INFO1113 / COMP9003   Assignment

Due: 20 October 2024, 11:59PM AEST

This assignment is worth 20% of your final grade.

Task Description

In this assignment, you will create a game in the Java programming language using the Processing library for graphics and gradle as a dependency manager. In the game, balls spawn and move around the screen and the player can draw lines to direct them into holes of the same colour. When balls are reflected off a player-drawn line it disappears. If a ball enters a hole of a wrong colour, score is lost and it respawns. Once all balls are captured by holes, the player wins.

You have been given the task of developing a prototype of the game. A full description of gameplay mechanics and entities can be found below. An simple demonstration of the game and has posted it on your online forum (Ed). You can also play a similar game here, or watch a video of it here.

You are encouraged to ask questions on Ed under the assignments category if you are unsure of the specification – but staff members will not be able to do any coding or debugging in this assignment for you. As with any assignment, make sure that your work is your own, and do not share your code or solutions with other students.

Working on your assignment

You have been given a scaffold which will help you get started with this assignment. You can download the scaffold onto your own computer and invoke gradle build to compile and resolve dependencies. You will be using the Processing library within your project to allow you to create a window and draw graphics. You can access the documentation from here.

Gameplay

The game contains a number of entities that will need to be implemented within your application.

Level

Each level is read from a text file of characters 18x18. The size of the window should be 576x640, meaning each character in the file corresponds to 32x32 pixels.

The level layouts are defined in files provided in the "layout" attribute of the JSON configuration file described below. Each level must have an associated layout file.

Note that the file does not need to contain exactly 18x18=324 characters. It may have less than this if they are not necessary (such as spaces at the end of a line, or missing lines at the bottom). In such situations, your program should still work, and must reflect balls off the edges of the screen.

Figure 1.
The second provided sample level

There are 5 main types of characters that could be present in the file:

- X – denotes a wall (wall0.png). Balls reflect off this, but do not change colour. The game board does not need to be surrounded by walls – balls should reflect off the edge of the screen.
- 1,2,3,4: walls 1,2,3 and 4 respectively, as provided in the scaffold resources. When a ball hits one of these walls, it is reflected and changes colour to that of the wall.
- S – Spawner. Balls spawn from this location (one spawner is chosen randomly of all available spawners in the current level, each time a ball is ready to be spawned).
- H – Holes. The hole takes up 4 tiles, where the 'H' character is the one in the top left. The number in the character to the right of the H is the colour of the hole.
- B – Balls. Instead of spawning after the spawn interval, a ball may be present immediately from the level beginning, at a specific place on the board. The colour of the ball is denoted by the character to the right of the 'B'.
- Spaces – empty space, just ignore it (blank tile).

```
1   {
2     "levels": [
3       {
4         "layout": "level1.txt",
5         "time": 120,
6         "spawn_interval": 10,
7         "score_increase_from_hole_capture_modif
8         "score_decrease_from_wrong_hole_modifie
9         "balls": ["orange", "blue", "orange", "
10      },
11      {
12        "layout": "level2.txt",
13        "time": 180,
14        "spawn_interval": 15,
15        "score_increase_from_hole_capture_modif
16        "score_decrease_from_wrong_hole_modifie
17        "balls": ["green", "grey", "grey", "blu
18      }
19    ],
20    "score_increase_from_hole_capture": {
21      "grey": 100,
22      "orange": 50,
23      "blue": 50,
24      "green": 50,
25      "yellow": 150
26    },
27    "score_decrease_from_wrong_hole": {
28      "grey": 0,
29      "orange": 25,
30      "blue": 25,
31      "green": 25,
32      "yellow": 50
33    }
34  }
```

Config

The config file is in located in config.json in the root directory of the project (the same directory as build.gradle and the src folder). Use a json library to read it. Sample config and level files are provided in the scaffold.

