Arjun's Brick-Breaker

Submitted by T.H.Arjun, CSD, 2019111012

DASS Assignment 2 and 3

Brick Breaker Terminal Game in Python

Dependencies

- colorama
- numpy
- linux to run aplay for sound effects

Instructions to Run

- Install all dependencies
- Open the directory of game in a full screen terminal
- Run `python3 main.py`

All Assignment requirements have been implemented

Assignment 3 Edits:

- Levels
    - Levels have been implemented easily because of modularity in `game.py` by adding an extra loop and some if statements.
    - `l` is the key assigned to skip levels
- Falling Bricks
    - The time for falling bricks can be edited in `config.py` by editing value of `MOVE_BRICK`.
    - The `move()` function was added to `Brick` class which is an overriding over `Game_Object move()`

```
def move(self):
      self.y += 1
      if self.y+1 == 29:
          return True
      else:
          return False
```

    - The following changes were made to `game.py` to move the brick on time limit.

```
if self.get_change_in_secs(self.level_start) >= MOVE_BRICK and
self.last_move >= 30:
          self.last_move = 0
          for brick in self.bricks:
              if not self.over:
                  self.over = brick.move()
              else:
                  brick.move()
```

- Rainbow Bricks
    - For adding Rainbow Bricks a new `rainbow_brick` class was added in `rainbow_brick.py`
    - It inherits from `Brick` class
    - Changes were made in `game.py` to change the strength each frame.
    - Following methods were overridden or defined:

```
def get_array(self):
      '''get's body of rainbow_brick
      Polymorphism: Overrides Game_Object get_array'''
      return self.array

  def did_collide(self, obj):
      collided = super().did_collide(obj)
      if collided == True:
          self.change = False
      return collided

  def change_strength(self):
      if self.change == True:
          self.strength = np.random.choice([1, 2, 3, 4, 5])

  def get_color(self):
      return self.strength_color[self.strength]
```

- Power-Up 2.0
    - Implementing this feature was fairly easy considering the modularify of the code.
    - The following code was added in `powerup.py` to implement this.

```
def did_collide(self, obj):
      collide = super().did_collide(obj)
      if(collide):
          return (True, self.get_array())
      else:
          return(False, None)

  def trajectory(self):
```

```
        '''Returns the points on the trajectory of the next move'''
        x1 = self.x
        x2 = self.x+self.xv
        y1 = self.y
        y2 = self.y + self.yv
        trajectory_return = []
        if x2 == x1:
            step = 0
            if y2 > y1:
                step = 1
            else:
                step = -1
            for y in range(y1+step, y2+step, step):
                trajectory_return.append((x1, y))
            return trajectory_return
        if y2 == y1:
            step = 0
            if x2 > x1:
                step = 1
            else:
                step = -1
            for x in range(x1+step, x2+step, step):
                trajectory_return.append((x, y1))
            return trajectory_return
        step = 0
        if x2 > x1:
            step = 1
        else:
            step = -1
        for x in range(x1+step, x2+step, step):

            y_ = ((y2-y1)*(x-x1))/(x2-x1)+y1
            y = int(round(y_))
            if y == y_:
                trajectory_return.append((x, y))
        return trajectory_return

    def move(self, x=-0.5, y=-0.5):
        '''moves the ball around to x,y. If no x,y directly moves. This
is overriding the basic move with extra functionality
                                    (example of polymorphism)'''
        super().move(x, y)
        flag = False
        if(self.x <= 1 or self.x > SCREEN_WIDTH-2):
            self.x = 1 if self.x <= 1 else SCREEN_WIDTH-2
            self.xv *= -1
            flag = True
        if(self.y <= 1):
            self.y = 1
            self.yv *= -1
            flag = True
        if(self.y > SCREEN_HEIGHT-3):
            self.set_inactive()
```

```
            flag = True
        return flag
```

- Shooting Paddle
    - Changes were made to `game.py` and `paddle.py`
    - Press `s` to shoot.. There is a delay between each shoot
    - The power up is represented by `!`
    - new class of `bullet` was made in `bullet.py`
    - It inherits `Game_Object`
    - All necessary criteria are met
    - Changes to `paddle.py`

```
def get_shoot_time(self):
      return self.shoot_time
```

