Bouncing Balls Documentation

This game was programmed in 4 weeks for a school project in the Processing programming language which is based on Java.

If you want to learn the syntax (code) of Processing, this is the primary resource https://processing.org/reference/.

This program uses physics-based math to calculate how a ball moves.

This doc will be easier to understand after trying the program.

Main File (Bouncing_Balls):

This file controls how the game runs.

In uninterrupted operation, it does 2 main things: setup() and draw()

- the code in setup() runs once when the program starts up
- the code in draw () repeats every frame. This is where most things happen

It also accepts user mouse inputs with mouseClicked() and keyboard inputs with keyPressed().

Before anything else, the program declares or initializes all objects and global variables

- "initializing" is creating a new variable with an assigned value
- "declaring" is creating a new variable without an assigned value
- if variables are created in setup, they only exist in setup and can't be accessed in draw
    - this is called a local variable as opposed to global variables

Setup():

In setup(), the program:

1. Sets the framerate to 120fps (frames per second) (the screen refresh rate)
    - Processing's basic framerate is 60fps, but I increased it because it kept lagging with the addition of an image background
    - All the math is still done assuming 60fps though since the decision to use an image for the background was made after programming the math
2. Sets the screen size to fullscreen
3. Loads the image for the background and resizes it to fit the screen
4. Loads the high score from the "High Score.txt" document
5. Sets the balls' types ('standard' for the player ball)
6. Assigns the balls' starting positions
7. Creates and randomizes the victory block (this goes before the obstacles for a reason)
8. Creates and randomizes the obstacles
9. Creates the balls for predictive aiming

Draw():

In draw(), the program:

1. Displays the image for the background loaded in setup
2. Calculates the force of "gravity" to (9.8 pixels/60 frames) * (ball mass)
    - This is based on the formula for the force of gravity on Earth F = mass * gravity
3. Displays and applies movement to the ball
4. Displays the predictive aiming balls
5. Displays and checks for contact with the victory block
6. Displays and check for contact with each obstacle
7. Checks for collision with the edges of the screen and the ball menu
8. Displays the ball menu
9. Displays the instructions menu only if the player has toggled it

This order of operations could be played around with. Things ordered later happen later.
For example, the instructions menu shows up above everything else because it is drawn last.

<u>mouseClicked():</u>
When the mouse is clicked,

1. The function checks whether the click was inside the ball menu. If it was, it skips launching the ball and checks if any of the example balls were clicked
2. If the click was anywhere else on screen, the ball will be launched

<u>keyPressed():</u>
When a key is pressed,

1. The function grabs the key that was just pressed
2. If it has instructions for the key, it will execute them
    - The instructions need to end with "break;" otherwise it would execute all the code meant for other letters. I don't know why this is the case, it just is.

<u>Ball, Classes, and Functions:</u>
The balls are objects made using a class.
This way of programming is called "object-oriented programming."

You can think of a class as a collection of data and functions grouped up for organizational purposes. For example, you'd want to keep a ball's x-position and display function in one place away from an obstacles' x-position and display function

Objects are instances of a class. How I think of it is that when you create a new object, you create a new pocket of data that's below but connected to its originator, much like a family tree. Whether this is the right mindset or not, I have no idea, and I have 2 years of programming experience so don't worry if you don't get it.

In programming, you can create your own functions.

In Processing, you first have to declare that the function's datatype will be "void" - this just lets the program know that your instructions won't return a value but will rather execute some instructions. From there, you just give your function a name and do whatever you want with it. You can strictly use the pre-built functions in the language or reference your other custom-made functions – the sky's the limit. Creating functions makes it so that you don't have to rewrite the same code every time you want to do the same thing. Your function will be pre-loaded when the program runs, so you can treat it like using the pre-built functions.

The ball class has the following properties: x-position, y-position, diameter, mass, x-velocity, y-velocity, and colour.

The player ball is referred to as b1 in the code while the menu balls are ex1, ex2, and ex3 from left to right.

The ball-type changing functions only change a ball's colour and mass. Gravity does the rest of the work when movement happens.

The balls are drawn using Processing's pre-built ellipse() function.

Commands:

This file contains all of the functions to be called upon when the user inputs something.

instructMenu()

- Displays a white rectangle in the middle of the screen
- Displays the instructional text; each line is 40 pixels apart

reset()

- Resets the ball's position, movement, and the number of boosts to where they started
- I chose (screen width/19) and (screen height/2) after trial and error

newLvl()

- Generates a new position for the victory block
- Generates new positions for each obstacle

quit()

- Saves the current score to the "High Score.txt" document if it is a new high score
- Exits the program
- I made a new array (list) of one item to store the score in because the pre-built saveStrings() function only accepts items in a list

Boosts

- Checks if the player has boosts left to use

- Adds movement to the ball in the direction of the boost. Because the boost is additive, it drastically speeds up the ball if it's already going in that direction and slows the ball down if it's going in the opposite direction.
- Subtracts one boost for the player to use