The map layout files will also be located in the root directory of the project. However, sprites or images such as the `ballx.png`, and `wallx.png` will be located in the resources folder (src/main/resources/inkball/) or (build/resources/main/inkball/).

Figure 2.
Example config file.

For each level, the following properties are provided in the config:

- `layout`: the level file containing the characters determining the position of tiles in the grid for walls, holes, spawners and initial balls.
- `time`: the maximum number of seconds this level should last for. If the time is exceeded, the player loses the level and it restarts. If the time is invalid (eg, non-integer or negative value like -1) or missing, there is no timer for this level.
- `spawn_interval`: the number of seconds in between when balls spawn.
- `score_increase_from_hole_capture:` The amount of units score is increased for each ball type when they successfully enter a hole.
- `score_increase_from_hole_capture_modifier:` Multiply the score values gained on this level by this modifier.
- `score_decrease_from_wrong_hole:` The amount of units score is decreased for each ball type when they enter the wrong hole.
- `score_decrease_from_wrong_hole_modifier:` Multiply the score values lost (when a ball enters a wrong hole) on this level by this modifier.

Balls 🔘🟠🔵🟢🟡

Balls may appear in the level layout file, as "B0", "B1", "B2", etc in which case they are spawned immediately in that location when the level begins. Alternatively, they may also be specified in the configuration file, which will cause them to be spawned at a spawner 🔲 throughout the duration of the game. The frequency of spawning is determined by the `spawn_interval` configuration property of that level, which determines how many seconds in between when balls spawn. From being initially at that given value * App.FPS, it counts down on each frame and is displayed in the top bar, next to the display of where balls yet to be spawned appear. The order of balls in this display should be the same as the configuration file (only the next 5 balls yet to be spawned are shown). When the spawn interval counter reaches 0, the next ball is spawned in the game. All other balls remaining yet to be spawned, will gradually move to the left in the display at a rate of 1 pixel per frame.

When balls spawn in the game, they have a random velocity vector which is either -2, or 2 pixels in the x direction, and -2, or 2 pixels in the y direction (assuming 30fps – if using 60fps, this would be halved). Throughout this document, vectors will be notated as `(i,j)` where `i` is the velocity in the x direction and `j` is the velocity in the y direction. Balls collide with walls and player-drawn lines which change their velocity vector trajectory, as described below.

Hitbox

A hitbox is a series of points, which form a sequence of line segments. For example, a player-drawn line may appear as below:
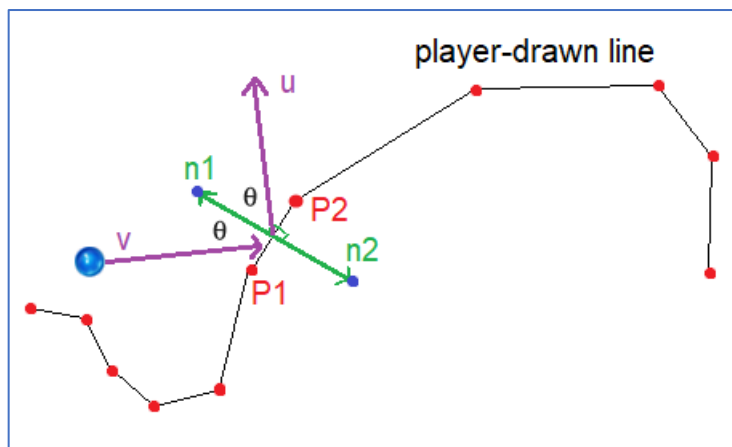

Figure 3.
The hitbox comprises of points (shown in red) that create line segments. V is the velocity vector of the ball, and U is the new velocity vector when it hits the line, shown in purple. N1 and N2 are the normal vectors of the line, shown in green.

