PROJECT: ”ASTEROIDS RETRO GAME”

CLASS: OBJECT-ORIENTED PROGRAMMING

STUDENTS: PATRICK BUDEA,

CRISTIAN CHIRA

PROFESSORS: MARIUS JOLDOS

ARON BAKA

TABLE OF CONTENTS

SHORT DESCRIPTION

"Asteroids" was one of the first computer games ever released. Its simple concept, difficulty curve and an infinite potential score-wise, made it a huge hit in arcades and homes across the world.

Through this project we wanted to bring a modern and fresh approach as an homage to this piece of history.

The game allows the user, who is controlling a spaceship, to try and survive a never-ending onslaught of asteroids by exploding them with its laser gun. Each asteroid destroyed provides 100 points, the game becomes more difficult due to the inertia of the ship, just like in space.

When hit by an asteroid, the game ends and the user can log his score in the hall of fame. Afterwards, a choice is given: try again or see the top 5 players that have survived and destroyed the most asteroids.

The game was developed by respecting OOP principles and using the JavaFX Library for the game logic, as well as for the interface. The database component was written with MySQL.

ENTITY MOVEMENT

The Ship

The movement is realized by updating the position of the game objects repeatedly. This is done using the AnimationTimer class from the JavaFX library by overriding the handle method of this class, the movement method being called multiple times per second (≈ 60 times/second). This makes the animation smooth and easy to modify in case more elements should be introduced.

```
new AnimationTimer() {
    @Override
    public void handle(long now) {

        keyHandler.movement(ship);
        keyHandler.shoot(projectiles, ship, pane);

        if (ship.collisionWithAsteroid(asteroids, textScore)) {
            stage.close();
            EndScreen endScreen = new EndScreen(new Stage());
            this.stop();
        }

        Asteroid.laserCollision(projectiles, asteroids, pane, points,
textScore);
        Asteroid.addAsteroids(ship, asteroids, pane);

    }
}.start();
```

The movement of the Ship (player character) is done by using key handlers (instances from KeyHandler.java). A map is used for detecting the pressed keys, which switch boolean values depending on the keypress.

```
public void movement(Ship ship)
{
    if (pressedKeys.getOrDefault(KeyCode.LEFT, false)) {
        ship.turnLeft();
    }
    if (pressedKeys.getOrDefault(KeyCode.RIGHT, false)) {
        ship.turnRight();
    }
    if (pressedKeys.getOrDefault(KeyCode.UP, false)) {
        ship.accelerate();
    }
    ship.move();
}
```

Just like in space there is no braking, so, if the player wants to slow down, he must use the acceleration in the opposite direction. The movement controls of the ship are simple: acceleration (operated by ↑) and orientation (operated by keys → and ←). If the ship reaches a certain speed, this speed will be maintained until the player inserts a new input or collides with an asteroid.

The asteroids

The movement of the asteroids is similar, except that there is no user input, that is when a new asteroid is created it has a pre-determined size, direction, velocity and rotation, all being randomly generated. During the game, new asteroids appear on the screen, making the game more difficult if the player doesn't efficiently use his laser gun.

```
public void move() {
    character.setTranslateX(character.getTranslateX() + movement.getX());
    character.setTranslateY(character.getTranslateY() + movement.getY());

    if(character.getTranslateX()<0){
        this.character.setTranslateX(this.character.getTranslateX() + GameWindow.getWidth());
    }
    if(character.getTranslateX()>GameWindow.getWidth()){
        this.character.setTranslateX(this.character.getTranslateX()%GameWindow.getWidth());
    }
    if(this.character.getTranslateY()<0){
        this.character.setTranslateY(this.character.getTranslateY()+GameWindow.getHeight());
    }
    if(this.character.getTranslateY()>GameWindow.getHeight()){
        this.character.setTranslateY(this.character.getTranslateY()%GameWindow.getHeight());
    }
}
```

COLLISION DETECTION

The collision detection is made by comparing the shapes of the entities and checking if they intersect in the cartesian plane.

```
public boolean collision(Entity character){
    Shape collisionArea=Shape.intersect(this.character, character.getCharacter());
    return collisionArea.getBoundsInLocal().getWidth()!=-1;
}
```

SCREEN WRAPPING

The original “Asteroids” game was revolutionary due to *screen-wrapping*. This feature is present in this rendition of the game as well, by checking if the entity is out of the game window’s bounds, and if yes, making the entity appear on the opposite side of the game window. (see the movement *if-s* above.)

