Word Count: 2830

Overview

This project involved developing a game inspired by the classic Breakout, using Processing with Java.

We implemented all of the primary requirements:
- For the scoring mechanism, the number of hits to destroy a particular brick corresponds to the number of points for destroying that brick: yellow is 1 hit 1 point, green is 2 hits 2 points, red is 3 hits 3 points

We implemented all of the secondary requirements:
- The speed of the ball increases between level 2 and level 3 of our game. The spec says that "according to rules of your own devising: trigger changes in the speed of the ball". Clearing level 2 is what triggers the change in speed
- The angle of the ball becomes steeper the further from the centre of the paddle it hits. If it hits the exact centre, the x direction of the ball is unchanged and the y direction is flipped. If the ball's angle is too steep (i.e. almost vertical) it will be adjusted to be more shallow (more horizontal) after hitting the paddle
- The paddle gets larger after green blocks are destroyed (with a max width), and narrower after red bricks are destroyed (with a min width)
- Level 2 has moving bricks, and all levels have bricks that take more than one hit to destroy
- We have 2 extra levels. Level 1 is standard. Level 2 has moving bricks. Level 3 has the same brick arrangement as 1 but with power-up bricks, a creature that randomly changes the direction of the ball on collision, and a faster ball
- We have a 'tnt' power-up brick, that hits nearby bricks when destroyed. We have a 'heart' power-up brick that gives an extra life when destroyed

We implemented **all of the tertiary requirements**:
- We have a triangle-shaped creature that bounces around the screen and, when it collides with the ball, changes the direction of the ball randomly
- We have created a separate testing class which tests individual methods from the other classes

Team Working

Throughout the whole project, our team had good communication and productivity. As soon as one member was free to work on the project, they did and updated the group chat to make sure that no one else would work on it to cause collisions with the project.
240021654 started off the project during the Spring break giving the whole group a start by completing the primary requirements as well as working on some Secondary Requirements. 240010314 took off from where 240021654 left off by refactoring the code: they added an enum for brick types, cleaned up the code, renamed the variables to more reasonable names, and changed the ball to use a direction vector and speed, instead of xspeed and yspeed. 240010314 also updated and fixed the logic issues as well and improved the

collision system which really got the game to work properly. 240010314 also implemented the creature functionality.
240026231 joined in working on the leaderboard, power-ups as well as completed the testing for the project. Since 240026231 was unable to complete the leaderboard, 240021654 helped in making the leaderboard persistent and making sure only the top 5 get displayed and constantly updated after each run.

We had a TODO file to devise and assign remaining tasks after the start given by 240021654 had given. This helped us keep track of who was doing what and what was left to do. We also used a 'thoughts' text file to record our thoughts about possible improvements, questions about the code, and potential TODO tasks. We kept in frequent communication using a Whatsapp group chat and met as necessary to discuss. When we met, we occasionally used pair programming.

Design

See the docs/ folder with the code for the flowchart and UML diagram

Ball.pde

We decided to have all of the collision methods with the ball in the *Ball* class to make things more clear, with the functionality grouped. The only deviation from this was the collision with Creatures. This is because different hypothetical creatures could override the standard collision behaviour of *creature* and the ball. Therefore, it made more sense to have this collision method in the respective *creature* classes.

makeDirectionVectorUnitVector() & direction setters

The *makeDirectionVectorUnitVector()* was introduced after the ball was made to use a direction vector and a speed, as opposed to an *xspeed* and a *yspeed*. It is necessary so that the speed does not change when the direction changes. It ensures that the *xdirection* and *ydirection* of the ball always produce a vector of magnitude 1 (a unit vector). We also added setters for the direction vectors so that whenever they are changed, the *makeDirectionVectorUnitVector()* is called. The *xdirection* and *ydirection* attributes were also made private, so that they cannot be changed without using the setter.

updatePosition()

This method moves the ball in steps. This is so that during each frame of movement for the ball, collision is checked a number of times (determined by *stepCount*). We have it set to *8*, but this can be changed if required. This method also calls the methods to check collision of the ball with the walls, bricks, or the bottom of the screen. These checks are in this method so that in *main*, we can simply call *ball.updatePosition*() and all of the collision checks will be 'automatically' made.

checkBrickCollision() & checkBrickCornerCollision()

The *checkBrickCollision()* method first checks if the ball is colliding with the corner of a brick by calling *checkBrickCornerCollision()*. This is because if the ball hits a corner, the resulting collision should ideally reverse both the x and y directions of the ball, which is different from the standard collision model with bricks.

Creature.pde

This is an abstract class. We chose to use an abstract class so that if we had time to make more creatures, it would be fairly straightforward, as well as the fact that all creatures have a reasonably standard set of features (e.g. bouncing off of walls, a *draw()* method, an *update()* method…).

