Assignment Report of Asteroids

Group Name: Try One Try

Group Members:

Ma, Benteng	18206096	Wang, Yushi	18206384
Yu, Xiaoxiao	18206362	Cao, Guangzhao	18206100
Liu, Xuan	18206361	Yang, Shuwen	18206359

Basic Introduction of the Game Design

The main process of making this game began on Nov. 20th, and on Dec. 4th, we finished most of the things. Most of the solutions are created by ourselves, and some ideas are learnt from the example code. We are proud to see that after half-month hard working, our game works perfectly well, and we had a very pleasant cooperation during this time. Therefore, we would love to use a much longer report to record this precious experience.

Responsibilities of Components and Detailed Introduction

Implementation of screen display – Cao, Guangzhao, Ma, Benteng & Wang, Yushi
Design of game structure and game loop – Ma, Benteng
Player Listener – Cao, Guangzhao
FPS controller & game timer – Ma, Benteng
Collision judgement – Yu, Xiaoxiao
Design of bullets' movement & collision area – Ma, Benteng
Design of asteroids' shape & split – Liu, Xuan & Cao, Guangzhao
Asteroids' showing up & movement – Liu, Xuan & Cao, Guangzhao
Design of player's ship's shape, rotate, hyperspace jump & rebirth – Wang, Yushi
Player's ship's control & movements: acceleration & friction - Ma, Benteng
Design of alien's ship and its movements – Yu, Xiaoxiao & Wang, Yushi
Rank of scores and recording – Yang, Shuwen
Dialogue windows display – Wang, Yushi & Ma, Benteng
Game testing on sample website and of our own – Yu, Xiaoxiao

Special thanks to Liu, Xin, Chinese University of Hong Kong, Shenzhen, for drawing two of our amazing background pictures.

Game Content

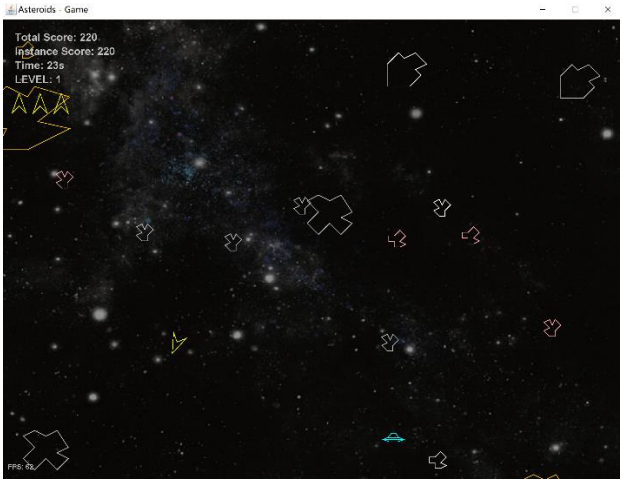
When running the program, the main menu will appear on the screen, player will interact with it by using the keyboard. To exit, player should press “X” and the window will be closed with the program stopped. Pressing “I” so that the second interface will be shown, telling the player how to play. If the player would like to go back to the main menu, he or she can press “M”. To see the rank of top ten players, the player can press “H”, then the third interface will be shown with all names and corresponding scores which are sorted, same as last one, press “M” to return to the main menu. Finally, the player can start a new game by pressing “N”. After starting a new game, the fourth interface comes into view which is the gaming interface.

There, the player can control the movement of the spaceship which is initially on the center of the screen by pressing “left”, “right” and “up”. The ship will jump to another safe area once “shift” is pressed, which is called “hyperspace Jump”. Bullets are fired when “space” is pressed.

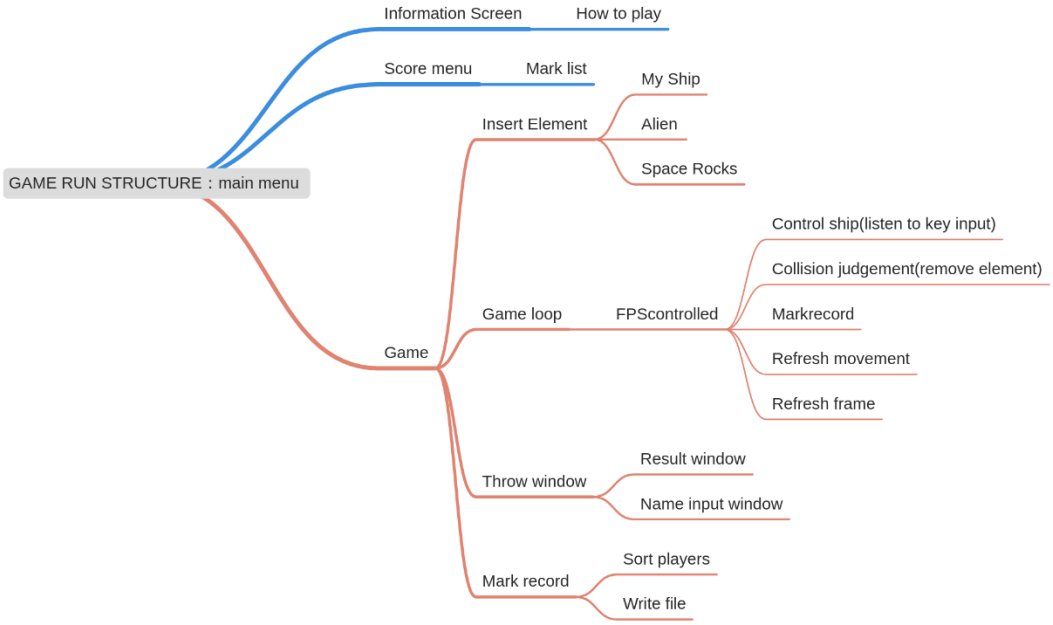
By killing enemies, the player earns scores. Enemies including asteroids and alien ships will fly into the window randomly. Alien ships have two types: larger and smaller, both of which can shoot bullets. As the level increases, the number of asteroids – space rocks will go up.

