Grade

Space Adventure

Team: Cindy (Xinzhu) Fang, Jenny Kim, Justin Warring
May 2, 2018

Functionality

The program compiles without error. We have created all the situations we can possibly in this game and we have annihilated all the errors came up. It should ignore you when you sit on the keyboard unless you move around in a specific pattern, for example, sit on “tab” ,“tab”, and then “enter”, the game will exit so that you cannot play it.

Design

Our program is well organized, well documented and concise. The design we have used for this program, as further described in our program specification document, tries to emphasize clarity of code, organization, and documentation. Because we have just extended Sphere for all our object classes (sans Debris and Bullet which are extended from Asteroid), we have reduced the need to repeat code in various classes. We have also further organized individual methods for each object into a larger method for both updating objects and drawing them in the world, which helped with also reading what we coded for.

We also had a collision method that was slightly over-engineered, especially because it was determining collisions by testing every point on the outline of a shape. We replaced this with Shapes, which is a java feature that determines for us whether there is an intersection. This way, we didn't use the less precise method when we had access to a more accurate way of detecting collisions given that we were working with fairly simple objects in this game.

We also kept in mind the various functionalities that a player would want within a game, including an opening instruction message that helps the player with the controls, another message that checks if they truly want to play the game (allowing them to exit if they decide to change their mind), an exit button within the game, and a pause and unpause button. These were all functionalities designed with the purpose of making it easier for the player to play the game.

The planet is supposed to die in the death area not in the space. Given more time, we will figure out why it dies early.

Creativity

By generating a completely new game, albeit influenced by famous games like Snake and Asteroids, we believe that we have indeed met the creativity aspect of this project. Unlike Asteroids, we have implemented a way for players to rescue other ships, thus forcing players to not only attack asteroids for itself but also to defend the other ships. We felt that the interplay between attacking and avoiding and the balance between the reward and expense on picking up a ship would get the player to make more complex decisions during the game.

The game may be used for psychological studies on decision making. Also, we can record the brain wave using an EEG cap while participants playing the game to investigate the mental representation of fluke and failure, the neural correlate of anger and surprisal. etc.

In most other action games, arrow keys are used to change velocity, which is more controllable but less realistic. In our game, arrow keys are used to apply acceleration emulating the effect of expelling burning fuel into the vacuum. The velocity of myShip wouldn't change with no key pressed obeying Newton's first law. As a consequence, myShip is comparably harder to control. However, through infinite times of debugging runs, the code monkeys are now so much more adept at playing this game. This game could also be adopted in studies on implicit memory. As the player folster a better estimation on the consequence of pressing an arrow at a given velocity for a specific duration, it is reasonable to hypothesize that a classifier trained to predict an action with clusters of fMRI data would have significant increase in decoding accuracy in the brain region that concrete action planning takes place.

The game only accepts one player at this point. Given more time, we will implement the game so that it accepts two players, who would have both antagonism and synergy relationships depending on the situation. For example, the other player is your enemy considering that it takes away the available free ship; meanwhile, you can hide behind its trailing captured ships from asteroids. In our original plan, we would limit the amount of ammunition rather than allowing the player to shoot as much as it wants, given more time, we would implement this too.

Also, with more time, we could make the dialogue window before the game start bigger and instead indicating lives with numbers, we would just draw four ships for four lives, which is more obvious. Visualization is actually easy, it's just we are running out of time. We would also do more research on threads. We are barely handling three threads using some booleans; we probably need to do more manipulations such as waiting and synchronizing rather than adding infinite threads and conditions when the game become more complicated. We would also adding I/O so that players can save games to local and resume later.

Broadness

1. Java libraries that we haven't seen in class
    - To detect overlap for collision among asteroids, bullets, and ships, for MyShip's picking up other ships, and for MyShip's rotation using affine transformation, we created a myShape field for every Sphere (Asteroid, Ship, Debris have Polygon; Planet and Bullet have Ellipse2D) that implement Shape, who is also implemented by Area. This way, we can measure the intersection among all Spheres using the intersect() method of Area.
    -
2. Subclassing (`extends`)
    - Asteroid, Planet, and Ship inherit from Sphere; MyShip inherits from Ship; Debris, and Bullet inherit from Asteroids. Using inheritance saved us lots of unnecessary repetitiveness when building the classes since they share several fields and methods, and when calling their fields or methods in World since the overriding fields and methods have the same name as their parents'. By setting the shared Sphere fields static and overloading Ship constructor, all the other entities get the essential parameters as soon as myShip is initialized without asking for them at birth.
3. Interfaces (`implements`)
    - We had to implement Runnable to keep the game running.
    - We had to implement KeyListener to process keyboard input.
4. User-defined data structures
    - We used our own generic Linked List to store asteroids, bullets, freeShips, and capturedShips in the world. Being generic was actually not much of use. Since all the entities in the world inherit form Sphere, we conveniently just cast all the elements in the Linked List to Sphere.
    - We used Pair to store all vector data: acceleration, velocity, and position.
5. Built in data structures
    - In World.checkCollision(), we first store the index of colliding entities in an ArrayList of Integers, then call World.magicSort(), who takes values from ArrayList one by one, generates a Set of them with no repetition, hands them to an Array of int one by one, and then return the Array.
6. Randomization
    - There are multiple places where we used randomization, the arrival time of next planet and next asteroid, the initial position of asteroids and planet, the moving direction of asteroids and debris, the shape of asteroids and debris, and the increasingly increase of difficulty over time.

7. Generics
    - As said, we thought we would need Generics for having multiple classes but that turned out unnecessary.

Sophistication

We have multiple threads: Runner threads, KeepPlanetComing thread, and KeepAsteroidsComing thread.

World.collision is a nested loop, loops through all the asteroids at outside. At inside, first, it loops through all the previous asteroids (an upper triangular matrix); second, it loops through all the bullets; third, it loops through all the captured ships; at last, it checks on myShip.

We believe in terms of sophistication, we have developed code that is well-documented, well-organized and fairly complex, having to take into account multiple objects in a space in addition to constant updating of the world.

Code quality

Code Quality

- Commenting throughout
    i. Most of the naming is pretty clear, allowing users to understand what each method or variable is used for. Important comments are dispersed throughout the code along with url for citations
- Code organization across files
    i. We used differences in capital case throughout the files. All classes start with capital case, all variables and methods start with lowercase. Due to a strict hierarchy in the entities classes, the main methods: update(), updateShape(), and drawShape() and some other fields are consistently overridden in subclasses.
- Code cleanness
    i. All abandoned code or code for debugging were cleaned out

Product Quality:

We have a welcome message at the beginning of the game, we ask for their consent before we expose the game (no unsolicited visual distraction). We have a panel of buttons on top of the game panel so that they can pause and resume the game. The game is challenging (we don't underestimate our players). Players gain lives if they do well, they lose lives if they don't, and they will "die" if they lose all lives. The difficulty level also increases over time, which means asteroids and planets move faster (i.e., the free ship on planet moves faster), bullets move slower. Lots of the parameters are randomized, which makes it less hackable and less boring.