We decided to leave the *updatePosition()* and *draw()* methods abstract because these are likely to be unique across every creature (*draw()* will contain the drawing of the particular shape/sprite of the creature).

Batty.pde

This class extends the *Creature* abstract class. The starting position is the top right of the screen, and is calculated in the constructor of the class according to the radius (so that the *Batty* doesn't spawn in the walls).
We decided that *Batty* should just be a simple equilateral triangle. This meant that drawing the triangle required some trig calculations because *Creature* only has a *currentXPos* and *currentYPos*, which we used as the centre of the triangle (i.e. equidistant from each vertex).

collide()

This method overrides *Creature*'s *collide()* method because *Batty* reflects the ball in a random direction. This method uses the *random()* method to determine whether to multiply *xdirection* and *ydirection* by a positive or negative random number respectively, when the ball collides.
The collision model is rather primitive at the moment, and simply checks the distance from the center of the ball to the centre of the triangle. However, the distance from the centre of a triangle varies along the sides of the triangle, so the collision model is not perfect.

Brick.pde

Initially, we had the type of brick stored as an integer (e.g. 2 = yellow brick that takes 1 hit to destroy). This made the code very confusing as we had seemingly arbitrary numbers littered throughout the code. We therefore decided to instead use an enum *BrickType* for the types of bricks. This allowed us to have a static data structure to store different types of bricks. Also, enums can have methods which was very useful, because it made it much easier to get the colour of a brick after it was hit.

get___Corner() methods

These methods were created for the *checkBrickCornerCollision()* method of *Ball*, and simply return an array with the first value the x coordinate and the second the y coordinate of the respective corner of the brick.

BrickType.pde

This enum was written using a template from a GeeksForGeeks article [2]. We have three types of bricks that are constructed within the enum, and that take a number of hits to destroy and an array of colors. These colors represent the different colors of each respective brick as that brick is hit (i.e. the bricks get lighter in color before they are destroyed). Initially, we tried to use *Processing*'s *color()* method, but we encountered an issue with calling this method from a static context (from the enum) [3]. This was resolved by using the hex code for each color, instead of rgb values [4].

PowerUpBrick.pde

This class is a child of *Brick*. We decided to extend *Brick* because the powerup bricks are very similar to regular bricks, except that they employ some special effect when destroyed and have an image drawn over them.

Leaderboard.pde

The end of this class contains the definition for another class, *ScoreEntry*, which is just a simple class to associate a player name with their score. For storing scores, we used a simple csv-style text file that has the name and score separated by a comma. We added another feature in the leaderboard class, that if more than 5 people have the same high score, the user can enter their name and see newScores.txt to view the file.

Score.pde

This class is distinct from *ScoreEntry* in that it is used to score the user's current score as they are playing the game.

main.pde

This is probably the most complex file of our project. This is the file that contains the driving code for the actual game.
We have a number of 'global' variables at the top of the file that are used in many of the methods and control the characteristics of the game.
We decided to have a grey border around the edges of the screen to make the game feel a bit more retro. This did, however, lead to some difficulties later on when it came to the ball getting trapped in the borders.

draw()

There are a number of checks in this method that return if the condition is true. It is necessary for these checks, so that the ball and paddle are not drawn on the leaderboard and game-over screens.
The *ball.updatePosition()* method is called from this method without any calls to collision methods. This was the intention behind calling collision methods from within the *updatePosition()* method of the *Ball* class: it makes things simpler.

startLevel2() & startLevel3()

We decided to use methods to create the subsequent levels, because they can simply be called from *draw()* once all of the bricks are destroyed.
Level 2 has moving bricks, and level 3 has brick powerups, a faster ball, and *Batty.*
The arrangement of power-up bricks is pre-defined in level 3. This is because it makes sense for the outer bricks to be hearts (so that they are easier to get – players are likely to lose lives by the time they get to level 3), and for inner bricks/red to be tnt (so that they are more difficult to get).

keyPressed() & keyReleased()

These methods are used to control the movement of the paddle. The movement of the paddle is controlled by two Boolean variables to keep track of if the left or right key is pressed. In *keyPressed()*, if the key pressed is a direction key, that direction is set to true and the paddle is moved in that direction in *draw()* until *keyReleased()* is automatically called by *Processing* once they key is released. The Boolean variables are then both set to false to stop the paddle moving.

Testing

Please see the *docs/* folder for videos and screenshots taken during development, and a file *videoAndScreenshotKey.txt* which gives a brief description of what each screenshot and video show.

We created a tests.pde file in which we have created 9 methods that test different methods and functions of the classes. The methods were added to the main class, which means that when you run the program the results of the tests will appear in the console in Processing, as seen in one of the screenshots (4).