Furthermore, all those elements are displayed above an amazing “universe” background. Notice that the scores of the player will be displayed on the top left of the screen, which includes total score, instance score, the time in seconds he or she have played, the level reached at present and lives the player has left. At bottom left, fps at the moment is displayed. When the ship crash onto enemies or shot by aliens, player will lose lives and back to the center of the window with a short invincible time given when the ship will flicker. When there is no life left, the player will be asked if he or she would like to buy a new life using “instance score” rather than “total score” so that it would not affect the rank. However, the more times buying a new life, the more costly it will be.

Finally, when either the player has no life left or cannot afford to buy a new life, the game will ask the player’s name and then shows how many bullets was fired, the type and the number of the enemies was killed as well as the hit-shoot rate.

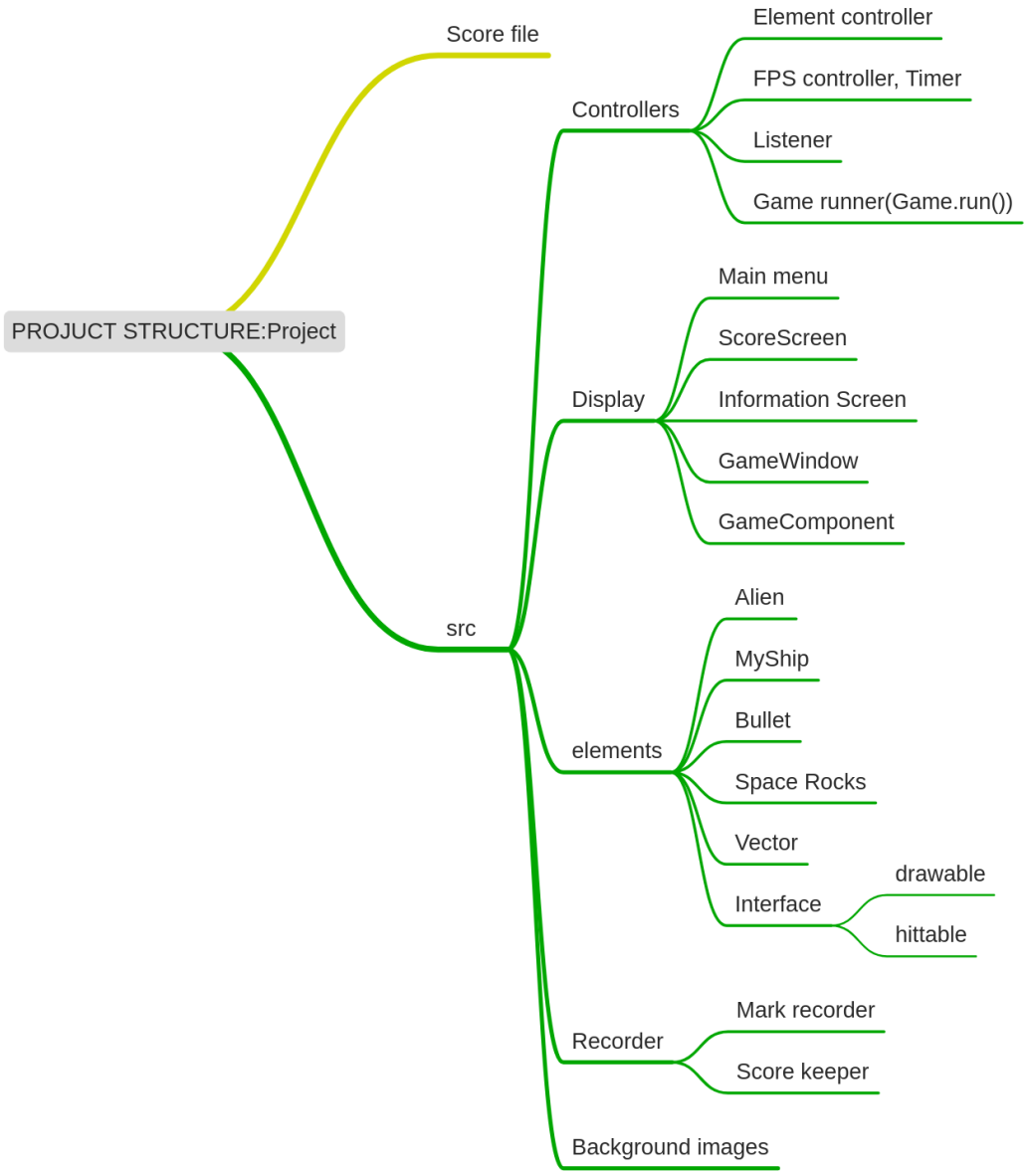


Structure of the Program



The game is run with a game loop which the frequency of the refreshment is controlled by the FPS controller. In each level the element controller will insert new elements include aliens and space rocks into the game environment before the loop starts, each turn of the game loop includes the controlling of the ship movement, refresh of the movements, judge collision, record score and refresh the screen. Different dialogue window will be thrown out during different occasion and the score will be sorted and saved by the score keeper after the game is finished.

