', self.newpath
        self.path = self.newpath
        self.newpath = path[0]
        print ``path ='', self.path, ``newpath ='', self.newpath
        endpos = self.adjust (points[self.newpath])
        self.route = bres.walk_along (self.curpos, endpos)
        self.direction = d
```

Mario sprite class

```
def on_ladder (self):
    if self.pathname != None:
        path = map[self.pathname]
        if path != None:
            for l in path[1]:
                print l
                if self.is_on (points[l][0]):
                    return True, l
    return False, self.newpath

def go (self, k):
    if k == K_RIGHT:
        self._horizontal (1, stand_right)
    elif k == K_LEFT:
        self._horizontal (3, stand_left)
    elif k == K_UP:
        self._vertical (0, up_right)
    elif k == K_DOWN:
        self._vertical (2, up_left)
```

Mario sprite class

```
def _horizontal (self, newdir, o):
    if self.direction in [0, 2]:
        # could be going up a ladder or between ramps at the end
        if self.route.finished ():
            # we have reached the end of the ladder or end of the up/down route
            self.orientation = o
            self._change (o)
            self.next_update_time = 0
            self.new_goal (newdir)
    else:
        if self.direction == newdir:
            # same direction, just continue, faster
            self.Xspeed = min (self.Xspeed + step_horizontal, max_speed)
        else:
            self.orientation = o
            self._change (o)
            self.next_update_time = 0
            self.new_goal (newdir)
```

Mario sprite class

```
def _vertical (self, newdir, o):
    if self.direction in [1, 3]:
        # going left or right, check if we can use ladder
        b, self.newpath = self.on_ladder ()
        if b:
            print ``using a ladder``, self.newpath
            self.orientation = o
            self._change (o)
            self.next_update_time = 0
            self.new_goal (newdir)
        elif self.route.finished ():
            # can also go up at the end of the ramp
            self.orientation = o
            self._change (o)
            self.next_update_time = 0
            self.new_goal (newdir)
```


Mario sprite class

```
else:
    # already going up or down, might be on a ladder or end of a ramp
    if self.direction == newdir:
        # same direction, just continue, faster
        self.Xspeed = min (self.Xspeed + step_vertical, max_speed)
    else:
        # change of direction
        self.orientation = o
        self._change (o)
        self.next_update_time = 0
        # check to see if already on ladder
        if (self.pathname != None) and (self.pathname[0] == ``l``):
            # make new goal the previous start
            self.newpath = self.path
            self.new_goal (newdir)
        else:
            self.new_goal (newdir)
```

Mario sprite class

```
def _change (self, d):
    self.image = self.images[d]
    self.image_height = mario.image.get_height()
    self.image_width = mario.image.get_width()
    self.next_update_time = 0
def update (self, current_time):
    if self.next_update_time < current_time:
        if self.Xspeed > 0:
            self.curpos = self.route.get_next ()
            self.rect.topleft = self.curpos
            self.Xspeed -= 1
            self.next_update_time = current_time + 1

def adjust (self, p):
    return [p[0], p[1]-self.image_height]

def is_on (self, x):
    return not ((self.curpos[0] + self.image_width < x) or
                (self.curpos[0] > x + xpos (ladder_width)))
```

Mario sprite class



```
def checkInput():
    for event in pygame.event.get():
        if event.type == KEYDOWN:
            if event.key == K_ESCAPE:
                sys.exit(0)
            elif event.key in [K_RIGHT, K_LEFT, K_UP, K_DOWN]:
                M.go(event.key)
            elif event.key == K_f:
                pygame.display.toggle_fullscreen()
```

Mario sprite class

```
def play_game (screen):  
    global M  
    o = -1  
    M = mario (stand_right, 1, points['`r3l''], ``r3l'')  
    while True:  
        t = pygame.time.get_ticks()  
        if o != t:  
            activity_scheduler (t)  
            o = t  
        checkInput()  
        screen.fill(white) # blank the screen.  
        draw_polygons ()  
        for b in barrels:  
            b.update (t)  
            screen.blit (b.image, b.rect)  
        M.update (t)  
        screen.blit (M.image, M.rect)  
        pygame.display.flip ()
```

Homework and tutorial

- finish the path map definition and integrate the movement into your code
- make Mario jump, fall off ladders
- improve speed of movement and smoothness/playability
- scoring, timing, sounds etc

PGE input

- implementing Mario using the Physics game engine is much easier!
- since the ball representing Mario is free running it just needs to be given a push when we want it to move
- we could
 - push it left with the left mouse button
 - push it right with the right mouse button
 - up with the middle mouse button

PGE input

```
def mouse_hit (e):  
    global m  
    mouse = pge.pyg_to_unit_coord (e.pos)  
    if e.button == 1:  
        m.put_xvel (gb.get_xvel ()-0.3)  
    elif e.button == 3:  
        m.put_xvel (gb.get_xvel ()+0.3)  
    elif gb.moving_towards (mouse[0], mouse[1]):  
        pos = m.get_unit_coord ()  
        # print ``mouse ='', mouse, ``ball ='', pos  
        m.apply_impulse (pge.sub_coord (mouse, pos), 0.4)  
    else:  
        m.put_yvel (m.get_yvel ()+0.4)
```

PGE input

- in the main function we register the mouse event with our function

- ```
pge.register_handler (mouse_hit, [MOUSEBUTTONDOWN])
```

- please see the implementation of breakout to see how this is integrated into a game [breakout example](http://floppsie.comp.glam.ac.uk/Glamorgan/gaius/pge/homepage.html) `<http://floppsie.comp.glam.ac.uk/Glamorgan/gaius/pge/homepage.html>`



## Collisions in PGE

- refering again to the [breakout source code example](http://floppsie.comp.glam.ac.uk/Glamorgan/gaius/pge/example_games.html) `<http://floppsie.comp.glam.ac.uk/Glamorgan/gaius/pge/example_games.html>`
- notice that the section of code containing `delete_me` and `box_of`

## Collisions in PGE

```
def delete_me (o, e):
 global blocks, winner, loser

 blocks.remove (o)
 o.rm ()
 if blocks == []:
 if not loser:
 winner = True
 pge.text (0.2, 0.3, ``Winner``, white, 100, 1)
 pge.at_time (4.0, finish_game)

def box_of (pos, width, height, color):
 global blocks

 blocks += [pge.box (pos[0], pos[1], width, height, color)\
 .fix ().on_collision (delete_me)]
```

## Collisions in PGE

- the function `box_of` creates a blue box at `pos` with a `width` and `height`
- it also stipulates that this box is `fixed`
- furthermore if anything hit this box then the function `delete_me` is called

## Collisions in PGE

- the function `delete_me` is a call back registered by the call to `on_collision` (described on the previous slide)
- this call back must be defined taking two parameters
  - the first, `o`, is the object whose callback is being called
  - the second, `e`, is the collision event which describes the collision
- by using the event, `e`, it is possible to find out the other object in collision and other properties (if necessary)