Movement:

This file handles how the ball moves and some of the collision.
The variable "dampener" controls how much the ball slows down when it collides with something.

move()

- Moves the ball by its x- and y-velocity values

edgeCheck()

- Detects the ball's collision with the edges of the screen
- To prevent the ball from getting stuck in the walls, it only triggers collision if the ball is going toward the edge
- It bounces the ball back and subtracts from its velocity
- To prevent the ball from going backward when its velocity is subtracted from, if the velocity would go backward, change it to zero instead
- How the bouncing works is reminiscent of Law of Reflection from physics. The Law of Reflection is only used for light in the real world, but it suffices for this program.

menuCheck()

- Detects the ball's collision with the ball menu using the same method as edgeCheck()
- I could have made a ball menu object or made the menu's dimensions global variables to make the code simpler

go()

- Launches the ball
- I dealt with the movement in 2 parts: x and y
- X-movement was easy because it was just constant motion. I divided the x-distance the ball needed to go by the frames for one second (now half a second) to get x-velocity
- Y-movement was a bit more tricky because I had to review my Physics 11 notes. I used a rearranged physics kinematics formula for projectile motion to get a starting y-velocity. One challenge I thought I would have was that Processing reverses the y-axis. Where y = 0 on a traditional graph, that is the total height in Processing. However, that was solved by gravity being a positive force in Processing instead of a negative one normally. That meant that the formula spat out negative numbers which go up visually in Processing's system.

gravity()

- Applies the gravity calculated in draw() to the ball
- Doesn't work if the ball is stationary because I don't want gravity to instantly take over when a level starts
- Gravity is only applied when the ball is above the screen's bottom edge to prevent it from falling into the abyss

Obstacle:

Obstacles are objects created using a class. For an explanation of what those are, see the Ball section.
Obstacles have the properties: x-position, y-position, width, height, and colour.
The obstacles are held in an array (list) because it allows the use of loops to repeatedly do something to every object which simplifies and shortens programming.
The victory block is just a differently coloured obstacle object that uses different functions.

vObstacle()

- Differentiates the victory block from normal obstacles by making it green

wincon()

- Triggers when the ball goes within the bounds of the victory block
- Adds one to the score
- Adds an additional point to the score for every 2 boosts unused. This encourages playing without relying on the boosts. The number of leftover boosts gets divided by 2, and because of how integers work in programming, the result gets rounded down if has decimals
- Updates the high score in real-time if the current score is higher
- Calls reset() from Commands to reset the ball
- Calls newLvl() from Commands to generate a new level

vRandom()

- This function randomizes the position and size of the victory block (always smaller than the obstacles)
- After randomizing the position, it uses true/false flags to check if any part of the block is outside the bounds of the screen, inside the ball's starting location, or inside the ball menu. If they satisfy any of those conditions, they fail that check. If any check comes back failed at the end, everything is re-randomized and re-checked until the criteria are satisfied.

obstacoll()

- This function checks if the ball is touching an obstacle
- If it is, it uses the same mechanism as edgeCheck() to bounce the ball

randomize()

- This function randomizes the positions and sizes of the obstacles
- It uses the same true/false flag approach to checking the "good"-ness of a randomization with added checks for the victory block and other obstacles
- It compares one obstacle to the others one at a time
- It checks if any part of the obstacle is inside of the other

UI:

This file handles some of the visual and user-interactive features of the program

menuRect()

- Draws the grey box for the ball menu

menuBalls()

- Displays the ball menu's example balls and their associated text

menuMouseCheck()

- Checks whether the mouse is within the bounds of any of the menu balls
- Only gets called when the mouse clicks within the bounds of the menu

instructText()

- Displays the text for how to toggle the instructions, high score, current score, and remaining boosts

predict(Ball, float)

- This is the only custom-made function that accepts parameters
- It's used to place and draw the balls used in predictive aiming
- It takes a ball object and a number to determine its place in the prediction line
- Similar to the go() function, I dealt with this in 2 parts: x- and y-placement
- It gets the x-position by quartering the x-distance from the ball to the mouse and putting the predictive ball in its appropriate place. It also adds the ball's x-position to the prediction's x-position to get it to start from the ball.
- It uses a rearranged version of the projectile motion formula from physics. It first gets the expected y-velocity of the ball if it were to be launched at that moment. It then plugs it in to the rearranged formula to get y-displacement. After that, it adds the ball's y-position to get the line to start from the ball
- Displays the predictive ball
- Function called in loops

Known Issues:

1. The ball slips through obstacles if it stays in constant contact with their edges. This is most likely due to the application of gravity. An exception to gravity needs to be added for when on an obstacle's edge either by changing the gravity function or by making gravity zero when the ball doesn't can't bounce off an obstacle's edge.
2. The program crashes if anything besides a number is in the High Score text file. I ran out of time for this project to bug test and to incorporate error detection as high score keeping was a last-minute addition.