Structure of Files



The source file includes four packages: controllers, display, gaming elements and recorder. Background images are also stored inside the source file. Controllers are mainly the elements helps the game to run. Display package includes main menu, score and info screen, game window and game component. Gaming elements are alien, ship, asteroids and bullets

implementing two interfaces. Vectors are very helpful for movement calculation, and enum type Size, which is not included in this chart, is used for distinguish the size of asteroids and alien ships. Mark recorder in the recorder package is used to record the score during the game, and score keeper ranks the scores, reads and store them. The score file is out of the source file.

Detailed Introduction

Implementation of Screen Display

The basic game window, main menu and info score menu is implemented with JFrame. The polygenes, and strings are painted with Graphics, which works quite familiar to a real pen. However, it is discovered that painting directly onto JFrame can cause flicker, so JComponent was used for the game window to avoid it; for info score menu, JPanel was used to print strings. The difference between these two is that one is transparent but the other is not. To refresh, repaint() method of the window – JFrame is called, and this method will call paintComponent() method of the component and the panel, to paint with Graphic.

Design of Game Structure & Game Loop

At the beginning, because we were not familiar with any graphic use of Java, most of our work began around JFrame. Having no experience, we did a lot of experiments including opening a window on the screen, draw a rectangle on it and move it with keyboard, etc. As a result, the game structure was quite different from now, that there was no controller of game itself, and the frame itself took charge of all the game loop and the movement of elements.

However, separating the gaming running part from the graphic part becomes very important when it came to collision judgement and settings of levels. Therefore, an element controller was created first and then the Game class which is in charge of game loop. The score recording system is independent from game loop, making itself convenient to be called by both the game loop and the scores window. Main menu, score & info window are separated from the Game class, the main menu is able to call a new game, info window and scores window, or exit the game.

The conception of game loop was learnt from a python online lecture during last summer holiday. During the class the teacher recommended PyGame package to practice python coding with making a very small game. That is where the conceptions of game loop and FPS control were first learnt.

However, design of this game loop is totally original creation. It combines control, movement calculation, collision and other judgements, with refreshment of screen together, which allows the elements move in a steady speed and be controlled, in an unchanged frequency. It is also responsible for judge whether the game is end, and deliver the player's game result to score keeper.

FPS Controller

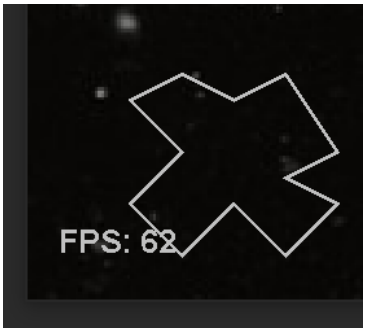
The idea of creating an FPS Controller to control the number of frames in a second during the game came from many real-time games like DOTA, CS and League of Legends, for which the refresh frequency of the screen can be a very important factor which influence the gamers' performance. Although Asteroids is not such a complex game, providing a proper frequency is still very helpful to keep the game running smoothly, as well as making it much easier to control the speed of every element in the game.

Proudly, the whole implementation of FPS controller is created by ourselves, without borrowing any existed ideas. Despite those hot-selling gaming screens which are able to provide 144 Hz or even 240 Hz refresh rate, most computer screens refresh 60 times per second, and it is predictable finishing all the calculations for each turn of our game loop takes much less than 1/60 seconds, so the main job of this controller is to let the computer wait for proper time before starting another turn of loop.

Therefore, the main process of this implementation is like:

```
set waitingPeriod to be 1000 / expectedFPS
get lastTime from system
loop:
    get newTime from system
    if newTime - lastTime is over the expected waitingPeriod:
        lastTime <- newTime
        start another game loop
    else:
        wait a short period
```

However, in the real code, the method – timeCheck() does not provide a loop or a waiting method itself, but it tells the game loop whether to begin a new refreshment or to wait for a short period and check the time again. It returns true to allow the game loop to continue, or a false to make it sleep. By adding a variable inside the method, the actual rate in each turn is calculated and displayed on the screen. During the whole game, it is quite steady to be between 62 and 58.



In addition, the possible situation is also considered that one turn of the game loop might take time which is longer than the refresh period - almost never happens but worth to think. The immature solution of it is to calculate the number of time gaps – number of periods that has been missed during the last refreshment, and fill this gap by keeping refreshing for this number of times. However, it still takes a long time for calculations during these turns, so when they take place, only the displacement of the game elements is considered instead of any collisions or commands from the player. As a result, the ship loses control for a short period and the collisions may not be very accurate, but the frames in front of the player are expected to be fluence.

