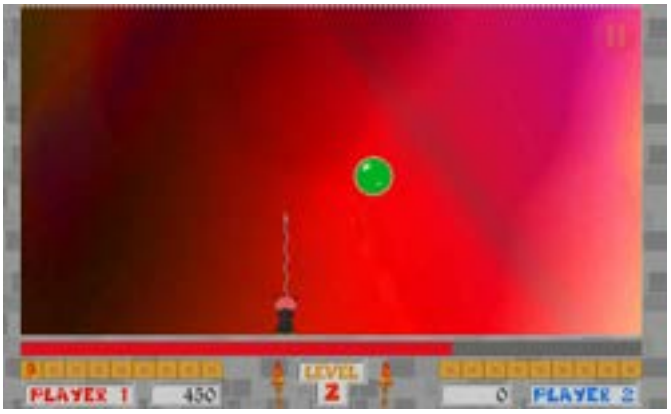
Assignment 2 Bubble Trouble

Due: April 17th, 2023 (04.00 AM) **CMPE 160**

In this assignment, you are asked to code a Java program -the clone of the Bubble Trouble game- using the StdDraw library.



(a) **Figure 1:** Actual Bubble Trouble game

(b) **Figure 2:** Expected output of your implementation

Example of expected output: https://drive.google.com/file/d/132V-sJYd-1rGYziQkB0bit2SO5Jhbb4A/view?usp=share_link

The actual game has many more features, but we expect you to develop a simple program that follows the instructions below. It's highly recommended that you look at the game and play to observe the mechanics. Also, you can check the suggestions we provided at the end of this description to have a better experience while designing the game. Most importantly, you should use the video of the game we provided as a reference for your implementation and read the whole document before starting the design and implementation.

In Bubble Trouble, the player in a 2D environment can move left and right inside the screen and must shoot down all the floating bubbles above him. When the player shoots a bubble, an arrow will go upward, causing the ball to pop, and if the bubble is big enough, it will split into multiple smaller bubbles that the player must also shoot down. Furthermore, if the player makes contact with a bubble, the player loses. The game has a time limit, indicated by the rainbow bar at the bottom of the screen that slowly depletes. If this time runs out, the player loses. The game has four essential elements: **player**, **ball**, **arrow**, and **time bar**. We will further elaborate on the rules related to each element below. At the end of the description, we will attach some suggestions regarding the implementation.

1. Environment Class

Environment class consists of game-related constants and methods.

Environment class should also include a method for running the game. First, you should initialize the environment, bar, player, and balls. Then you should start the game and simulate events frame by frame. Events include displaying background, player movements, ball movements, arrow movements, ball-arrow intersections, ball-player collisions, and replaying the game.

Remember to check the detailed descriptions of each class.

a) Displaying Background

The first thing you draw must be the background. Every other object must appear on top of the background. All images will be provided in the assignment package including the background image named **"background.png"**.

The second thing you have to implement is a time bar that calculates and shows the remaining time of the game. You should use the **"Bar"** class to create the time bar. Please check our video to see the background behavior in different situations.

SUGGESTED BACKGROUND CONSTANTS:

`canvasWidth`	800
`canvasHeight`	500
`scaleX`	(0.0, 16.0)
`scaleY`	(-1.0, 9.0) -> For Time Bar = (-1.0, 0.0) For Game = (0.0, 9.0)
`TOTAL_GAME_DURATION`	40000 ms
`PAUSE_DURATION`	15 ms

b) Player Movements

The player should be able to move left and right on the canvas by using the left and right keyboard buttons. The player can't jump or do anything else except moving left and right. You should get key movements from the user and draw the actions.

The player's appearance will be given to you as **"player_back.png"**.

c) Ball Movements

Balls should make a ballistic motion. The ball image also will be given to you as **"ball.png"**.

Remember that balls should do elastic collisions while hitting the game's left, right, and base walls. (You do not need to implement a wall, you can use the borders of the canvas.)

d) Arrow Movements

Arrows should go upwards starting from the player's exact position in the X-axis when the spacebar is pressed. Arrows should terminate and disappear when they reach the ceiling. You should get key movements (the spacebar) from the user, and draw the actions.

Remember that there can be only one arrow in the game. It means the player can't shoot two arrows while the other arrow is not completed.

Arrow's appearance will be given to you as **"arrow.png"**.

e) Ball-Arrow Intersections

You should check whether the arrow hits the ball or not using circle equations. **Do not assume that the ball is a rectangle.** You can assume the arrow is a straight line. It would be best if you came up with a solution for how to check arrow-ball intersections. You can use mathematical inequalities to observe arrow-ball behavior.