    - Changes to `game.py` including bullets array and moving them around
- BOSS Enemy
    - The boss enemy is implemented as mentioned in the assignment PDF
    - `ufo.py` has the new class `UFO` created for the boss
    - `bomb.py` has the new `Bomb` class created for the bomb.
    - Both of these inherit from `Game_object`
- Bonus Assignment 3
    - Fireball Power Up
        - Represented by `F`
        - Changes made in `game.py`
    - Sounds
        - Sound effect were added by running `aplay` command in background in terminal using `os.system()`
        - Sound effect files are in `sounds` folder

All OOP Concepts were followed for Assignment 3 as well

Below given is OOP Concepts and Game Rules

OOP Concepts

- Encapsulation

- Everything is a class. We access them using their objects. Following are the classes and their files.
- `Game_object` - The base class for all game objects - `Game_Object.py`
- `Ball` - The class of the ball - `ball.py`
- `Brick` - The class of the normal brick -`brick.py`
- `chain_brick` - The class of the explosive brick **BONUS** -`chainbrick.py`
- `Game` - The class of a game, logic of game, score etc - `game.py`
- `Paddle` - Class of the Paddle - `paddle.py`
- `Power_up` - Class of the Powerup -`powerup.py`
- `Game_Screen` - Class of the game screen, handles printing - `screen.py`
- `Unbreakable` - Class of the unbreakable brick - `unbreakable.py`

- Inheritance

  - Every object in the game like `Brick`, `Paddle`, `Ball` , `Power_up` is a child of the `Game_object` class. They have common properties of a game object like x,y,xv,yv,array, color etc. These are common to all of them.
  - `chain_brick` which is the explosive brick of **BONUS** inherits from `Brick`
  - `Unbreakable` brick inherits from `Brick`
  - The hierarchy of inheritance is as follows:
    - `Game_object` -> `Brick`
    - `Game_object` -> `Ball`
    - `Game_object` -> `Paddle`
    - `Game_object` -> `Power_up`
    - `Brick` -> `chain_brick`
    - `Brick` -> `Unbreakable`

- Polymorphism

  - In `Ball` I extend the functionality by overriding the basic `move` of `Game_object`. I extend the functionality to deal with edge collissions. I also call `super().move()` since I am extending functionality with `move` for ball.

```
  def move(self, x=-0.5, y=-0.5):
      '''moves the ball around to x,y. If no x,y directly moves. This
is overriding the basic move with extra functionality
                                (example of polymorphism)'''
      super().move(x, y)
      flag = False
      if(self.x <= 1 or self.x > SCREEN_WIDTH-2):
          self.x = 1 if self.x <= 1 else SCREEN_WIDTH-2
          self.xv *= -1
          flag = True
      if(self.y <= 1):
          self.y = 1
          self.yv *= -1
          flag = True
      if(self.y > SCREEN_HEIGHT-2):
          self.set_inactive()
```

```
            flag = True
        return flag
```

- In `Brick` I override `did_collide` of `Game_object` class to extend its functionality to change ball velocity when it collides with brick. As always, I am calling `super().did_collide()` since I am extending it's functionality here.

```
 def did_collide(self, obj):
      '''checks collission with ball- also changes the velocity of
ball
      polymorphism- Overrides Game_Object did_collide with extra
functionality'''
      collided = super().did_collide(obj)
      if collided:
          if obj.x < self.x + self.xlength and obj.x >= self.x:
              obj.yv *= -1
          if obj.x < self.x+2:
              obj.xv -= 3
              return collided
          if obj.x < self.x + 4:
              obj.xv -= 2
              return collided
          if obj.x < self.x + 6:
              obj.xv += 2
              return collided
          else:
              obj.xv += 3
              return collided
      return collided
```

- In `chain_brick` I override `get_array` of `Game_object` since I want a different body from normal brick, I also overload `hit` and extend the functionality now to destroy the neighbours and intitiate chain reaction.

```
  def get_array(self):
      '''get's body of chain_brick
      Polymorphism: Overrides Game_Object get_array'''
      return self.array

  def hit(self, bricks):
      '''hits the brick and reduces its strengh
      Initiates chain reaction over other bricks
      Polymorphism: Function overloading over Brick hit'''
      curr_strength = super().hit()
      if curr_strength == 0:
          for brick in bricks:
              if isinstance(brick, chain_brick):
```

```
                    for other_brick in bricks:
                        if (other_brick.x == brick.x or other_brick.x ==
brick.x-brick.xlength or other_brick.x == brick.x+brick.xlength) and
(other_brick.y == brick.y or other_brick.y == brick.y-1 or
other_brick.y == brick.y+1):
                            other_brick.set_inactive()
        return curr_strength
```