The process is like:

```
listen to the player
refresh the position of all the elements in the memory
judge collision
do:
    refresh screen
while gap exists, do again
```

Unfortunately, this solution does not really work as expectation, but after all, our testing environment is also harsh: When adding 1,000 large rocks into the game environment, the FPS drop to around 30, and when rocks become 10,000, it drops to 3. Actually, because the length of existence of bullets are designed to be based on a time limit, when they move slower, their range also decrease. Of course, no body can make it to hundreds of rocks, but it is a considerable problem if making a much more complex game.

Game Timer

Compared to FPS controller, game timer seems so much easier, with only 10 lines of code. But this small element is used widely in the whole game. It starts counting time when it is constructed, and it returns a true when the time is up. The difference between it and the sleep() method in Threads is that sleep() method pauses the whole gaming process, but this timer is able to count time by the time the game is running normally. It is used to control the range of bullets; it also works for keeping the player's ship to be unkillable every time it is reborn.

The process is easy:

```
when it is constructed:
```

```
    zeroTime <- read time from system
    difference <- the expected time length
when timeCheck() is called:
    if difference > a new time read from system – zeroTime:
        return true
    else:
        return false
```

With this process, the timer can work independently from the game loop and tell the time.

Key Listener

“key listener” can be treated as a bridge which connect the user with the game elements - to be exact - the player’ ship. All functions of this class can be concluded to be: helping the computer receive the message from the keyboard and then pass it to other methods to let them do actions.

To achieve it, java.awt.KeyEvent and java.awt.KeyLlistener were first imported. Then a new class called “PlayerListener” which implements “KeyListener” was built. Furthermore, we override “KeyPressd” as well as “KeyReleased” in class “KeyListener” and passed the KeyEvent as the parameter into those methods. When player press a button on the keyboard, the methods above will change the corresponding Boolean value to “true” or vary it to “false” when the button is released. Those Boolean values could be used by “if” or “while” in other classes. For example, if the corresponding value for “up” on the keyboard is pressed (true), then other method could do some operations like: “x = x + 1;” (if (“up” is true) { x = x + 1; }) .

Collision judgment

Collision judgement is the most important part in the game loop. With the help of hittable interface, the element controller is able to have a unified control of all the hittable elements.

The collision in the game has a complicated logic, because there are three object: Myship, Alien and Space rock, each of them can hit by another. At first, we considered to put these judgment methods in their own classes, but we found that it is hard to control all of them, so we created hittable interface which provides methods that are able to get the collision area and let elements kill themselves, cooperate with Element Controller. The collisionJudgement() method in the class combined six possible collision. The judgement of the collisions import java.awt.geom.Area and use intersect to find if there is a superposition between two areas. It is noticeable that among the judgment, we added the score indicator in it. Also, in order to let the bullet shoot by player’s ship cannot hurt itself, an alien judgment was added.

Vector class

Vector class was originally designed to be used as assistant class to help with the movement of the ship. However, its huge benefit was discovered later and it becomes the mostly used class that serves for the calculation of movement for many elements.

The vector does not consider of any real position, it records displacement only. The instances of vector class includes displacement, speed and acceleration.

The method getModule() returns the actual length of the vector, with a^2 = b^2 + c^2.

The static method toVector() can be used as a constructor of vectors with an angle and a length given as parameters. The calculation is based on trigonometric functions.

Bullets

Bullets class implements Hittable and drawable interfaces.

This class actually contains two types of bullets shot by player's ship and aliens. They have different speed and range. In addition, the speed of bullets shot by player is partially based on the ship's own speed, while for aliens' their own speed is not considered since they move much slower than the player.

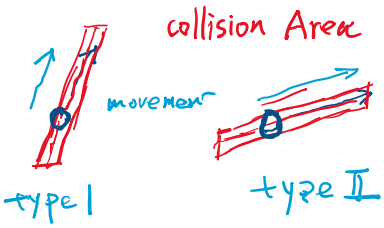
The shape of the bullet is originally designed to be a hexagon; however, it is so small that it looks like a dot.

We think it is quite ingenious to create a static link list in the class to contain all the bullets exist in a moment. When a new instance is constructed, it puts itself into the list; as well as using the method to kill itself.

The bullets is designed to come back from the other side of the screen once they exit it from one side.

When alien or player's ship call the shoot() method, they give the bullet a start point and the angle of direction, for ship, the bullet is also given the speed vector of the ship itself. With this given data, it creates its own speed vector, and make its movement with it. Like other elements, it goes to the other side of the screen when getting out of it from one side.

A difficult problem is the judgement of collision of bullets. They are too small and their movement of every turn is much longer than their own size. As a result, they may “cross” other elements instead of hit them. To solve this problem, another ingenious design is used: for each bullet, a special parallelogram is prepared to be used as their collision area. This parallelogram covers all the area this bullet moves during each turn, so that they still hit even though they actually “cross”.



In drawBullets() mothed, all the bullets are actually painted onto the component together, by the same time, is any of them are discovered to be killed, they will be removed from the list.

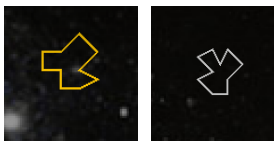
Finally, the speed of bullets shot by aliens and ship is different. The control of the range is based on setting a timer when the bullet is shot, when the time is up, it kills itself and disappear.