Also, remember that you should check the borders of the ball's circle, not the center.

When an arrow hits a ball, the ball must split into two small balls if its level is not 0. Check Ball class for further details.

f) Ball-Player Collisions

You should check whether the ball hits the player or not. You can assume that the player is a rectangle, but you should not assume that the ball is a rectangle. Ball's shape is always circular in all cases. Using

mathematical inequalities, you should check that the circle intersects with the rectangle. Also, keep in mind that you should check the borders of the ball, not the center.

g) Replaying the Game

Game can end in 2 states:

GAME OVER STATES	*WIN STATES*
If one of the balls hits the player.	If a player shoots all the balls in a given time.
If a player can't shoot all the balls in a given time.	

You must draw a rectangle using the provided image file **"gamescreen.png"**. There should be a text called **Game Over!** or **You Won!** based on the end state. After that text there should be 2 small texts called **To Replay Click "Y"** and **To Quit Click "N"**.

SUGGESTED GAME SCREEN CONSTANTS:

	FONT, STYLE, SIZE	*PLACEMENT*
Game Over! / You Won!	Helvetica, Bold, 30	(scaleX/2, scaleY/2.0)
To Replay Click "Y"	Helvetica, Italic, 15	(scaleX/2, scaleY/2.3)
To Quit Click "N"	Helvetica, Italic, 15	(scaleX/2, scaleY/2.6)

	PLACEMENT, SCALED VALUES
Game Screen	x: scaleX/2 y: scaleY/2.18 scaledWidth: scaleX/3.8 scaledHeight: scaleY/4

2. Player Class

The player is expected to do the following:

- Can move left and right on the screen using left and right keyboard buttons
  - Update the x position of the player according to the clicked key (left or right)
- Can't go outside the screen's borders
  - Compare the player's x position with the board's x position (do not forget to consider the width of the player)
- Can shoot an **Arrow** by clicking the space key
  - Activate the Arrow to show it on the screen
  - Arrow's starting point has x coordinate that is equal to the player's x position when he shooted

Note: After the player shoots the arrow, the arrow's x position does **not** change if the player moves left or right.

- If the ball touches the player, **(Game Over!)**
  - Consider the player as a rectangle. Hence, if a ball touches that rectangle area, Game Over!

(Note: the player is not exactly a rectangular shape, but this is an approximation to make the calculations easier)

SUGGESTED PLAYER CONSTANTS:

`PERIOD_OF_PLAYER`	6000 ms

`PLAYER_HEIGHT_WIDTH_RATE`	37.0/27.0
`PLAYER_HEIGHT_SCALEY_RATE`	1.0/8.0

`PERIOD_OF_PLAYER:` The total time required for a player to traverse through the X-axis.
`PLAYER_HEIGHT_WIDTH_RATE:` It's the given "**player.png**"s **height / width** rate.
`PLAYER_HEIGHT_SCALEY_RATE:` It's the **player's height / scale of Y**.

3. Arrow Class

The arrow is expected to do the following:

- It should be set as inactive first. When the user presses the spacebar, it becomes active until it reaches the canvas's top or shoots a ball.
- Arrow should be placed under the player, which means you should draw the arrow first, then the player. You can check **Figure 1** and **Figure 2** to see the difference.
- When the arrow becomes active, you should hold its starting position and find a way to print the current arrow length.

SUGGESTED ARROW CONSTANTS:

`PERIOD_OF_ARROW`	1500 ms

`PERIOD_OF_ARROW:` The total time required for an arrow to traverse through the Y axis excluding bar. (0.0, scaleY)

4. Ball Class

Balls are expected to do the following:

- They should make a projectile motion without fractions.
- They should do an elastic collision with the left, right and bottom walls.
- When they hit the player, the game should end.
- Balls are not created randomly. They have levels that determine their attributes. There will be 3 levels in our game: Level 0, Level 1, Level 2. Level 0 will be the smallest ball in our game, Level 2 will be the biggest ball in our game. You can see the starting points of the balls in the game beginning.
  - Level 0: (scaleX/4, 0.5) moving right
  - Level 1: (scaleX/3, 0.5) moving left
  - Level 2: (scaleX/4, 0.5) moving right
- When an arrow hits the ball there are 2 options:
  - If the ball is the smallest one, the ball and arrow should disappear.
  - If the ball is not the smallest, it should split into 2 smaller levels of balls and the arrow should disappear. **For example, if a level 2 ball splits, it should generate two level 1 balls.**
  - **Newly generated balls should start their projectile motion from the splitted ball's center.**

Do not forget that an arrow can only shoot one ball. One arrow shooting two balls is invalid.