The steps to calculate the new trajectory (u) could be as follows:

1. Determine the line segment that the ball is hitting (if the ball will hit any line segments). To do this, check the distance of the ball (x,y) + velocity of the ball, between two adjacent points, P1 and P2. The distance formula is as below:

$$distance(A, B) = \sqrt{(B_x - A_x)^2 + (B_y - A_y)^2}$$

If distance(P1, ball+v) + distance(P2, ball+v) < distance(P1,P2) + ball's radius, then the ball is considered to be colliding with the line segment connecting P1 and P2.

2. Calculate the normal vectors of this line segment, N1 and N2 from P1(x1,y1) and P2(x2,y2). If we define dx = x2 - x1 and dy = y2 - y1, then the normals are (-dy, dx) and (dy, -dx).[1]

---

[1] Source: https://stackoverflow.com/questions/1243614/how-do-i-calculate-the-normal-vector-of-a-line-segment

3. Normalise the normal vectors so that their magnitude is 1. (ie, divide by $\sqrt{i^2 + j^2}$).

4. Find the normal vector on the side of the line that we want to use, either N1 or N2. The one that should be used is the one which is closer to the ball. To do this, perhaps check the distance of the midpoint of the line segment + normal vector with the ball's position (these are the blue points shown in the diagram), and choose the vector which results in a closer distance.

5. Calculate the new trajectory of the ball. This is given by the following formula: [2]

$$u = v - 2(v \cdot n)n$$

Where $v \cdot n$ is the dot product, and n must be normalised – the normal vector of the line segment.

Hitboxes for player-drawn lines are rendered on the game board with a black line of thickness 10 units. Once a collision occurs, the line is removed from the game.

Walls

A wall is a tile with a hitbox comprising of the points of each of its 4 corners. Collision handling for walls will work the same as above for line segments in player-drawn lines. When a ball hits an orange, blue, green or yellow wall, it will change its colour to that of the tile.

Even if the game board is not surrounded by walls at the edges, balls should still reflect off the edges.

Holes

Holes take up 2x2 regular tile spaces (64x64 pixels). When a ball is within 32 pixels of the centre of a hole (from the centre of the ball), it starts to be attracted into the hole. Its size reduces proportionally to how close it comes to the centre of the hole, until when it is on top of the centre, then it will be captured by the hole and disappears. The force of attraction is approximately 0.5% of the vector from the ball to the centre of the hole.

Figure 4.
Your program needs to show a proportional reduction in the ball's size, to give the illusion of it falling into the hole. To do this, specify width and height of the sprite when drawing it. You must calculate the force (ie, acceleration) of attraction by adding a proportion of the vector from the ball to the hole, to the ball's velocity on each frame.

If the hole colour matches the ball's colour (or it's a grey ball, or grey hole), it is a success and the score increases by the amount given in the configuration file, multiplied by the level multiplier. Grey balls are allowed to enter any holes, and balls of any colour can enter a grey hole to count as a success.

If the colour capture was not successful, the ball rejoins the queue of balls yet to be spawned, and score will instead decrease by the amount specified in the configuration file.

---

[2] Source: https://math.stackexchange.com/questions/13261/how-to-get-a-reflection-vector

Player Actions

During the game, players can cause the following actions to occur:

- Press 'r' to restart the level, or if the game has ended because all levels were completed, restart the game. The level returns to its original state, including the timer, balls, and clearing any player drawn lines. Score will return to the state it was at before the level started.
- Press spacebar to pause the game. Balls should not move until the spacebar is pressed again to un-pause the game. The player can still draw lines while the game is paused. To indicate that the game is paused, display *** PAUSED *** in the middle of the top bar.

Players can draw lines with the left mouse button, and can remove those lines by right-clicking over them.

Score and Timer

The score value is persistent across levels. The timer for a level starts at the value specified in the config file, and should count down each second. When it reaches 0, the level will end, display the message === TIME'S UP === in the top bar. The player must then press 'r' to restart the level. In the ended state, balls do not move and the player cannot draw lines.