Space rock (Asteroids)

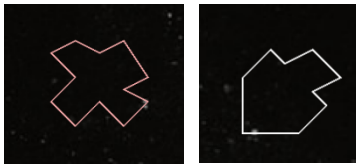
SpaceRocks class implements two interfaces: drawable and hittable.

Shape

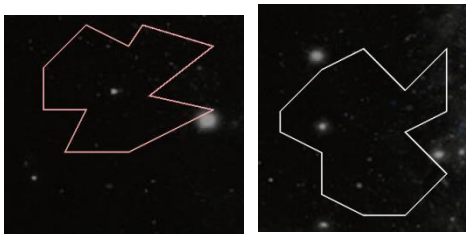
There are three different types in the SpaceRocks class: Small, Medium and Large, with each type of them has diverse polygon appearance. Besides, there are four gorgeous color for every single randomly.



(Small Rocks)



(Medium Rocks)



(Large-Rocks)

Similar to bullets, SpaceRocks class has its own static link list to store all the types of asteroids, and an enum type, Size, is created to distinguish different sizes of asteroids.

Split

The spilt of rocks is implemented in drawSpcaceRocks() method, when any of the large and medium rock is found to be killed, the list removes it and insert three new smaller one, with position of the dead rock and a random direction. It is very convenient that one the instance is constructed; it handles everything of itself automatically.

Movement and appearance

Firstly, we set the center of the rocks as objectpos [0] and objectpos [1], representing x-axis coordinates and y-axis coordinates respectively, meanwhile, the random speed is given to rocks with the code: “Double speedmodule = random. Nextdouble() + 0.3” and Double angle; The speed vector of the Space-Rock is determined by the module speed and angle together. Besides, we use the do-while cycle to achieve a more perfect angle of incidence. The code is shown below:

```
do {
   angle = random.nextInt(360);
      } while ((angle <= 105 && angle >= 75) || (angle <= 285 && angle >= 255) || (angle <= 15
&& angle >= -15) || (angle <= 195 && angle >= 165));
```

When every new level begins, the large rocks will enter into the screen from four sides randomly, these rocks will also be given a random angle of movement with a random speed, with the assistance of vector.

Secondly, in order to allow Space-Rock and random entry the window, we defined four sides of the screen as: 0 means up side; 1 means down side; 2 means left side; 3 means right side. Using the switch method, let's take one of them as an example:

```
switch (enteringSide) {
case 0:
    angle = random.nextInt(120) – 6;   // Setting random angle
    this.movementVector = Vector.toVector(angle, speedModule);   // Direction of motion,
random angle and random velocity of Space-Rock
    this.objectPos[1] = -50;   // Setting the initial position of y be - 50
    this.objectPos[0] = random.nextDouble() * Game.WINDOW_WIDTH * 0.8 +
    Game.WINDOW_WIDTH * 0.1;   // The position of y is 10% - 80% of the window width
   break;
```

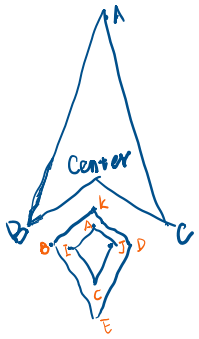
Thirdly, to make the movement smoother, we use switch again and add the code of “Objectpos [1] > game.WINDOW_HEIGHT + number” in each case, potentially expanding the window so that the rock will not disappear and flash when the center of the rock touches the window’s sides. Therefore, like other elements, the rocks also come back into the screen once

they leave it from the other side.

Myship Class

Design and rotate

In the design of the ship, we did not use pictures to draw, but the polygon drawing method. This is closer to the original game. But when it came to designing the rotations of the spacecraft, it was more complicated.



We set one point in the center of the spacecraft as a standard point, and the other points move as the center rotates. The mathematics is used here to calculate the trigonometric functions.

For ship:

A: (CenterX+a$\sin(angle)$, CenterY+a $cos(angle)$)

B: (CenterX-b$\sqrt{2}\sin(45+angle)$, CenterY-b$\sqrt{2}\,cos(45+angle)$)

C: (CenterX+b$\sqrt{2}\sin(45-angle)$, CenterY-b$\sqrt{2}\,cos(45-angle)$)

We also designed a small flame here that can only be displayed when accelerated. Small flame can also move as the center point rotates.

For flame:

A: (CenterX -a/3$\sin(angle)$, CenterY-a/3 $cos(angle)$)

K: (CenterX -a/4$\sin(angle)$, CenterY-a/4 $cos(angle)$)

B: (CenterX-b$\sin(angle)$-(b/2-1)*1.3 $cos(angle)$, CenterY-b$cos(angle)$+(b/2-1)*1.3$\sin(angle)$)

C: (CenterX -a$\sin(angle)$, CenterY-a$cos(angle)$)

D: (CenterX-b$\sin(angle)$+(b/2-1)*1.3 $cos(angle)$, CenterY-b$cos(angle)$-(b/2-1)*1.3$\sin(angle)$)

E: (CenterX –(a+b)$\sin(angle)$, CenterY-(a+b) $cos(angle)$)

I: (CenterX-b$\sin(angle)$-(b/2-1)*1.3 $cos(angle)$, CenterY-b$cos(angle)$+(b/2-1)*1.3$\sin(angle)$)

J: (CenterX-b$\sin(angle)$+(b/2-1)*1.3 $cos(angle)$, CenterY-b$cos(angle)$-(b/2-1)*1.3$\sin(angle)$)

When we first designed the spacecraft to rotate, When the ‘keylistener’ hears that the left and right keys of the keyboard are triggered, the angular velocity increase 1°per frame. In practice, however, we found that this angular velocity was too high, causing the player to be unable to accurately control the ship's rotation angle. In order to keep the game flowing smoothly, we finally designed to rotate each frame by 4°.