You can refresh your physical knowledge by checking ***"projectile motion"***, lab 4 slides, and ***"elastic collision"***.

SUGGESTED BALL CONSTANTS:

`PERIOD_OF_BALL`	15000 ms

`HEIGHT_MULTIPLIER`	1.75
`RADIUS_MULTIPLIER`	2.0
`minPossibleHeight`	player's height in scale * 1.4
`minPossibleRadius`	scaleY * 0.0175
`GRAVITY`	0.000003 ms^{-2} * $scaleY$

`PERIOD_OF_BALL:` The total time required for a ball to traverse through the whole X axis.
`HEIGHT_MULTIPLIER:` The multiplier we use to calculate the maximum height that different levels of balls can reach.

I. $currentHeight = minPossibleHeight * HEIGHT\_MULTIPLIER^{(current\ level\ of\ ball)}$

`RADIUS_MULTIPLIER:` The multiplier we use to calculate the radius that different levels of balls can have.

II. $currentRadius = minPossibleRadius * RADIUS\_MULTIPLIER^{(current\ level\ of\ ball)}$

`minPossibleHeight:` Minimum level ball's maximum height that it can reach starting from ground. **Check the game video we provided to see the behavior of the different levels of balls.**
`minPossibleRadius:` It is the minimum level ball's radius.
`GRAVITY:` It is the calculated gravity constant for your game.

5. Bar Class

The time bar lies on an image we also provided as **"bar.png"**. The time bar can be drawn using a rectangle at the top of the bar image. Firstly, It should appear as a full-sized time bar shrinking over time, and when the player wins, it should stop. If a player cannot win in the given amount of time, no time bar should be seen at the end. In addition, the time bar starts as yellow *-in RGB : (225,225,0)-*, changes linearly by time, and ends as red *-in RGB : (225,225,0)-*.

When the game starts, you can hold a **startTime** variable which takes your computer's current time in milliseconds by using **System.currentTimeMillis()**. Then after each drawing, you can compare the time difference between the **currrentTime** and **startTime**.

SUGGESTED BAR CONSTANTS:

`Total Height Scale Given to Bar`	*total of 1.0 -> (-1.0, 0.0)*
`Time Bar's Height Scale`	*0.5*

`Total Height Scale Given to Bar:` It indicates the given height scale for bar image. It should be placed between (-1.0,0.0) in the Y axis and (0.0,scaleX) in the X axis.
`Time Bar's Height Scale:` It indicates the time bar's scale on top of the bar image. The time bar should be centered.

Figure 3: "Time Bar" lays on the top of the "bar.png" image

6. Main Class

In practice, you should use the Main method to initialize and run the game by calling the Environment class, but feel free to use main as you wish.

Keep in mind that the game should be replayable. You should take into account the replay status of the game. Your main class is going to be run only once, your code must do the replay inside the execution.

Suggestions:

- Collecting relevant game constants in **the "Environment"** class as static constants is recommended to make your code clean. Then you can call the constants from other classes, such as `Environment.TOTAL_GAME_DURATION`.
- Finding the best-fit value for the game constants may require lots of trials and errors, so to make changes in values easily you should assign the values to variables and do the calculations not with constants, but with variables.
- Designing and defining player's moves in Player class and then calling them in Environment class can be a good choice of implementation. It works for Ball and Arrow classes too.
- Do not forget that using main for everything will be hard and confusing, also can cause you to get lower grades.

Evaluation Criteria and Grading for Assignments

Code
10% Compliance to programming style, e.g., naming conventions, indentation, comments.
80% Correctness of the solution and Object-Oriented Design Choices

Report
10% Completeness of the report, compliance to the report format, correctness of the content and language.

Submission Guide

Submission Files
Submit a single compressed (.zip) file to Moodle.
Name your zip file as name_surname.zip.
Zip file should contain all source codes (under the \code directory), and report (in PDF format, under the \report directory).
Name the main code which is used to run your assignment as name_surname.java.
Name your report as name_surname.pdf.
Contents of each Java file should start with your name, student ID, date, and a brief code summary in a comment block.

Mandatory Submission
Submission of assignments is mandatory. If you do not submit an assignment, you will fail the course.

Late Submission Policy
Maximum submission delay is two days. Late submission will be graded on a scale of 50% of the original grade.
Submission is mandatory even if you submit your assignment late.

Plagiarism
Plagiarism leads to grade F and YÖK regulations will be applied