Level End and Game End

A level ends when all balls are captured by holes successfully, (ie, there are no more balls remaining to be spawned, and no balls currently in play). Any remaining time gets added to the player's score, at a rate of 1 unit every 0.067 seconds. As this is occurring, two yellow tiles which begin in the top left corner and bottom right corner will move in a clockwise direction around the edge of the game board, also at a rate of 1 tile every 0.067 seconds.

When the last level ends, the game ends – display === ENDED === in the top bar.

The user may press 'r' to restart the game.

Application

Your application will need to adhere to the following specifications:

- The window must have dimensions 576x640
- The game maintains a frame rate of 30 or 60 frames per second (physics is frame rate dependent)
- Your application must be able to compile and run using Java 8 and gradle run. Failure to do so, will result in 0% for Final Code Submission. Later versions of Java may work, but you should not use any newer Java language features.
- Your program must not exhibit any memory leaks.
- You must use the processing library (specifically processing.core and processing.data), you cannot use any other framework for graphics such as javafx, awt or jogl

You have been provided a /resources folder which your code can access directly (please use a relative path). These assets are loadable using the *loadImage* method attached to the PApplet type. Please refer to the processing documentation when loading and drawing an image. You may decide to modify these images if you wish to customise your game. You will be required to create your own sprites for any extensions you want to implement.

Extension

The extension is worth 2 marks maximum. For an extension, you must choose to implement one of the following. COMP9003 students must complete an extension that involves multiple additional tile types that have an action associated with them, in order to achieve marks for the extension component.

- Bricks – become progressively more damaged when balls hit them, and then disappear after being hit 3 times
    - Different colour bricks can only be damaged by the ball of corresponding colour, unless it's a grey brick.
- One-way wall. Allows balls to pass through in one direction but not the other.
    - Optional colour can indicate that only balls of a certain colour can pass through
- Colour restricting walls – only balls of a certain colour can pass through, in both directions
- Timed tiles – they progressively become transparent over time
- Acceleration tiles – accelerate the ball in the given direction
- Key wall and key wall activator. When a ball hits a key wall activator, the key wall is toggled to be either retracted (balls can pass through) or solid (balls collide and cannot pass through).
    - Optional colour can indicate that only balls of a certain colour can activate the key wall
- Variations of key wall activator – an object which when hit will a ball, enables or disables holes (show an indication above the hole to show that it's disabled, such as a grate for example)

OR, a feature you come up with which is of a similar or higher level of complexity (confirm with your tutor first)
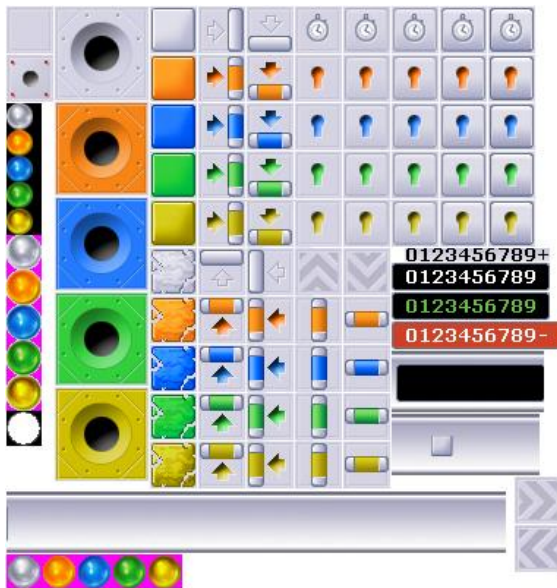
Please ensure you submit a config and level layout file with the features of your extension <u>in the first level</u>, and ensure the extension doesn't break any of the default behaviour. Also, describe your extension functionality in the report.

Figure 5.
The inkball sprite sheet has been provided in the scaffold resources to assist you with the sprites that may be needed for your extension.