Hyperspace Jump

For the ship, when the player set off the ‘shift’ key, the ship jumps to a ‘safe position’. We used the reverse thinking here. Since it is hard to say ‘no collision’, remove all the point of collision and the safe position would be left.

In order to ensure that the ship jump at different angles would not has a collision position, we calculated the maximum side length of the ship and made it the side length of the standard square. Then we divided the screen window with standard square and treat each cut square as a "colliding object". Crash test each object against aliens, bullets and meteorites, and each

available square is listed in a linked list. Number the available positions, one of the numbers is randomly selected. Finally, moved the ship to the specified center point. We used a loop to ensure that each jump ship would not appear at the boundary point and allows players to quickly confirm the ship's position.

Resurrected and immune damage

Every time the ship is killed, it will come back to life with a 4 second flash. During this period, the spacecraft could immune damage. We used class Timer() here.

When the ship was killed, the parameter isDestroyed comes to be true. The position of the ship becomes the starting point, the speed and angle return to zero. During resurrection (4 seconds), the ship will skip all crash tests and remain invincible. We used the current time and death time difference to make the flash effect. It controls the timing of each appearance and disappearance.

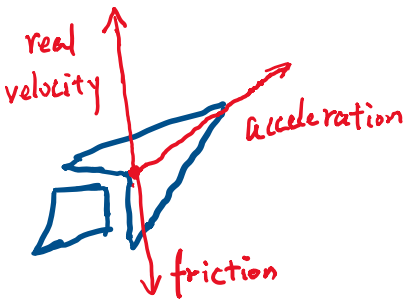
Shoot

Since the bullets are able to control themselves, the main job of the ship is to determine the position that the bullet is shot, with trigonometric functions. The direction of the bullet is based on the angle of the ship itself; and the speed vector of the bullet is added with the ship's speed vector. We would talk more about it in the part of bullet.

Movement

The ship has ability to accelerate and rotate, when rotates to the opposite, the acceleration works like deceleration. When pushing forward, the ship has a maximum limited speed; as well as a counter-acting force works like friction. It took a long time and a lot of attempts to make the speed limit and friction work smoothly. Although not required in the instruction, adding speed limit and friction is very helpful to make the ship easier to control.

The ship has two vectors works to help with movement, one is speed vector, which does not point along the head of the ship, but the real direction it is moving. The other one is acceleration vector, which points to the front of the ship when the ship turn on its engine. If no friction is added, the ship moves like sliding on a total slick platform.



When the ship is moving, its speed vector is divided into two directions with orthogonal decomposition. The percentage of speed decrease happened from these two directions with three different level depends on the actual speed, calculated by getting the module of the vector. When the speed is already very small, a much stronger friction will be added, and turn the speed to zero. The limit of the speed is also made by adding a very strong friction, offsetting the acceleration. With different level of the friction, it moves just like a real plane, the reason it reaches the max speed is not because the pushing force disappears but because of the strong friction – like any moving objects, the

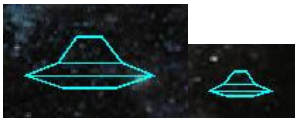
friction rises as soon as its speed increases, when the friction equals to the push force, the acceleration drop to zero.

Gladly to see that with these well-designed factors, the ship is able to do any complex movements like drifting, in a perfectly smooth way.

Alien

Design

We use polygon method to draw the two types of alien ships: large and small, with the enum type Size.



Movement

The alien can enter into the screen in a random side and high. This part of the code is displayed in the method createNew. Once it has been killed, a new alien in random size will appear in 4 seconds. The movement of the alien is a difficulty, it not only contains a translational motion, but also can move with an angle, so that means we need to use random method to decide its move direction.

Unlike other components, the alien only flies horizontally once in the game window. If it flies out from left or right boundary, it wouldn't come back from the other side. The alien horizontal movement is divided into 5 segments by the width of the window. For each cut-off point, we used random number function to change the speed and direction of the movement. To avoid the vertical speed be too fast, we used a second FPS controller with FPS of 30 to reduce the speed per second, to changed the alien's vertical movement from moving 60 times per second into 30 times. Particularly, to prevent the alien from staying too long at the boundary, when it approaches the boundary, it's vertical speed increases, making it back into the center of the screen in a shorter time.