LEADERBOARD

At the end of the game, the player can input his username and score into a database and see how he matches with other players. If the score is among the top 5, he will have a privileged spot on the leaderboard screen. (See EndScreen.java and Leaderboard.java)

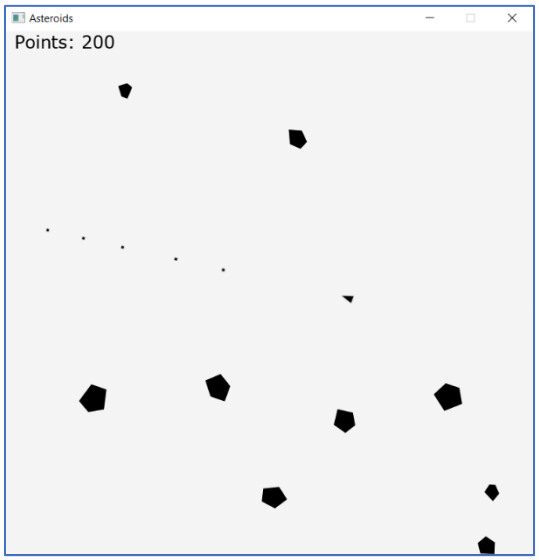
SHIP PROJECTILES

In order to protect themselves from the incoming asteroids, the ship is equipped with a laser gun, with which the pilots can destroy the asteroids. This is done by pressing the *Space* key. In order to reduce the number of entities on the screen there can be a fixed number of lasers on the screen at any time, after a few seconds, these projectiles will disappear. This mechanism also serves as a method to discourage projectile spamming. The projectile collision is similar with the other ones. The projectiles also benefit from *screen-wrapping*.

```
public class Projectile extends Entity {

    private final long startTime;

    public Projectile(int x, int y) {
        super(new Polygon(2,-2,2,2,-2,2,-2,-2),x,y);
        startTime = System.currentTimeMillis();
    }

    public long getStartTime() {
        return startTime;
    }


}
```

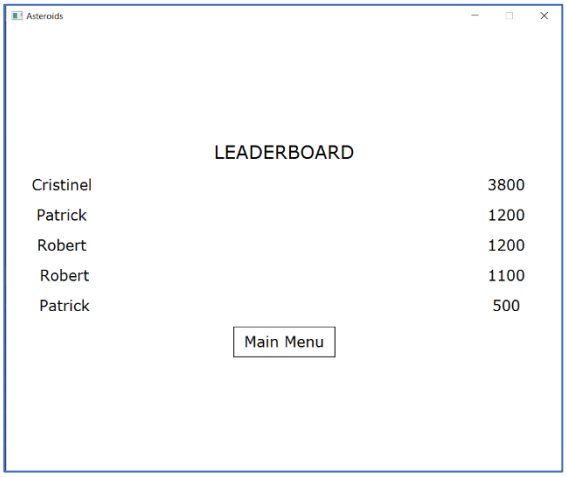
APPLICATION SCREENSHOTS

THE MAIN MENU



THE GAME WINDOW

YOU LOST WINDOW



LEADERBOARD WINDOW

SOLUTION PRESENTATION

The project was built using different design patterns, data structures, event handlers, key listeners, which have been either created by the authors or already implemented by the Libraries used.

Design Patterns

As creational design pattern the *Abstract Factory* design pattern has been used. This is the most suitable one, because every single entity on the screen is essentially a polygon. That is why the *PolygonFactory.java* has been implemented in order to create the objects by using mathematical formulas.

```
public class PolygonFactory {

    public Polygon createPolygon(){
        Random rand=new Random();
        double size=10+rand.nextInt(10);
        Polygon polygon=new Polygon();
        double c1=Math.cos(Math.PI*2/5);
        double c2=Math.cos(Math.PI/5);
        double s1=Math.sin(Math.PI*2/5);
        double s2=Math.sin(Math.PI*4/5);

        polygon.getPoints().addAll(size,0.0,size*c1,-1*size*s1,-1 * size * c2, -1 * size *
s2, -1 * size * c2, size * s2,
                size * c1, size * s1);
        for (int i = 0; i < polygon.getPoints().size(); i++) {
            int change = rand.nextInt(5) - 2;
            polygon.getPoints().set(i, polygon.getPoints().get(i) + change);
        }
        return polygon;
    }
```

As behavioral design patterns the following ones are present: *Iterator* – for accessing elements of collections, *Mediator* – the communication between classes is simple, mostly instantiating the required application window (i.e. Main Menu → Leaderboard) .

Data Structures, Event Handlers and Key Listeners

The simplest data structure that has been used is *ArrayList*. It is used for storing the asteroids, and by using this data structure, the continuous creation of asteroids is trivial. It also plays a big part in storing the players data retrieved from the database for the *Leaderboard*. Another important application of this data structure is for storing the projectiles and controlling their behaviour.

Another data structure used is *Map*, which "maps" the keycode of a pressed key with a Boolean value. This is used for the *key listener* in order to detect the user's inputs.

For the interface, other data structures are used to create and display frontend components.

Event Handlers and Key Listeners are used for detecting either movement changes of the ship or for frontend related actions (button animations, navigation and layout changing).

```
public class GameWindow {

    private static Pane pane;
    private Entity ship;
    private Scene scene;
    private ArrayList<Asteroid> asteroids;
    private ArrayList<Projectile> projectiles;
    public static int width = 600;
    public static int height = 600;
    private TextScore textScore;
```

```
public class KeyHandler {

    private final Map<KeyCode,Boolean> pressedKeys;
    private final AtomicBoolean pressed;

    public KeyHandler() {
        this.pressedKeys = new HashMap<>();
        this.pressed = new AtomicBoolean();
    }

    public void initialize(Scene scene)
    {
        scene.setOnKeyPressed(keyEvent -> {
            pressedKeys.put(keyEvent.getCode(), Boolean.TRUE);
        });
        scene.setOnKeyReleased(keyEvent -> {
            pressedKeys.put(keyEvent.getCode(), Boolean.FALSE);
            pressed.set(false);
        });
    }
```

```
if (ship.collisionWithAsteroid(asteroids, textScore)) {
    stage.close();
    EndScreen endScreen = new EndScreen(new Stage());
    this.stop();
}

Asteroid.laserCollision(projectiles, asteroids, pane, points, textScore);
Asteroid.addAsteroids(ship, asteroids, pane);
```

```
if("EXIT GAME".equals(button.getText())) {
    button.addEventHandler(MouseEvent.MOUSE_CLICKED, new EventHandler<MouseEvent>() {
        @Override
        public void handle(MouseEvent mouseEvent) {
            closeGame(stage);
        }
    });
}
if("START".equals(button.getText())) {
    button.addEventHandler(MouseEvent.MOUSE_CLICKED, new EventHandler<MouseEvent>() {
        @Override
        public void handle(MouseEvent mouseEvent) {
            //GameWindow gameWindow=new GameWindow(new Stage());
            new GameWindow(stage);
        }
    });
}
if("LEADERBOARD".equals((button.getText()))){
    button.addEventHandler(MouseEvent.MOUSE_CLICKED, new EventHandler<MouseEvent>(){
        @Override
        public void handle(MouseEvent mouseEvent) {
            new Leaderboard(stage);
        }
    });
}
```

```
btn.setOnAction(new EventHandler<>() {
     @Override
     public void handle(ActionEvent actionEvent) {
         String name=userTextField.getText();
         if(!name.isEmpty()){
             DatabaseConnection.updateDB(name);
             userTextField.clear();
             userTextField.setPromptText("Score Submitted!");
             btn.setVisible(false);
             GridPane.setRowIndex(lbBtn, 2);
         }
     }
 });
```

CONCLUSION

Essentially, this game is a simple implementation of multiple aspects:

- OOP principles
- Data Structures
- Event Handling
- Design Patterns

The code is intuitive, easy to modify, having a wide range of customizability, while also giving feedback to the player through the leaderboard mechanic.

The components of the *JavaFX* library are widely utilized throughout the project, including both in the backend and the frontend. *Streams* and *Lambdas* are utilized by both the collision detection and the interface.

Data serialization is also present using a *MySQL Database Management System*. The link between the JavaFX project and the database is made using the *JDBC driver* (see *module-info.java*).

The interface is simple, user-friendly, has good readability, also giving feedback to the user by highlighting the hovered components.

FURTHER DEVELOPMENTS

The game can be easily modified and further developed due to the intuitive implementation, the readability of the underlying code.

New features may be added in, such as:

- Multiple difficulties
- Different color themes, ship models, projectile particles
- Sound effects and visual cues
- Asteroids splitting in half down to a certain point when hit with projectiles
- Multiple lives
- An account and login system