Marking Criteria (20%)

Your final submission is due on Sunday 20 October 2024 at 11:59PM. To submit, you must upload your build.gradle file and src folder to Ed. Please also include sample config and layout files that you have tested with your program to ensure it works. Do NOT submit the build folder (unless you only include the build/reports/ folder which contains the results of your testing and code coverage). Ensure src is in the root directory with the other files, and not part of a zip, then press MARK. Submit your report and UML to canvas.

Final Code Submission and Demo (12%)

You will need to have implemented and satisfied requirements listed in this assignment. Make sure you have addressed the following and any other requirements outlined previously. The demonstration is worth 2% and is conducted during tutorials in week 12, where you will be asked to show the functionality to your tutor and they will ask you questions about your codebase and how you implemented the functionality.

- Window launches and shows level layout correctly (empty tiles, spawners and walls).
- Initial ball and hole display is correct.
- Unspawned balls are shown in the top left corner (max 5) and move left 1px/f when one spawns.
- Ball spawn timer, and level time is correct according to the configuration file
- Level time decreases each second, and ball spawn timer decreases each second in increments of 0.1 seconds.
- Balls spawn when the spawn timer reaches 0. A random spawner is chosen.
- Balls have a random (x,y) trajectory when spawned that is (±2, ±2) px/frame and cannot be 0.
- Balls collide with walls, and the new trajectory is calculated correctly to reflect the velocity vector off the surface
- No bugs exist with ball / wall collisions (ie, balls cannot clip into walls, or cling unnaturally to the edge of walls)
- The player can draw lines in the game with left mouse button, which are black and have a thickness of 10 units
- Players can remove drawn lines with the right mouse button or alternatively ctrl+left click
- Player-drawn lines have a hitbox that reflects balls based on the normal vector of the line segment that's hit. When a collision occurs, they are removed.
- When a ball comes close to a hole, it is attracted towards it with a force proportional to how close it is
- When a ball comes close to a hole, its size reduces proportionally to how close it is to the hole
- When a ball is directly above a hole, it is captured by the hole
- When a ball of a different colour to the hole is captured by it, the ball enters the respawn queue, unless it is a grey ball or grey hole
- Score changes correctly when balls are captured successfully or unsuccessfully, to increase or decrease respectively based on the colour of that ball and the score values specified in the config file, including the level multiplier.
- Spacebar causes the game to pause, and the top bar displays *** PAUSED ***
- The current level ends in a win when no balls remain to be spawned, and no balls are currently on the game board.
    - Remaining time gets added to the player's score at a rate of 1 unit every 0.067 seconds.
    - Yellow tiles originating in the top left corner and bottom right corner move around the edge of the game board in a clockwise direction at a rate of 1 tile every 0.067 seconds.
- When the level ends in a win, the next level is loaded.
- When the level timer reaches 0, the level ends in a loss, meaning balls stop moving and the player can no longer draw lines. Display === TIME'S UP === in the top bar.
- The player can press 'r' to restart a level at any time, including when time has run out.
- Once the game has ended, a player can restart the game by pressing 'r'
- Ensure that your application does not repeat large sections of logic
- Ensure that your application is bug-free

Testcases (3%)

During development of your code, add testcases to your project and test as much functionality as possible. You will need to construct unit test cases within the src/test folder using JUnit. To test the state of your entities without drawing, implement a simple loop that will update the state of each object but not draw the entity.

Ensure your test cases cover over 90% of execution paths (Use jacoco in your gradle build) – average of branches and instructions. Ensure your test cases cover common cases. Ensure your test cases cover edge cases. Each test case must contain a brief comment explaining what it is testing. To generate the testing code coverage report with gradle using jacoco, run "`gradle test jacocoTestReport`".

Design, Report, UML and Javadoc (3%)

You will need to submit a report that elaborates on your design. This will include an explanation of any object-oriented design decisions made (such as reasons for interfaces, class hierarchy, etc) and an explanation of how the extension has been implemented. This should be no longer than 500 words. This report will be submitted through Canvas.