Shooting

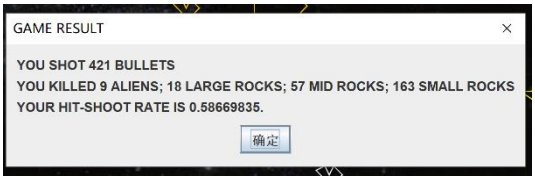
The big alien which is in low intelligence can fire a bullet in a random angle. The small alien which is much more sensitive and intelligence can shoot to the direction of the ship. In this part of the code, we use Math.atan() and Math.toDegrees() to get the accurate angle between those two objects. However, after testing the game, we found that it is too difficult to play against the small alien, so we add some error to the shooting angle and when the small alien shooting, the bullet will have deviation.

Mark Recorder

Mark recorder is used for recording the score during the game is being played, with mainly two static variables. Different scores for different elements will be added when addScore() method is called.

To print the total score, instance score and time onto the screen. We used Graphics to let this information appear. With "Font" to create a font of "Arial", bold and size of 15. The 15rawstring(String, int x, int y) method of Graphics is able to print information onto the screen.

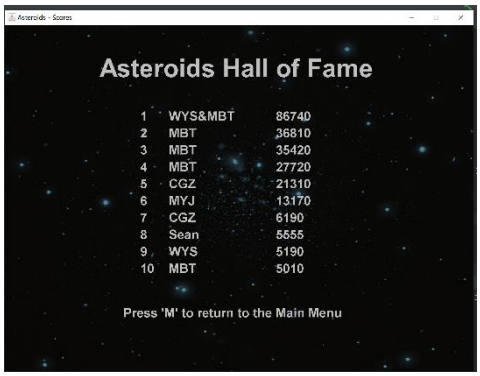
At the end of the game, a result information with the number of shot bullets, aliens and rocks will be shown on a dialogue window, with the hit-shot rate given behind.



When the game finishes, all the variables will be reset with the reset method.

Score Keeper

A part from mark recorder, score keeper is used to load, save and sort different player's total score. It is used to provide the score window a ranked top-10 score list.



The difficulty of the ranking list is to load the name and scores of the player and sort the scores according to the scores.

Firstly, an array that can hold 10 units was created. Then, read the name and scores from "scores.txt" file, borrowing the idea from the given sample code, we used BufferedReader, and try-catch statement, with nextInt and nextString methods to read the scores and name. The Algorithm works like:

try() {
Loop (i<10 && line != 0)
Scanner s <- new Scanner(line);
int score <- nextInt();
String name <- s.nextLine().trim();(trim() can remove blank characters on name sides)
...
} catch() {

```
...
}
```

To save this information, the method used was similar to the above method. The difference is that we use BufferedWriter, getName and getScores methods.

When sorting, we used quick sort method, for a practice of using what we learnt in a real situation.

To compare whether each new record is capable for this top-10 list, getLowestScore() method is used to return the existing lowest score.

Dialogue windows

Dialogue windows are used for requiring information or options from the player. It is surprisingly convenient to call them by using JoptionPane.showOptionDialog() & JoptionPane.showInputDialog() methods, with different returning values.

Hardships & Solutions

Collision Judgement

Collision judgement is one of the most important part of the whole game. With Area class, shapes like polygenes and rectangles can be transferred into area instance. With intersect() method, a common area will be given, making it very easy to distinguish if these two shapes have collision according to whether this common area is empty. In addition, with created

interface Hittable, this judgement of different elements are highly standardized, and area of collision for some elements, especially bullets, are optimized to ensure the stability of judgement.

Movement of Player's Ship

It is quite difficult to make the ship move smoothly, especially in a 360-degree environment. To solve it, all the factors of movement for the space ship are vectorized. The ship's position, angle, velocity and acceleration are calculated separately, and added together again. To simulate real friction, a four-level anti acceleration is added onto the ship, to make the movement smooth and natural.

Random Movement of Alien's ship

Unlike other components, the alien's movement is randomly changed from time to time. To achieve this, the horizontal movement of it is divided into 5 segments by the width of the window. For each cut-off point, a random number is given to change the speed and direction of the movement. In addition, to prevent the alien from staying too long at the boundary, when it approaches the boundary, it's vertical speed increases, making it back into the center of the screen in a shorter time.

Appearance of Asteroids

In the initial stage of building the asteroids, we first found it difficult to make the position and the moving direction randomly. We choose to import "java.util.Random" and then set the random number to the position of asteroids. However, it did not achieve our expectation because the asteroids should appear and move from one side into the game window rather than appear in the screen suddenly – problem 1. Furthermore, asteroids could only move in a straight and paralleled line – problem 2, for we used "x++" and "y++" and both of them are linear. Actually, we expected to make initial asteroids appear on four sides of the window randomly with random speed and angle. Therefore, we asked for random number 'M' from "0" to "3", Representing top, bottom, left and right. Here, problem 1 was solved. Then, our vector class was put into use, which defines the direction of motion – angle, and the speed of the object. Based on vector, things became a little easier. Just take a random double from "0" to "360" to represent the angle and ask another random double to represent the speed. As a result, problem 2 was basically resolved. However, sometimes asteroids' track of movement was too vertical or horizontal. To solve that, we used "do- while syntax" to ask for random number again when it is not perfect, until the angle is in the satisfactory range.