Besides all of this we also tested the program by playing the game. Doing that allowed us to see how a user would experience the game. We tested whether the space button starts the game and whether the left and right arrow move the paddle.

What is being tested?	**Name of the testing method**	**Expected outcome**	**Actual output**	**Explanation of the testing method**
update method from the MovingBrick class	testingMovingBrick()	Testing the update method from the MovingBrick class: Passed	Testing the update method from the MovingBrick class: Passed	Creates a movingBrick, calls on the update method and checks if the x coordinates have changed in which case it returns Passed
SetPowerType and hit method from the PowerUpBrick class	testingPowerUps()	Testing the setPowerType method from the PowerUpBrick class: Passed	Testing the setPowerType method from the PowerUpBrick class: Passed	Creates a powerUpBrick, assigns it a life power and then
Hit method from the Brick class	testingBrickHit()	Testing the hit method from the Brick class: Passed	Testing the hit method from the Brick class: Passed	Creates a yellow brick, calls on the hit method and checks if the brick is destroyed
Update position and brickCollision from the Ball class	testBallCollision()	Testing BrickCollision method from the Ball class: Passed	Testing BrickCollision method from the Ball class: Passed	Creates a brick and a ball, then it updates the position of the ball and cehcks the collision with the brick
Update position from the Ball class	testUpdatePosition()	Testing the UpdatePositio n method from the Ball class: Passed	Testing the UpdatePosition method from the Ball class: Passed	Creates a new ball, saves its current x coordinates. Then calls on the updatePosition method and if the x coordinates changed the test passed
AddPoints method from the Score class	testAddPoints()	Testing the addPoints method from the Score class: Passed	Testing the addPoints method from the Score class: Passed	Creates a new score, with initial value of 0, then uses addPoints method to add 4 points to it and checks whether the value of the score has changed
LoseLife method from the Lives class	testLoseLife()	Testing the LoseLife method from the Lives class: Passed	Testing the LoseLife method from the Lives class: Passed	Creates a new lives object with the initial value of 3, then it uses the loseLife method and checks if the new number of lives equals 2
LoadScores and Sort Scores method from the Leaderboard class	testSortScores()	Testing sortingScores from the Leaderboard class: Passed	Testing sortingScores from the Leaderboard class: Passed	Creates a new Leaderboard object and uses loadScores which also calls on SortScores method. These save all of the scores in an array list in the descending order of the scores
Top5 method from the Leaderboard class	testTop5()	Testing the top5 method from the Leaderboard class: Passed	Testing the top5 method from the Leaderboard class: Passed	First uses the loadScores method to get all of the scores and then creates a boolean thareturns the boolean from the top5

				method with a new score which is in the top 5 scores

Evaluation and Conclusion

All the requirements — primary, secondary, and tertiary — have been successfully implemented in our Breakout game. This project has given us a deeper understanding of OOP and has improved our team-working skills.

We are fairly pleased at how the game has turned out. We made some good use of more advanced concepts such as inheritance and enums. The game is definitely playable (and reasonably fun) and actually has somewhat of a skill curve.

However, the fact that *Processing* does not strictly enforce OOP usage and modularity enabled less-than-ideal usage of global variables, which did cause confusion during development.

Our collaboration was good and we were able to communicate effectively.

There are still some things that we would like to improve given more time:

- Improving the collisions, particularly with the ball on the bricks
- Fixing the bug where the ball gets stuck in the border
- Optimising the way that the ball checks collision with bricks (i.e. not checking every brick, but maybe instead only the nearby bricks)
- Maybe removing bricks from the *bricks* array, instead of just marking them as *destroyed*
- Investigating a better way to do levels, perhaps with classes
- Improve the collision between *Batty* and the ball, perhaps with some further research about ways to do this [1]

References

[1] Phat Code. Circle-Triangle Intersection Method. By user 'The_Grey_Beast (Gabriel Ivăncescu)' Retrieved 2025-03-20 from https://www.phatcode.net/articles.php?id=459

[2] GeeksForGeeks. enum in Java. Retrieved 2025-03-17 from https://www.geeksforgeeks.org/enum-in-java/

[3] Processing Forum. Question by user 'cygig' about 'Static methods'. Answers from user 'GoToLoop' helped to resolve the issue. Retrieved 2025-03-16 from https://forum.processing.org/two/discussion/21780/static-methods.html

[4] Processing Forum. Question by user 'railon' about 'Fill with variable hex code doesn't work'. Top answer from user 'GoToLoop' helped by demonstrating that colors can also be hex codes. Retrieved 2025-03-16 from https://forum.processing.org/one/topic/fill-with-variable-hex-code-doesn-t-work.html