- In `Paddle` class I override `get_array` and `get_color` of `Game_object` to get different sizes. I also override , `did_collide` to add extra functionailty of ball deflection based on point of collission and `move` for edge detection. I call `super()` as this function overrides the parent function with extra functionality

```
    def did_collide(self, obj):
        '''Polymorphism over game_object did_collide. Calculates
Positions'''
        if super().did_collide(obj):
            if(type == 0):
                if(obj.x < self.x+4):
                    return -3
                if(obj.x < self.x+6):
                    return -2
                if(obj.x < self.x+9):
                    return 2
                return 3
            elif type == 1:
                if(obj.x < self.x+4):
                    return -4
                if(obj.x < self.x+8):
                    return -3
                if(obj.x < self.x+12):
                    return -2
                if(obj.x < self.x+16):
                    return 2
                if(obj.x < self.x+20):
                    return 3
                return 4
            else:
                if(obj.x < self.x+5):
                    return -5
                if(obj.x < self.x+10):
                    return -4
                if(obj.x < self.x+14):
                    return -3
                if(obj.x < self.x+18):
                    return -2
                if(obj.x < self.x+22):
                    return 2
                if(obj.x < self.x+25):
                    return 3
```

```
                if(obj.x < self.x+30):
                    return 4
                return 5

        else:
            return 0

    def move(self, x=-0.5, y=-0.5):
        super().move(x=x, y=y)
        if self.x <= 1:
            self.x = 1
        if self.x+self.xlength > SCREEN_WIDTH-1:
            self.x = SCREEN_WIDTH-2-self.xlength
```

- In `Power_up` class I override `move()` and `did_collide()` and extend its functionality by calling `super()` and add to its functionality, to get type of powerup etc.

```
    def move(self, x=-0.5, y=-0.5):
        super().move(x, y)
        if(self.y > SCREEN_HEIGHT-3):
            self.set_inactive()

    def did_collide(self, obj):
        collide = super().did_collide(obj)
        if(collide):
            return (True, self.get_array())
        else:
            return(False, None)
```

- In `Unbreakable` I override `hit` to return -1 since its unbreakable brick.

- Abstraction
    - I have getters and setters for Class variables. This is Java Concept of Abstraction.
    - All `Game_objects` have functions like `move()` , `did_collide()`, `is_active()` which hides the implementation from end user and is overriden in children with help of `super()` to add extra functionality.
    - `Ball` has `trajectory()`, `flip_move()`,`should_move()`
    - `Brick` has `hit()` and `pass_through_collide()`- for pass through powerup
    - `Game` class has many functions
    - `Paddle` has `make_shrink`, `make_enlarged`
    - All these functions along with getters and setters are example of Abstraction

Game Instructions:

Press `a` to move left. `d` to move right. Game screens like main menu, instructions, pause etc have instructions on key presses.

Brick Colors signify the Following strength

- Green - 1
- Yellow - 2
- Cyan - 3
- Blue - 4
- Red - 5
- White - Unbreakable

Bonus - Explosive Brick

|>>>>>>>| are explosive bricks. Once broken they initiaite a chain reaction, among the group, and destroy their neighbours also.

Power Ups

Power Ups appear when a brick is broken and comes falling down.

- E - Expand Paddle - Expands the paddle for 30 seconds
- S - Shrink Paddle - Shrinks the paddle for 30 seconds
- X - Ball Multiplier - Makes every ball into two
- > - Fast Ball - Increases Speed of ball for 15 seconds
- P - Thru Ball - The ball passes through every brick and breaks them, irrespective of strength for 30s
- G - Paddle Grab- Allows the paddle to grab onto the ball and release on pressing r. Lasts for 20 seconds. On reaching time limit, the powerup deactivates and no more balls could be caught. Already caught balls stays on paddle which should be released by pressing r. The ball follows the expected trajectory on release.

The time limits for each power up can be modified in `config.py`

Lives are available. A life is lost whenever all balls are lost. A ball is lost when it hits bottom. When a live is lost paddle resets to original position, with a ball on top.

A game is lost when all lives are lost. A game is won when all bricks except the unbreakable ones are broken down.

Scoring is easy. Whenever you hit a brick and reduce strength, your score increases by 1. Your goal is to maximise this score.

Score, Time Played, Lives Left, Time for each powerup is shown at the bottom of the screen.