Hyperspace Jump

To make the ship jump randomly, to an almost safe position – not totally because it might jump onto the track of a fast-moving rock or alien's, is really a complicated problem. This actually took us a whole morning to solve. We used the reverse thinking here: Since it is hard

to say ‘no collision’, remove all the point of collision and the safe position would be left. In order to ensure that the ship jump at different angles would not has a collision position, we calculated the maximum side length of the ship and made it the side length of the standard square. Then we divided the screen window with standard square and treat each cut square as a “colliding object”. Crash test each object against aliens, bullets and meteorites, and each available square is listed in a linked list. Number the available positions, one of the numbers is randomly selected. Finally, moved the ship to the specified center point. We used a loop to ensure that each jump ship would not appear at the boundary point and allows players to quickly confirm the ship’s position.

Creative Ideas and Designs

In addition to the title requirements, we added some innovative elements to make the game experience more fun and interesting. For example:

Players can spend their 500 points to get a new chance to play again, and this cost double for every death;

We have drawn a gorgeous and cute flame behind the ship, when players speed up, the flame will appear;

We also add some technical elements: the hit rate, frame number and link list. When the player finished the game, it will automatically count the hit rate of the bullets that the player fired in the whole process;

The FPS controller is created to keep the game loop running smoothly and steadily;

The link list is used to store the bullets of the ship and the six types of space-rock respectively, to make the judgements and controls much easier to achieve;

In order to further enhance the game experience and aesthetic feeling of the game, we have drawn three different background pictures by ourselves.

Conclusion of Authorship

We are very proud that we created so many good ideas to make this game, we also learnt a lot from the sample code and websites:

Totally Original Designs:

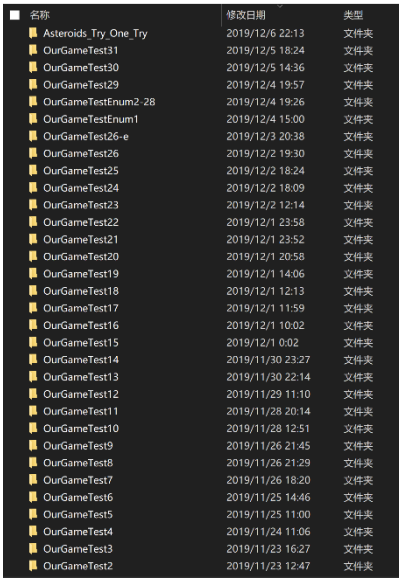
FPS controller, game timer; all the movements and controls of player's ship, alien's ships, asteroids & bullets; mark recorder.

The Idea Partially Learnt from Sample Code:
The design of score keeper & score menu; the idea of making interfaces Hittable and Drawable, the idea of using Jcomponent and Jpanel; the game window with Jframe.

The Idea and Method Learnt from Internet:
Dialogue windows implemented with JoptionPane; Area class for collision judgement.

Feelings of Teamwork

For us, this team's cooperation is not only a new experience, but also a difficult challenge. The reason is because We have never tried to work like a design game with so many technical difficulties. Thus, we have a stronger sense of cooperation than ever before. We meet with 6 friends and have a deep understanding of each other's thoughts in this process. It took us two weeks from designing games to writing code. We had two discussions during the time, wrote code together and exchanged our ideas. We kept working during weekends. We modified more than 30 versions, from the original framework, to the combination of the part from everyone, and made it gradually. This is really an experience that we will never forgot.



名称	修改日期	类型
Asteroids_Try_One_Try	2019/12/6 22:13	文件夹
OurGameTest31	2019/12/5 18:24	文件夹
OurGameTest30	2019/12/5 14:36	文件夹
OurGameTest29	2019/12/4 19:57	文件夹
OurGameTestEnum2-28	2019/12/4 19:26	文件夹
OurGameTestEnum1	2019/12/4 15:00	文件夹
OurGameTest26-e	2019/12/3 20:38	文件夹
OurGameTest26	2019/12/2 19:30	文件夹
OurGameTest25	2019/12/2 18:24	文件夹
OurGameTest24	2019/12/2 18:09	文件夹
OurGameTest23	2019/12/2 12:14	文件夹
OurGameTest22	2019/12/1 23:58	文件夹
OurGameTest21	2019/12/1 23:52	文件夹
OurGameTest20	2019/12/1 20:58	文件夹
OurGameTest19	2019/12/1 14:06	文件夹
OurGameTest18	2019/12/1 12:13	文件夹
OurGameTest17	2019/12/1 11:59	文件夹
OurGameTest16	2019/12/1 10:02	文件夹
OurGameTest15	2019/12/1 0:02	文件夹
OurGameTest14	2019/11/30 23:27	文件夹
OurGameTest13	2019/11/30 22:14	文件夹
OurGameTest12	2019/11/29 11:10	文件夹
OurGameTest11	2019/11/28 20:14	文件夹
OurGameTest10	2019/11/28 12:51	文件夹
OurGameTest9	2019/11/26 21:45	文件夹
OurGameTest8	2019/11/26 21:29	文件夹
OurGameTest7	2019/11/26 18:20	文件夹
OurGameTest6	2019/11/25 14:46	文件夹
OurGameTest5	2019/11/25 11:00	文件夹
OurGameTest4	2019/11/24 11:06	文件夹
OurGameTest3	2019/11/23 16:27	文件夹
OurGameTest2	2019/11/23 12:47	文件夹

screen shots for earlier versions: