Student: Simone CampoSID: 1911840

  ARU – Faculty of Science & Engineering

DISTRIBUTED PROGRAMMING

---

Rational Report

Table of Contents

[1. Racing Game – First Part 2](#_Toc132743341)

[1.1. Definition of classes 2](#_Toc132743342)

[1.2. Rotation technique 2](#_Toc132743343)

[1.3. OUTPUT 2](#_Toc132743344)

[2. Racing Game – Second Part 4](#_Toc132743345)

[2.1. Definitions of classes 5](#_Toc132743346)

[2.2. logic behind the CAR movement 8](#_Toc132743347)

[3. Racing Game – Third Part 9](#_Toc132743348)

[3.1. classes structure 9](#_Toc132743349)

[3.2. communication client-side 9](#_Toc132743350)

[3.3. communication SERVER-side 10](#_Toc132743351)

[3.4. Limitations & Future work 10](#_Toc132743352)

[4. Conclusions 11](#_Toc132743353)

# Racing Game – First Part

The first part of the Racing Game assignment consists in implementing a graphical user interface (GUI) using JFrame and JPanel, where two cars rotate for 360 degrees. The program has been developed to give a fluid animation of the cars’ movement, and these cars will be then controlled by the players in the second part of the assignment.

## Definition of classes

Part 1 of the assignment is composed by three main classes:

* **Main**, which is responsible for initialising and running a new JFrame object.
* **MyFrame**, that is responsible for setting the boundaries and graphical elements of the screen, as well as to initialise the JPanel object.
* **MyPanel**, that is responsible for loading and painting cars.

## Rotation technique

To produce a smooth animation of cars, the MyPanel class uses two “ImageIcon[]” arrays that store 17 different images of the car object, each at a different angle, with the cars being framed every 22.5o. The loadCar()function is used to load the images into the arrays. In the MyPanel constructor, a timer object is initialized with a delay of 1000/30 milliseconds. When the timer expires, an action event is triggered, and the component is repainted.

Every *.png* picture in the array is named according to their degree angle, starting from the initial position of the car, and they are stored in the array in ascendent order from angle 0o to angle 275.5o. An index called “currentImage” is initialised to 0, and each time the component is painted, the index is used to access the next image in the array. Since the cars rotate, the array needs to be iterated from the beginning once the last image in the array is shown. To accomplish this, the position of the image to show in the array is determined by taking the index modulo 16.

## OUTPUT

An example of the output produced by Part 1 of the assignment is shown in Figure 1.

Graphical user interface

Description automatically generated

Figure 1: GUI Racing Game - Part 1

# Racing Game – Second Part

Following the playground developed in part 1, part 2 of this assignment consists of a complete racing game where two players control the car objects in the racetrack and compete with each other’s to cut the finish line. Figure 1 shows the code in execution, whose GUI displays two instances of cars that can be controlled with the keywords painted in the legend below the racetrack. A checkpoint system has been implemented to keep track of the completion of the game for each player. Also, car boundary checks make sure cars are on track, and they crash as soon as they hit both the inner and outer boundaries. If the cars collide, the game ends, and the players are notified from the GUI (Figure 2). If one of the cars cuts the finish line, a congratulations message is displayed in the GUI (Figure 3).

Shape

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

## Definitions of classes

The code for part 2 is designed following an Object-Oriented approach. In this respect, each class has a specific purpose, and it is designed in a way that each component has their own logic. In more details, the software comes with the following classes:

1. **Main**

As per definition, the main function executes the game, and it is in charge for setting up a new “GameFrame” object.

1. **GameFrame**

The GameFrame class extends the “JFrame” class in Java’s Swing library. It creates a window for the racing game using the “GamePanel” class as its content pane. The window has a fixed size specified by the “Configurations” class. The constructor sets the title, size, and default close operation of the window, and adds the “GamePanel” instance to the container. In the end, the window is set to visible.

1. **GamePanel**

The “GamePanel” class, which extends the JPanel and implements KeyListener, is responsible for managing the GUI of the game application and listening to user input through the keyboard to control the game objects. In this respect, the constructor initialises a GameThread object, adds KeyListener to the panel, and sets the focus to the panel so to receive keyboard input and allow the users to control the cars. The paintComponent()method is overridden, and it is responsible for rendering the graphics. Firstly, the method checks if the game is over, then it calls other methods to draw the racetrack, red and yellow cars, and their related components. The paintRaceTrack() method is responsible for drawing the racetrack with the grass, outer and inner edges, mid-lane marker, and start line. The paintCar() method draws the car by getting the object as a parameter.

The paintRedCarElements() and paintYellowCarElements() methods draw the legend and statistics for the red and yellow cars respectively. The paintEndGame() method is called when the game is over, and it displays the appropriate message to the user.

The keyPressed() method is also overridden, and it is called when a key is pressed. It checks which key is pressed and performs the corresponding action. For example, if 'w' is pressed, it turns the red car left, if 's' is pressed, it turns the red car right, if 'q' is pressed, it increases the speed of the red car, and so on.

