Technical Design of “The Frog” Project

Team DHI1V.So-3

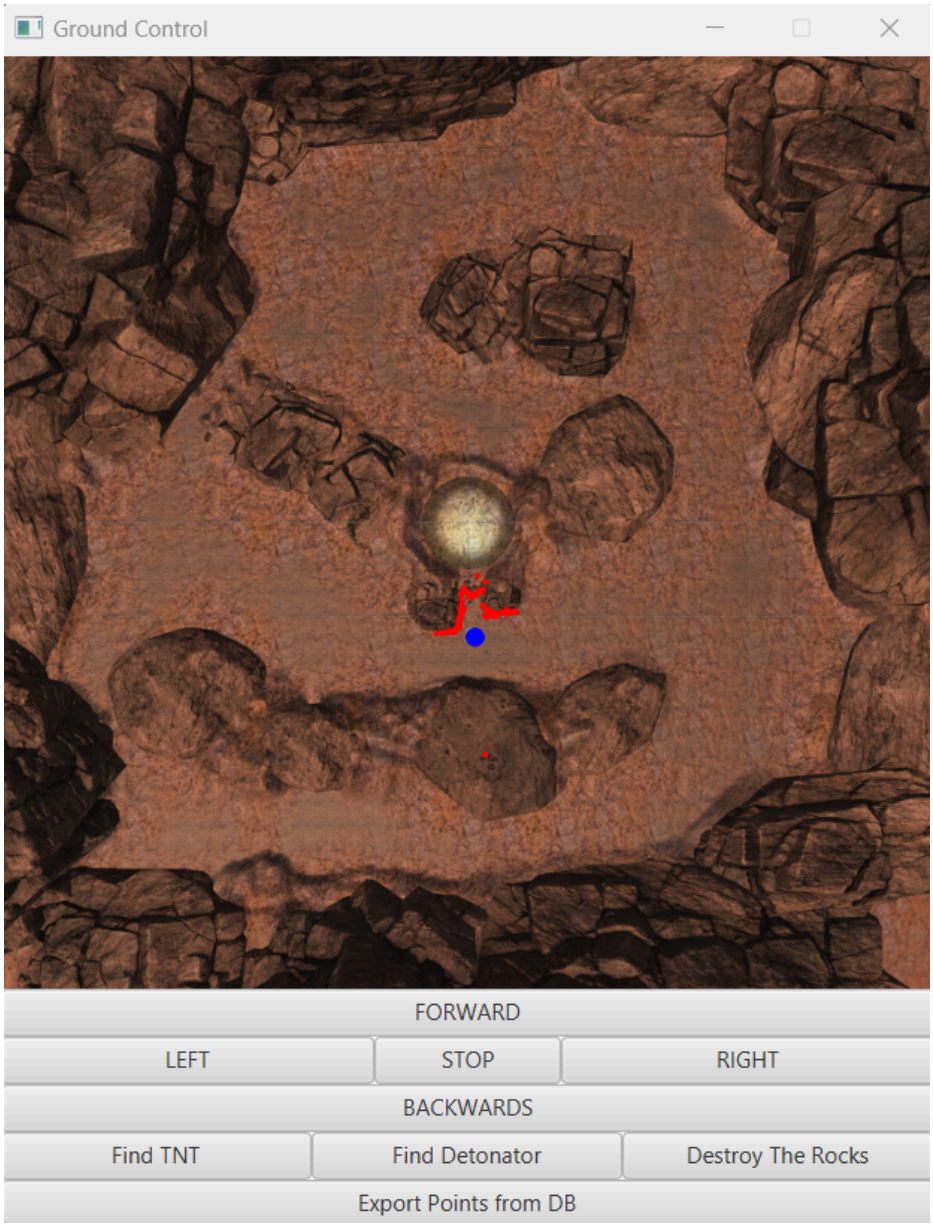
Anastasiia Khylyk
Danylo Kurbatov
Radin Soleymani
Maksim Sadkov
Tanya Ivanova