You will need to submit a UML diagram in PDF form to Canvas to provide a brief graphical overview of your code design and use of Object Oriented Principles such as inheritance and interfaces. Markers will use this to determine whether you have appropriately used those principles to aid you in your design, as well as figure out whether more should have been done. A general guideline is that markers will be looking for some use of inheritance or interfaces, how extensible the code is, and penalising repeated code. Note that you should not simply use a UML generator from an IDE such as Eclipse, as they typically do not produce diagrams that conform to the format required. We suggest using software such as LucidChart or draw.io for making your diagrams.

Your code should be clear, well commented and concise. Try to utilise OOP constructs within your application and limit repetitive code. The code should follow the conventions set out by the Google Java Style Guide. As part of your comments, you will need to create a Javadoc for your program. This will be properly covered in week 11 but the relevant Oracle documentation can be found here.

Report, UML and OO design: 2%
Javadoc, comments, style and readability: 1%

Extension (2%)

Implement an extension as described above. Partial marks may be awarded if you choose a more limited extension or it is partially completed. Please specify what extension you decided to implement within your report, and show it during your demo in week 12.

Suggested Timeline

Here is a suggested timeline for developing the project. Note that it is released on September 10 (start of week 7) and due October 20 (end of week 11).

Week 7: Familiarise yourself with gradle and processing, utilising the processing Javadoc and week 8 supplementary lecture. Identify opportunities to utilise Object Oriented Design principles such as inheritance and interfaces and begin to plan a design for the codebase with regards to the classes that you will need to make. Make a rough UML diagram for your design that you can base your codebase from.

Week 8: Begin writing the actual code for the program. Start small, for example by initially creating the level layouts and tiles, then gradually add more like balls and spawners. At the end of the week, you should have loading in the map and ball movement finished, as well as some sprite management. If confident, use Test Driven Development (writing test cases at same time as writing the code). Conduct a large amount of user testing to ensure the initial mechanics work as expected.

Weeks 9-10: Develop more gameplay features, such as the collision handling, player drawn lines, ball spawning and scores. Sprite management should be streamlined at this point. You should have a fairly high code coverage for your test cases at this stage. If you are noticing any questionable design decisions, such as God classes or classes that are doing things they logically should not be doing, this is the time to refactor your code. Think about what extension you want to make and start to implement it.

Week 11: Finish developing the remaining features for your program, notably the configuration file, GUI enhancements, timers and level progression. Additionally, finish writing your testing suite. Create the UML and Javadoc for the program. Fix any remaining bugs that your code exhibits. Submit your code to Ed (by uploading the entire project and pressing MARK) and submit your UML to Canvas in PDF form.

Week 12: Demonstrate the completed program to your tutor during the week 12 lab. They will check each criteria item has successfully been completed, and may ask you questions about how you implemented it to test your understanding.

Academic Declaration

By submitting this assignment you declare the following:

I declare that I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure, and except where specifically acknowledged, the work contained in this assignment/project is my own work, and has not been copied from other sources or been previously submitted for award or assessment.

I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to severe penalties as outlined under Chapter 8 of the University of Sydney By-Law 1999 (as amended). These penalties may be imposed in cases where any significant portion of my submitted work has been copied without proper acknowledgment from other sources, including published works, the Internet, existing programs, the work of other students, or work previously submitted for other awards or assessments.

I realise that I may be asked to identify those portions of the work contributed by me and required to demonstrate my knowledge of the relevant material by answering oral questions or by undertaking supplementary work, either written or in the laboratory, in order to arrive at the final assessment mark.

I acknowledge that the School of Computer Science, in assessing this assignment, may reproduce it entirely, may provide a copy to another member of faculty, and/or communicate a copy of this assignment to a plagiarism checking service or in-house computer program, and that a copy of the assignment may be maintained by the service or the School of Computer Science for the purpose of future plagiarism checking.