1. **Car**

The “Car” class is responsible for modelling the car object so it can move, turn, and collide with the opponent car. The class has several properties such as ImageIcon array, carShape, direction, speed, prevX, prevY, checkPoints, crashSound, and engineSound, which are used to keep track of the state of the car.

The Car constructor takes four arguments, including the root directory, carName, x, and y coordinates. The constructor initializes the instance variables, loads the sound files for car engines and car crash, and creates the ImageIcon array that stores the images of the car using the approach from Part 1.

The class has several methods that allow the car to move, turn, and collide with the opponent car. The methods like turnRight(), turnLeft(), adjustShape(), moveForward(), causeCrash(), increaseSpeed(), and decreaseSpeed() manipulate the state of the car according to the key pressed by the user.

1. **RaceTrack**

The “RaceTrack” class defines a race track object with attributes such as grass, outer and inner edge, mid-lane marker, and checkpoints. The grass, outer, and inner edge are defined using the Rectangle class. The checkpoints are defined using an array of Rectangles.

The RaceTrack class comes with methods for accessing the attributes of the race track, including getGrass(), getOuterEdge(), getInnerEdge(), and getMidLaneMarker(). Additionally, the class provides two methods for checking if a car is on the track or has crossed a checkpoint, namely carInTrack() and checkPointCrossed().

The carInTrack() method checks if the car is on track by using the intersects() method of the Rectangle class to check if the car's shape intersects with the grass area. The method returns true if the car is not on the grass and its shape is contained within the outer edge of the race track.

The checkPointCrossed() method checks if the car has crossed a checkpoint or not. It uses the contains() method of the Rectangle class to check if the car's current position is within the boundary of a specific checkpoint. If the car has crossed a checkpoint, the method returns true.

1. **Game**

The "Game" class contains the core instructions to models the two-player racing game. The class contains a racetrack, two cars (red and yellow), and sound effects for game win and loss events.

The constructor initializes both the cars and the racetrack, and loads the sound files for game win and loss events. The playGameWonSound() and playGameFailedSound() methods play the corresponding sound effects.

The updateGame() method updates the game state by checking if the red and yellow cars have collided. If a collision occurs, the game ends, the cars shut down, and the playGameFailedSound() method is executed. Otherwise, the updateCar() method is run for each car.

The updateCar() method updates the state of a car by checking if it is on the track or has crossed a checkpoint. If a car has crossed all the checkpoints and is the first to do so, the game ends, the cars shut down, and the playGameWonSound() method is called. If either cars have collided or hit the boundaries of the track, the causeCrash() method is called. The other methods in the class are used to control the movement and speed of the red and yellow cars by turning them left or right and increasing or decreasing their speeds.

1. **GameThread**

The "GameThread" class, which extends the Thread class, is responsible for running the game in a separate thread to avoid blocking the main thread. It updates the game state and repaints the game panel in each iteration of the loop and then sleeps for a short period of time before starting the next iteration.

In more details, the class has two instance variables: "game", which represents an instance of the "Game" class, and "gp", which represents an instance of the "GamePanel" class. Both instances are initialised within the constructor.

The run() method overrides the run method in the Thread class, and contains a while loop that runs as long as the game is not over. In each iteration of the loop, the updateGame() method of the "game" object is executed, followed by the repaint() method of the "gp" object. Finally, the thread is put to sleep for a period of time defined by the "Configurations.TIMER\_DELAY" constant to slow down the game loop and provide a smoother gameplay experience.

1. **Configurations**  
   The Configurations class contains static constants that store values for different configurations used in the game. It includes constants like RED\_CAR\_COMMANDS and YELLOW\_CAR\_COMMANDS that store the commands for the two cars in the game. It also includes constants for the timer delay, frame width, and frame height, as well as racetrack constants such as the outer edge and inner edge sizes and checkpoint locations. These values are used throughout the game and provide an easy way to change different configurations without having to search through the code.

## logic behind the CAR movement

The turnRight() and turnLeft() methods in the Car class are responsible for changing the direction of the car by modifying the direction property. In fact, this variable represents the current direction the car is facing, measured in increments of 22.5 degrees (since there are 16 possible directions in a full circle). For example, if direction is currently 0, that means the car is facing directly to the right, and if direction is 4, the car is facing diagonally down and to the left. In turnRight(), the direction variable is incremented by 1 (with modulo 16 applied to wrap around to 0 when it reaches 16). This corresponds to the car turning to its right (clockwise) by 22.5 degrees. Likewise, in turnLeft(), the direction variable is decremented by 1 (with modulo 16 applied to wrap around to 15 when it reaches -1). This corresponds to the car turning to its left (anti clockwise) by 22.5 degrees.

The moveForward() method is responsible for moving the car forward in the direction it is currently facing. The car's position is updated based on its current speed and direction. The angle variable is calculated by converting the direction (in increments of 22.5 degrees) to radians. Then, the x and y positions are updated by adding the result of speed \* Math.cos(angle) to the current x position and speed \* Math.sin(angle) to the current y position, respectively.

So, for example, if the car is facing directly to the right (direction = 0) and has a speed of 1, then angle would be 0 radians, and calling moveForward() would move the car 1 unit to the right (i.e., increase its x position by 1). If the car is facing diagonally down and to the left (direction = 4) and has a speed of 2, then angle would be (i.e., ), and calling moveForward() would move the car 2 units down and to the left (i.e., decrease its x position by 2 \* Math.cos(angle) and decrease its y position by 2 \* Math.sin(angle)).

# Racing Game – Third Part

The third part of the assignment consists of creating a client-server application where two instances of the client can connect to the server using a the TCP handshake protocol, and control the cars. The game rules and structure are inherited from Part 2, with the only difference that now client control their own Car from their own instance of the game.

## classes structure

For the purpose of this exercise, we have three more classes:

* **ClientData**, which is responsible for storing information about the player and opponent’s cars.
* **ServerData**, which is responsible for storing information about the player one and player two’s cars.
* **ClientThread**, that allows the Client instance to connect to the server using the local port and set up the communication via TCP.
* **TCPServer**, which initialises a server that can accept multiple clients and create a thread for each of the client connected.
* **TCPClientHandler**, that allows the Server to accept connection from the Client and keep the communication active with those.

## communication client-side

The **ClientThread** class, which extends the Thread class, is designed in a way that it runs concurrently with other threads. Just like the other classes, the code is divided into methods that perform specific tasks.

The run()method is the main method of the thread, which creates a socket and initialises the input and output streams. Eventually, it sends a message to the server to establish a connection. Once the connection is established, the client waits for a response from the server. If the response is not null, it prints the response to the console and calls the handleServerResponse()method to process the response.

If the startGame() method returns true, that happens when two clients are successfully connected to the server, the client starts the game by calling startGame() method of the **GameFrame** class. Once the game is started, the client sends updates about the player’s car to the server every second by calling the sendCarUpdate() method.

The sendMessage() and receiveMessage() methods are used to send and receive messages to and from the server. On the other hand, the sendCar() method sends the client's car object to the server while the receiveOpponentCar() method receives the opponent's car object from the server and stores it in the **ClientData** class.

Finally, the handleServerResponse() method processes the server's responses by switching on the response string. If the response is "red" or "yellow", the client creates a new instance of the **GameFrame** class and sends its car object to the server.

## communication SERVER-side

As mentioned in Section 3.1., the **TCPServer** class is responsible for listening for incoming client connections, accepting them, and creating a separate thread for each connected client. The main method initialises a ServerSocket object and starts an infinite loop to accept clients. Once a client connects, a TCPClientHandler object is created and passed the connected Socket and player number. The TCPClientHandler objects is then started on a separate thread. The **TCPServer** class’s job is to monitor the active clients and keep track of their number.

The **TCPClientHandler** class implements the Runnable interface, which allows the code to be run on a separate thread. The run method handles incoming and outgoing messages to/from the client. When a client connects, it initializes BufferedReader and DataOutputStream objects to read and write messages from/to the client. Then, it initializes ObjectInputStream and ObjectOutputStream objects to read and write Car objects.

The handleClientResponse() method is responsible for processing incoming messages from the client and undertaking specific actions. Similar to **ClientThread** class, The sendMessage() and receiveMessage() methods are used to send and receive messages to and from the client. Also, these methods are synchronized to ensure that multiple threads do not try to write/read from the output/input stream at the same time. The sendOpponentCar() and receiveCar() methods are synchronized for the same reason as sending and receiving messages, and are used to send and receive Car objects between clients. Finally, isAlive() method is used to determine if the client connection is still active. The alive variable is set to false when the run method terminates.

## Limitations & Future work

Lots of time has been spent to implement a working solution that allow the clients to play using two separate game interfaces. Some milestones have been achieved, and clients are able to connect to the server and establish a successful TCP-Handshake communication. In this respect, when the first client connects, it comes across a lobby screen, which is replaced by the game race once the second client is connected. Also, each player can control their own cars in their own instances, which means that clients are able to exchange messages and objects with the server. The limitation arises when it comes to update the server with the current status of the player’s car. In this respect, the client is able to send the car object through while the server gets stuck when deserialising the object. Investigation has been conducted to identify the root cause of the problem, and it has been found that the objects can be serialised and deserialised (i.e., exchanged) when the clients connect to the server whereas they get stuck when the exchange becomes repetitive during the gaming phase. Also, it has been noted that objects are successfully exchanged in debugging. It is possible to assume that the limitation does not lie in the code itself, but rather in the way threads are executed. Whilst synchronize keywords has been introduced to prevent a object/message writing from being interrupted, future work should be done to improve the management of threads, and allow the exchange of objects without causing congestions.

# Conclusions

The assignment has been written with the aim to fulfil the learning objectives and outcomes of the Distributed Programming module. In this respect, the practical assignment demonstrates the ability of the author to analyse where it is necessary to deploy distributed systems and apply theory and principles underpinning distributing computing in the context of internet communications. Whilst the part 3 of the code has a few limitations, the author has conducted extensive testing and debugging to understand the root cause behind the problems and provide valuable suggestions.