1. Architecture

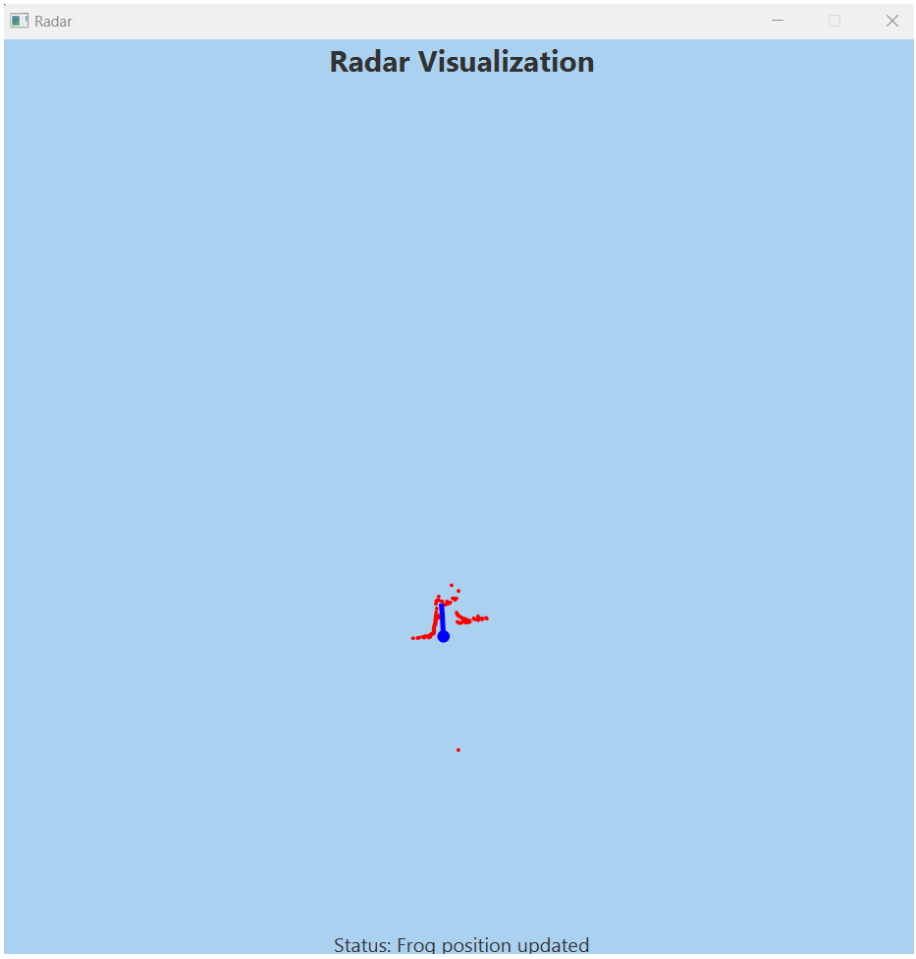
Classes and methods

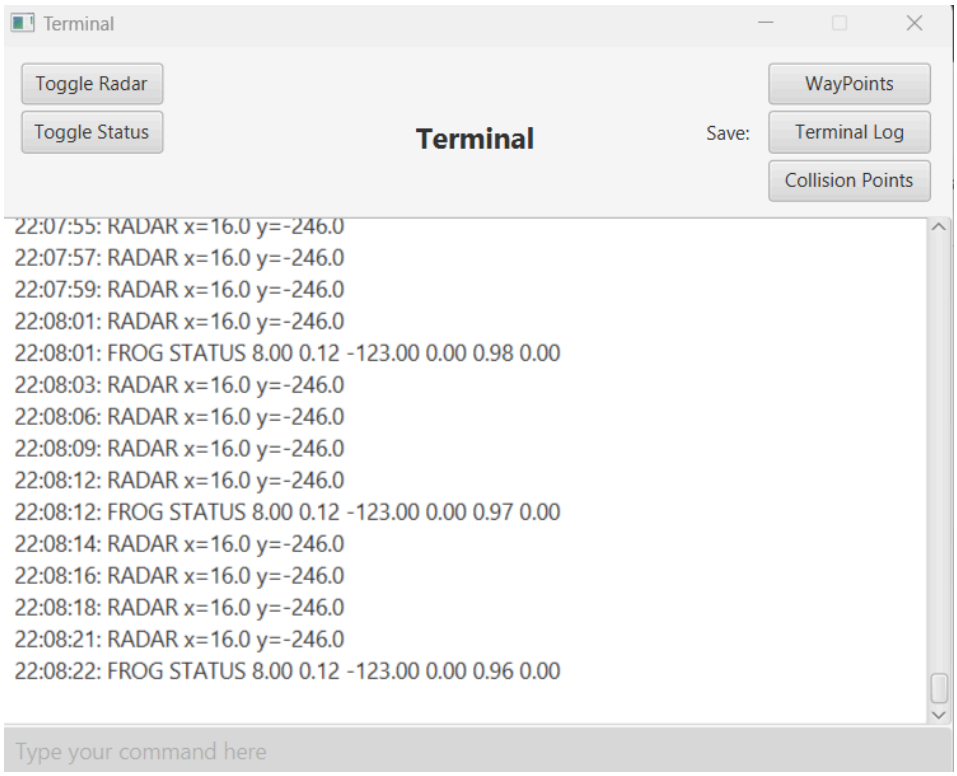
Modules

There are two modules, in which the classes of the project are stored: GroundControlFX and PilotFX. The GroundControlFX module contains everything related to the features of ground control map (see image below).



The PilotFX module contains all classes related to the Pilot features, which are showing radar points on a separate map and terminal, as well as navigation algorithms.


Radar
Radar Visualization
Status: Frog position updated



Ground Control Module

The ground control module contains a folder “Controller”, which has a class named MapController. Outside of this folder, there is a Main class.The MapController class has all of the methods, necessary for displaying the map, showing the obstacles and the position of the frog with its trajectory and adding functionality to the movement buttons under the map.

Methods in MapController class:

- void intialize()

  This method connects to the Pilot class in the PilotFX module. It ensures that the radar points, detected by pilot, can be accessed via the MapController class, as well as the frog position. This approach was chosen as it synchronizes processes within the two applications.It ensures that the same radar points are displayed simultaneously both on the Ground Control map and on the Pilot’s radar point map.
- void handleMouseClick(MouseEvent event)

  This method provides the functionality of clicking on the Ground Control map to choose a coordinate, and then sending the drive command of the frog there. It gets the coordinates of the mouse click and transfers them to the system of coordinates, used by frog. This is a necessary step, because the Frog application has a system coordinate, where (0,0) is in the center of the map, but for the JavaFX canvas, (0,0) is in the upper left corner.

  The method then calls a handleDraw(double x, double z) method, which draws a circle in a point where the mouse was clicked.

The method handles the multithreading for the buttons under the map and the driving the Frog to a coordinate as well. This method ensures that if the Frog is already driving to a chosen coordinate, or driving in a direction of a button (forward, right, left or backward), the user can click the map again to choose a different destination. It does so by interrupting the current thread and starting a new one.

- void handleDraw(double x, double z)

  Thos method draws a bright green filled circle in the point, where the mouse is clicked.
- void buttonClicked(ActionEvent event)

  This method includes an if statement, which calls a respective method when one of the 5 Ground Control movement buttons is clicked. It also ensures that when the Frog is already driving in a certain direction or to a coordinate, when any of the button is pressed the previous thread is interrupted and the new thread is started.
- void driveForward(), void driveLeft(), void driveRight(), void driveBackwards()

  These four methods call a drive() method from the same class with the parameters depending on the direction. For forward the power is 1, the angle is 0; for left the power is 1, the angle is -30; for right the power is 1, the angle is 30; and for the backwards, the power is -1 and the angle is 0; The number of seconds for each drive command is 0.5 seconds.
- void stop()

  The stop method does not do anything if there is no current process running. If a process is running, it stops it while interrupting its thread.
- void drive(double power, int angle, double duration)

  This method sends a drive command to the frog through the Pilot in a new thread. It also fetches messages from the Pilot to ensure frog position and radar points are updated.
- void updateFrogPosition(double x, double z, double frogAngle)

  This method draws the Frog's position and the trail of the path it has covered on the Ground Control map. It scales the coordinates received as the parameters to translate them from the Frog's system of coordinates to the JavaFx one. This method also checks that the scaled coordinated fit on the canvas and draws them only in this case to avoid errors.
- void onRadarUpdate(double x, double z)

  This method draws obstacles (sets of radar points) on the Ground Control map. Similarly to the previous method, it scales them according to JavaFX's system of coordinates and draws them only in the case if they fit on the canvas.
- void clearCanvas(GraphicsContext gc)

  This method clears the canvas, on which the frog's position, trail and obstacles are drawn. This method is necessary to successfully update the Frog's position.
- void onButtonFindTNT()

  This method drives the frog to the coordinate where TNT is located. It is triggered by the corresponding button in the user interface and uses the driveToCoordinate method with predefined coordinates (-223, -345) for the TNT location.
- void onButtonFindDetonator()

This method drives the frog to the coordinate where the Detonator is located. It is triggered by the corresponding button in the user interface and uses the driveToCoordinate method with predefined coordinates (-147, 241) for the Detonator location.

- void onButtonDestroyTheRocks()
  This method drives the frog back to the base entrance, which is blocked with rocks. It is triggered by the corresponding button in the user interface and uses the driveToCoordinate method with predefined coordinates (1, -100) for the base entrance.
- Void exportRadarPoints()
  This method handles export of radar points from the database for each received point draws a point on the map.

Methods in Main class:

- void start(Stage stage)
  This method runs three FXML files at the same time. When this class is run, it opens the window for the Ground Control Map, for the Pilot Radar Map and the Terminal. The .fxml scenes for these applications are stored in the same module in the “resources” folder, along with the map image that is displayed on the ground control.

Methods in TerminalLauncher class:

- void start(Stage stage)
  This method initializes the Pilot instance and loads the terminal view user interface. It sets up the stage with the terminal view FXML file and displays the window.
- public static void main(String[] args)
  This method launches the JavaFX application. It is the entry point of the application.

Methods in TerminalViewController Class:

- void initialize()
  This method sets up listeners for status and radar updates from the Pilot class. It also configures the terminal text area to automatically scroll to the bottom when new text is added. This ensures the latest messages are always visible.
- void onEnterPressed(KeyEvent event)
  This method handles the Enter key press event in the input field. It validates and executes the command entered by the user. If the frog is currently running, it raises an alert. Valid commands are added to the terminal log and executed by the Pilot.
- void saveCollisionPoints()
  This method saves the current collision points to the database and logs the action in the terminal. It ensures that all detected collision points are persisted.
- void saveWayPoints()
  This method saves the current waypoints to the database and logs the action in the terminal. It ensures that all waypoints are persisted.

- void saveTerminalLog()
  This method saves the terminal log to the database and logs the action in the terminal. It ensures that all command logs are persisted.
- void onStatusReceived(String status)
  This method handles status messages received from the Pilot. It logs the status message in the terminal if status messages are enabled. Duplicate status messages within a short period are ignored to reduce clutter.
- void onRadarReceived(double x, double y)
  This method handles radar messages received from the Pilot. It logs the radar data in the terminal if radar messages are enabled.
- void toggleStatus()
  This method toggles the display of status messages in the terminal. It updates the text of the toggle button to reflect the current state (show/hide status messages).
- void toggleRadar()
  This method toggles the display of radar messages in the terminal. It updates the text of the toggle button to reflect the current state (show/hide radar messages).
- private void raiseAlertWindow(String message)
  This method raises an alert window with the given message. It is used to notify the user of errors or important information.
- private void validateLine(String line)
  This method validates the format of a command line. It throws an IllegalArgumentException if the command line is invalid.
- private void validateStatusCommand(String[] lineParts)
  This method validates the format of a STATUS command. It throws an IllegalArgumentException if the command is invalid.
- private void validateRadarCommand(String[] lineParts)
  This method validates the format of a RADAR command. It throws an IllegalArgumentException if the command is invalid.
- private void validateDriveCommand(String[] lineParts)
  This method validates the format of a DRIVE command. It throws an IllegalArgumentException if the command is invalid.

Pilot Module

The PilotFX module contains three folders: “Controller”, “Model” and “utils”, as well as two classes not belonging to any folder. In the “Controller” module, there are classes that interact with FXML files for the Pilot. That is, RadarPanelController has all of the methods needed to manage the radar point FXML view, which means displaying radar points and displaying Frog’s position. The TerminalViewController class handles interactions of the user with the terminal window, such as pressing buttons and sending commands in the text area.

The “Model” folder includes classes that represent actual objects in Pilot-Frog-Ground Control scope. The “Pilot” represents the Pilot object, and UpdateListener is an interface used by the Pilot class, therefore it is included in the same folder. Pilot has the methods responsible for the navigation of the Frog and receiving information about its status and radar points.

The “utils” folder includes classes that provide the data storage functionality. There is a CSVLogger class that handles writing into CSV files, there is a Database class that stores data in a database and there is a Logger class that is used by both of the other classes.

Finally, there is one class outside of folders, which is“RadarPoint”. It is a runnable class that starts the FXML scene with radar point map.

Methods in RadarPanelController class:

- void initialize()

  This method connects to the Pilot class, adding listeners for radar updates and frog position updates. It ensures that radar data and frog positions are updated on the radar panel. The initial status label is also set to indicate that the radar is waiting for data.
- void updateFrogPosition(double x, double z, double frogAngle)

  This method draws the frog's position on the radar canvas and an arrow indicating its direction. It translates the coordinates from the frog's coordinate system to the JavaFX coordinate system. It ensures the frog's position is within the visible area of the canvas before drawing it.
- void onRadarUpdate(double x, double z)

  This method handles radar updates by drawing radar points on the radar canvas. It translates the coordinates from the frog's coordinate system to the JavaFX coordinate system. It ensures the radar points are within the visible area of the canvas before drawing them. Duplicate radar points are logged but not drawn again.
- void clearRadar(GraphicsContext gc)

  This method clears the radar canvas. It is used to refresh the radar display and avoid clutter from old data points.
- void clearFrog(GraphicsContext gc)

  This method clears the frog canvas. It is used to refresh the frog's position display and avoid clutter from old data points.

Methods in Pilot class:

- static Pilot getInstance()

  This method returns the singleton instance of the Pilot class. It ensures that there is only one instance of the Pilot class throughout the application.
- boolean isRunning()

  This method checks if the frog is currently running. It returns true if the frog is running, false otherwise.
- void setRunning(boolean running)

  This method sets the running state of the frog. It is used to start or stop the frog's movement.
- void addStatusListener(StatusListener listener)

  This method adds a status listener. The listener will be notified of status updates from the frog.
- void addRadarListener(RadarListener listener)

This method adds a radar listener. The listener will be notified of radar updates from the frog.

- void addFrogLocation(frogLocation locator)
  This method adds a frog location listener. The listener will be notified of frog position updates.
- void driveFrogToCoordinate(double x, double z)
  This method drives the frog to the specified coordinates. It calculates the required angle and sends drive commands to the frog. It also handles obstacle avoidance if the frog gets stuck.
- void sendDriveCommand(double motorPower, double steeringAngle, double duration)
  This method sends a drive command to the frog. It constructs the command string and sends it to the SaSaCommunicator.
- void executeCommand(String command)
  This method executes a command by sending it to the SaSaCommunicator. It is used to send custom commands to the frog.
- void sleep(long ms)
  This method pauses the current thread for the specified number of milliseconds. It is used to introduce delays between commands.
- void fetchMessages()
  This method fetches messages from the SaSaCommunicator and processes them. It ensures that status and radar updates are received and handled.
- private void avoidObstacle()
  This method attempts to avoid an obstacle by reversing and steering. It is used when the frog detects that it is stuck.
- private void processMessage(String message)
  This method processes a message from the SaSaCommunicator. It handles status and radar messages, updates the frog's position, and notifies listeners. It also handles checking if there is an obstacle in front of the Frog.

Methods in UpdateListener interface:

- void onUpdateReceived(String message)
  This method is called when an update message is received. Implementing classes should define the behavior for handling update messages.

Methods in RadarPoint class:

- void start(Stage stage)
  This method loads and displays the radar panel user interface. It initializes the stage with the radar panel FXML file and sets the window title and dimensions.

Methods in CSVLogger class:

- CSVLogger(String filePath)
  This constructor initializes the CSVLogger with the specified file path. It also clears the existing CSV file at the specified path.

- void insertXZPoints(double x, double z)
  This method inserts XZ coordinates into the CSV file. It writes the coordinates to the file with the appropriate headers.
- void insertCommandLog(LocalTime time, String command)
  This method inserts a command log with a timestamp into the CSV file. It writes the log entry to the file with the appropriate headers.
- private void clearCSV()
  This method clears the CSV file by creating a new empty file at the specified path. It is called during the initialization of the CSVLogger.
- private boolean checkForHeader(String[] header)
  This method checks if the CSV file contains the specified header. It reads the file and compares each line to the header.
- private void addToCSV(String[] header, String[] data)
  This method adds data to the CSV file, ensuring the header is written if it does not already exist. It appends the data to the file.

Methods in the Database class:

- Database(String url)
  This constructor initializes the Database with the specified database URL. It establishes a connection to the database.
- void addWayPoints(double x, double z)
  This method adds waypoints to the database. It inserts the XZ coordinates into the wayPoints table.
- void addCollisionPoints(double x, double z)
  This method adds collision points to the database. It inserts the XZ coordinates into the collisionPoints table.
- void addCommandLog(LocalTime time, String command)
  This method adds a command log to the database. It inserts the timestamp and command into the commandLog table.
- private void insertPoints(String sql, double x, double z)
  This method inserts points into the database using the specified SQL statement. It executes the prepared statement with the given coordinates.
- private void insertCommand(String sql, String time, String command)
  This method inserts a command log into the database using the specified SQL statement. It executes the prepared statement with the given time and command.
- List<String> exportRadarPoints()
  Exports saved radar points from collisionPoints table in Ground Control database.

Methods in the Logger class:

- static void addCollisionPoints(double x, double z)
  This method adds collision points to the log. It ensures the points are logged only if the system is not currently uploading data.
- static void addWayPoints(double x, double z)

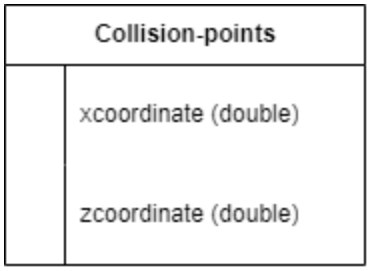
This method adds waypoints to the log. It ensures the points are logged only if the system is not currently uploading data.

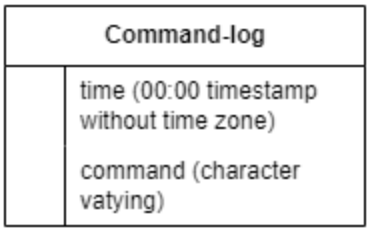
- static void addCommandLog(LocalTime time, String command)
  This method adds a command log entry to the log. It ensures the log entry is recorded only if the system is not currently uploading data.
- static void uploadWayPointsToDatabase()
  This method uploads waypoints to the database and the corresponding CSV file. It sets the uploading flag to true to prevent concurrent modifications.
- static void uploadCollisionPointsToDatabase()
  This method uploads collision points to the database and the corresponding CSV file. It sets the uploading flag to true to prevent concurrent modifications.
- static void uploadCommandLogToDatabase()
  This method uploads command logs to the database and the corresponding CSV file. It sets the uploading flag to true to prevent concurrent modifications.
- List<String> exportRadarPointsFromDB()
  Returns radar points as a list of Strings.

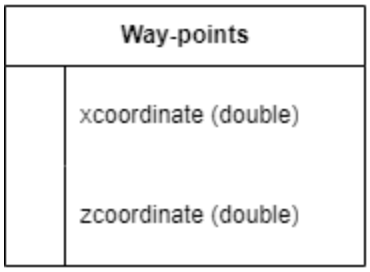
Database

The project includes two databases: one for the Ground Control module, and the other for Pilot module. There are three classes, responsible for saving the data: Database, CSVLogger and Logger. In contrast to the first two classes, the Logger class is static and can be called from any place in the project. The logger class saves two types of data - coordinates of obstacles and terminal commands - to the lists. After this, the user can press one of the “save” buttons in the Terminal, and these lists are uploaded both to the database and csv files. The Database class connects to the database and has methods with pre-made SQL statements to insert data into the tables. CSVLogger class is analogical in a sense that it writes to a CSV file and has a set of methods, that handle writing different types of data.

Below is the scheme (class diagram) of the database that shows the tables and fields in them. All of the tables are independent, which means they do not relate to the other tables. There are data types of the fields indicated in parentheses.





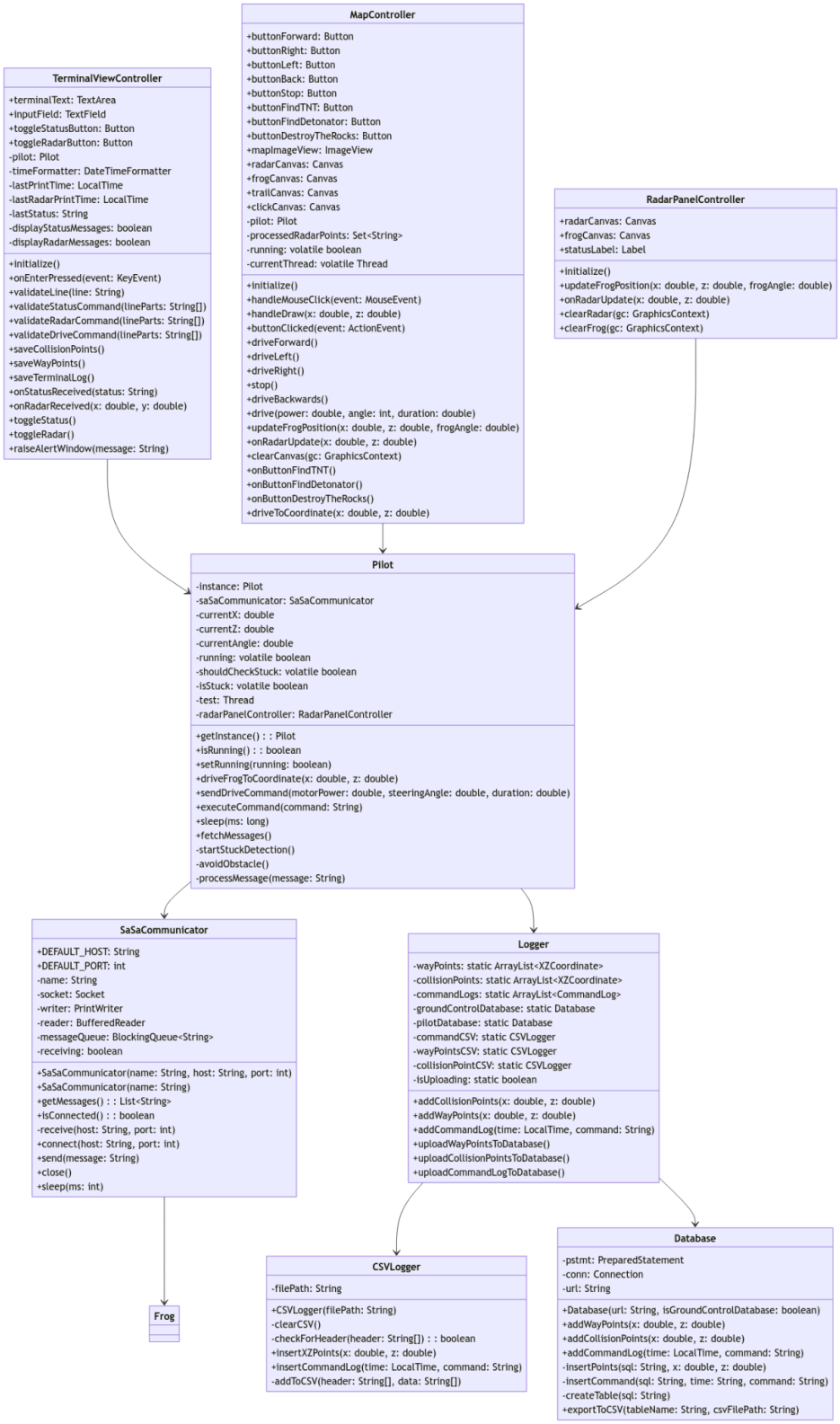


Architectural decisions

1) Explain why exporting and importing is in the pilot, even though under ground control requirements there is

RGS-02	A mission log can be exported to and imported from a CSV-file	F	MUST
RGS-03	A mission log can be saved in the database	F	MUST
RGS-04	A map of collision-points can be exported to and imported from a CSV-file	F	SHOULD
RGS-05	A map of collision-points can be saved to and loaded from the database	F	SHOULD

Class diagram: (svg file is provided)



SaSaCommunicator

- Connected to Frog
  SaSaCommunicator is responsible for handling all the communication between the system and the Frog. It sends commands to the Frog and receives updates from it, acting as an intermediary.
- Used by Pilot:
  The Pilot class uses an instance of SaSaCommunicator to send commands to the Frog and to receive status updates. Pilot delegates the task of managing the network communication to SaSaCommunicator.

Pilot

- Manages Communication with SaSaCommunicator:
  Pilot uses SaSaCommunicator to interact with the Frog. It sends commands through SaSaCommunicator and processes the messages received from the Frog via SaSaCommunicator.
- Interacts with Logger:
  Pilot logs significant events such as movement commands, status updates, and radar data using the Logger class. This helps in maintaining a record of operations for debugging and analysis purposes.
- Controls RadarPanelController:
  Pilot interacts with RadarPanelController to update the radar display. It sends radar and Frog position updates to the RadarPanelController for visualization.

Logger

- Connected to Pilot:
  Pilot calls various methods in Logger to log waypoints, collision points, and command logs. Logger handles the details of logging this information to both CSV files and databases.
- Uses CSVLogger and Database:
  Logger relies on CSVLogger to write logs to CSV files and Database to insert logs into the database. This allows for persistent storage of logs, making it easier to analyze and review the system's performance.

CSVLogger

- Interacts with Logger:
  Logger uses CSVLogger to record logs in CSV format. This involves writing waypoints, collision points, and command logs to CSV files for later review.
- Stores Logs:
  CSVLogger is responsible for writing log data to CSV files. It checks for existing headers, clears files when needed, and appends new log entries.

Database

- Interacts with Logger:
  Logger uses Database to store logs in a structured format. This includes inserting waypoints, collision points, and command logs into the database.
- Stores Logs:
  Database manages the creation of tables and insertion of data, ensuring logs are stored in a reliable and queryable manner.

TerminalViewController

- Connected to Pilot:
  TerminalViewController uses Pilot to execute user commands entered in the terminal. It sends commands like driving directions, status requests, and radar toggles to Pilot.
- Displays Status and Radar Messages:
  TerminalViewController receives status and radar updates from Pilot and displays them in the terminal interface. It also toggles the display of these messages based on user interaction.

MapController

- Connected to Pilot:
  MapController interacts with Pilot to drive the Frog based on user inputs from the map interface. It sends driving commands and coordinates to Pilot.
- Updates Frog Position and Radar Data:
  MapController receives updates about the Frog's position and radar data from Pilot. It then updates the visual representation on the map accordingly.

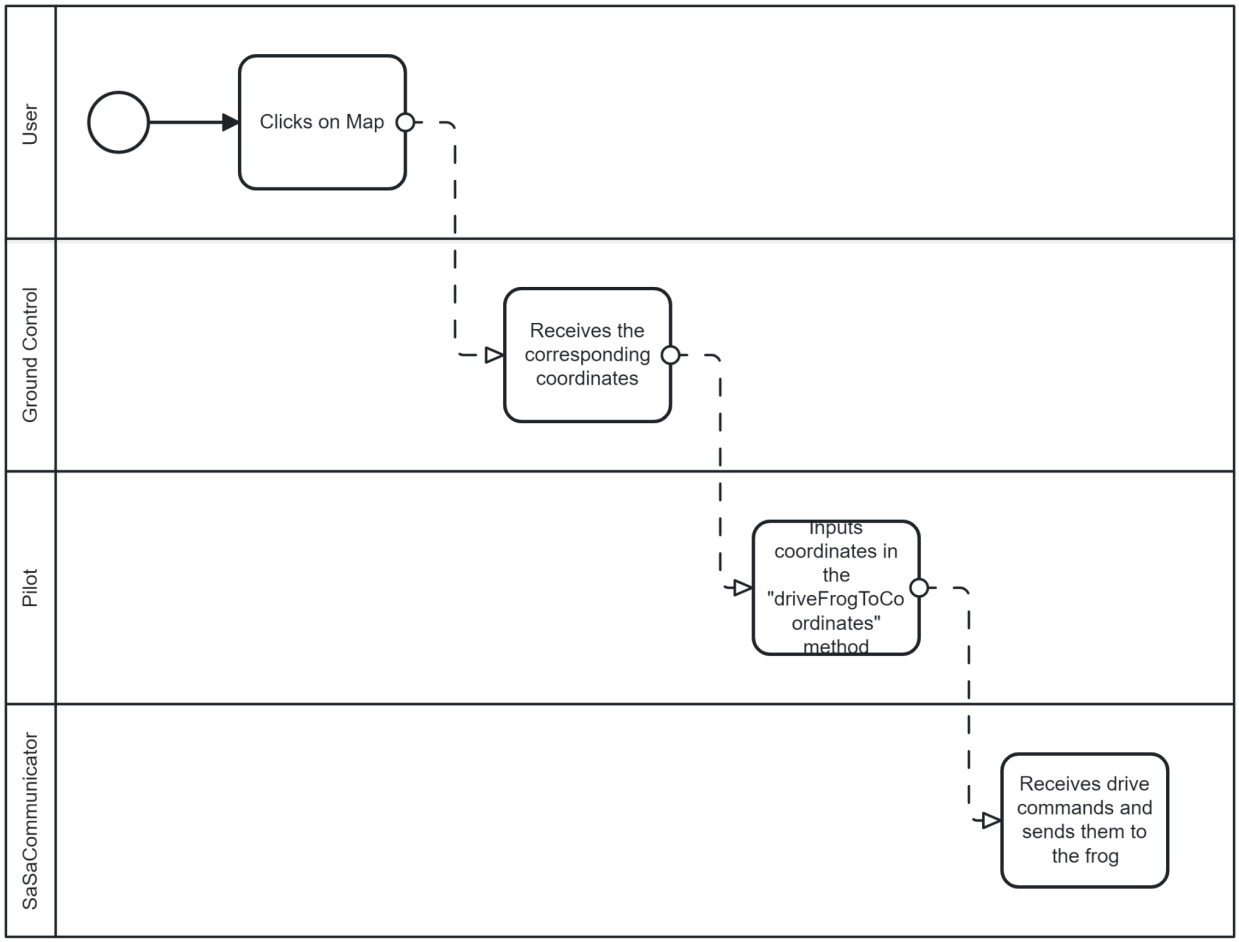
RadarPanelController

- Connected to Pilot:
  RadarPanelController receives updates about the Frog's position and radar data from Pilot. It uses this information to update the radar display in the user interface.

Interactions Overview

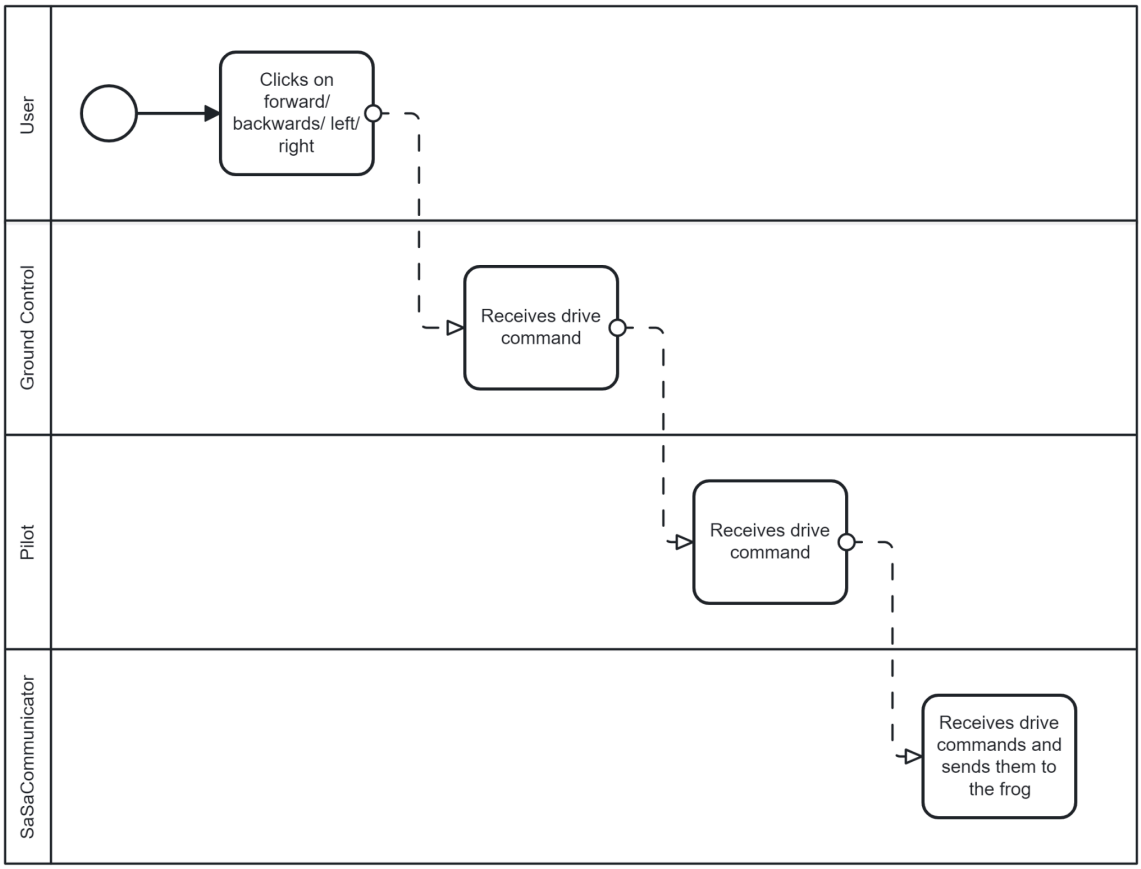
- Command Flow:
  User inputs (through TerminalViewController or MapController) -> Pilot -> SaSaCommunicator -> Frog
- Feedback Flow:
  Frog -> SaSaCommunicator -> Pilot -> (TerminalViewController / RadarPanelController / MapController)
- Logging Flow:
  Pilot -> Logger -> (CSVLogger / Database)

Activity diagrams



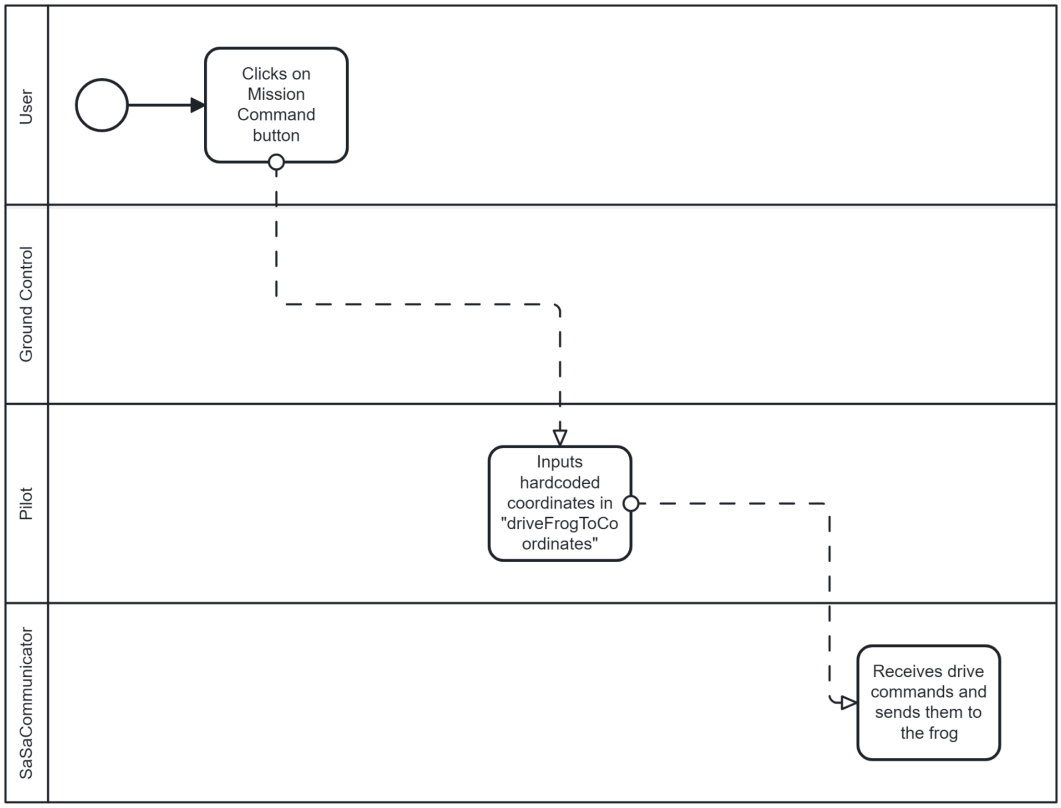
Steps:

1. **User**: the user initiates the process by clicking on a map. The clicked point is processed to the next step.
2. **Ground control**: the coordinates corresponding to the location clicked by the user on the map are received by the Ground Control System.
3. **Pilot**: the received coordinates are then input into a separated method called “driveFrogToCoordinates”. This method talks to the SaSaCommunicator.
4. **SaSaCommunicator**: finally, the SaSaCommunicator receives the driving commands for the pilot and sends them to the frog via its own methods.



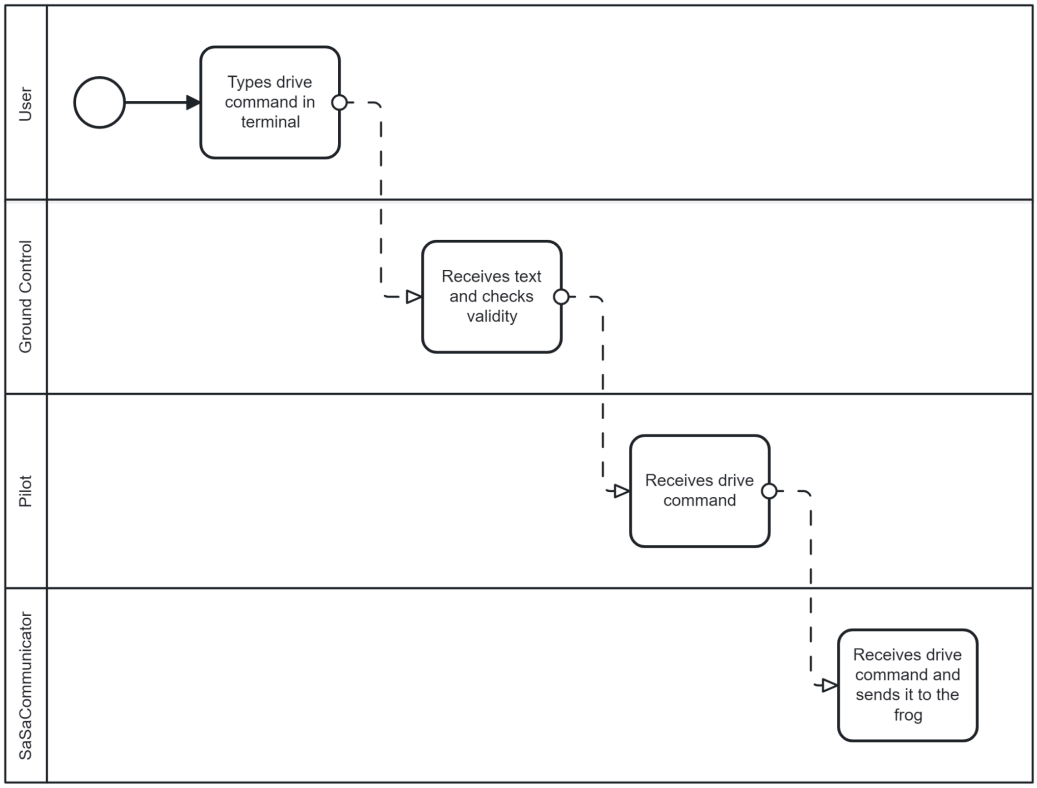
Steps

1. **User:** the user initiates the process by clicking on a direction (forward, backward, left, or right).
2. **Ground control:** the drive command corresponding to the user's click is received by the Ground Control system.
3. **Pilot:** the Ground Control System then forwards the drive command to the Pilot system method.
4. **SaSaCommunicator:** the SasaCommunicator receives the drive commands from the Pilot and sends them to the frog.



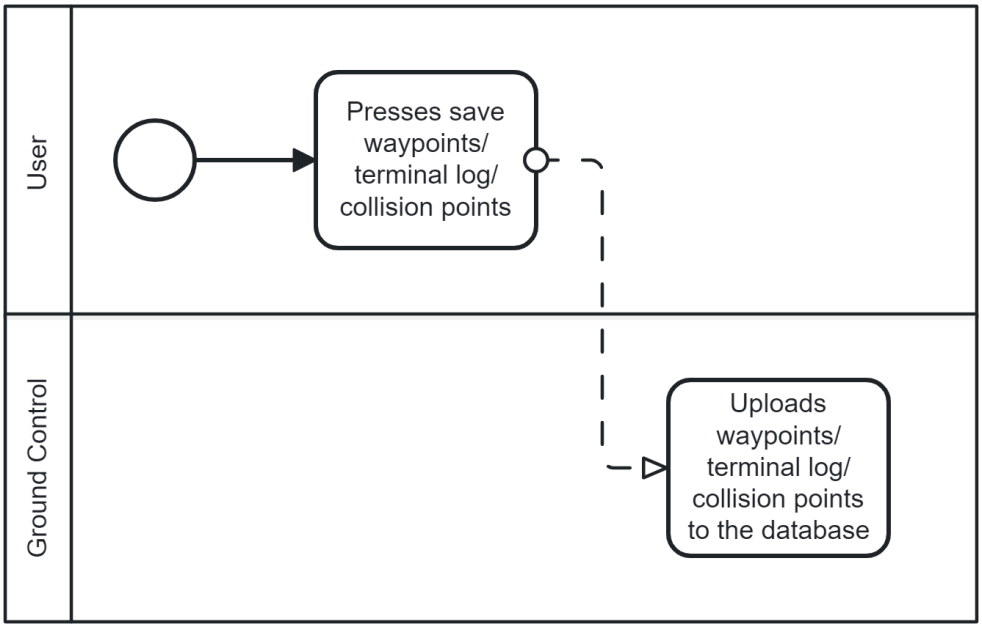
Steps

1. **User:** the user clicks on mission control buttons and then proceeds to the pilot directly.
2. **Ground control:** -
3. **Pilot:** then the pilot receives the button input and starts the “driveFromToCoordinates” with already hardcoded coordinates.
4. **SaSaCommunicator:** the SasaCommunicator receives the drive commands from the Pilot and sends them to the frog.



Steps:

1. **User:** the user types the command to the terminal window and presses enter to proceed to the next step.
2. **Ground control:** the ground control then receives the input and validates it so the pilot can use the final command without any confusion.
3. **Pilot:** the pilot receives the command and uses its method to contact the SaSacommunicator..
4. **SaSaCommunicator:** the SasaCommunicator receives the drive commands from the Pilot and sends them to the frog.



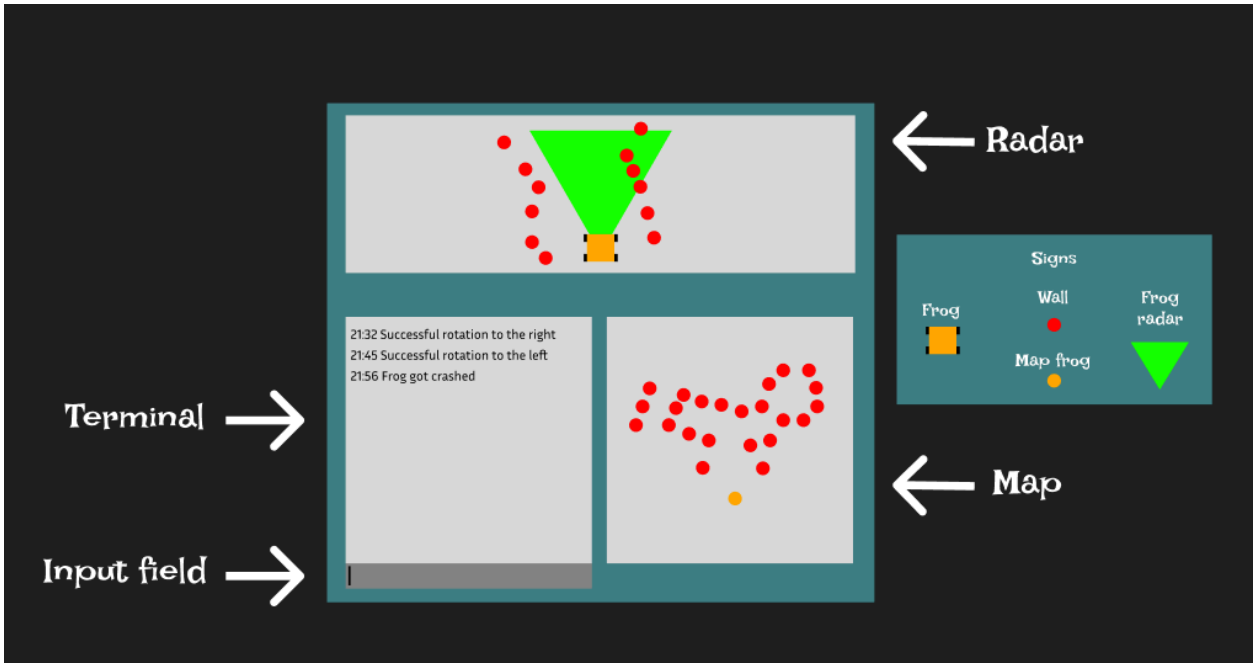
Steps:

1. **User:** the user presses the save button on the option that he wants (waypoints, terminal log, collision points).
2. **Ground control:** then ground control upload selected data to the database and provided csv files.

2. Wireframes

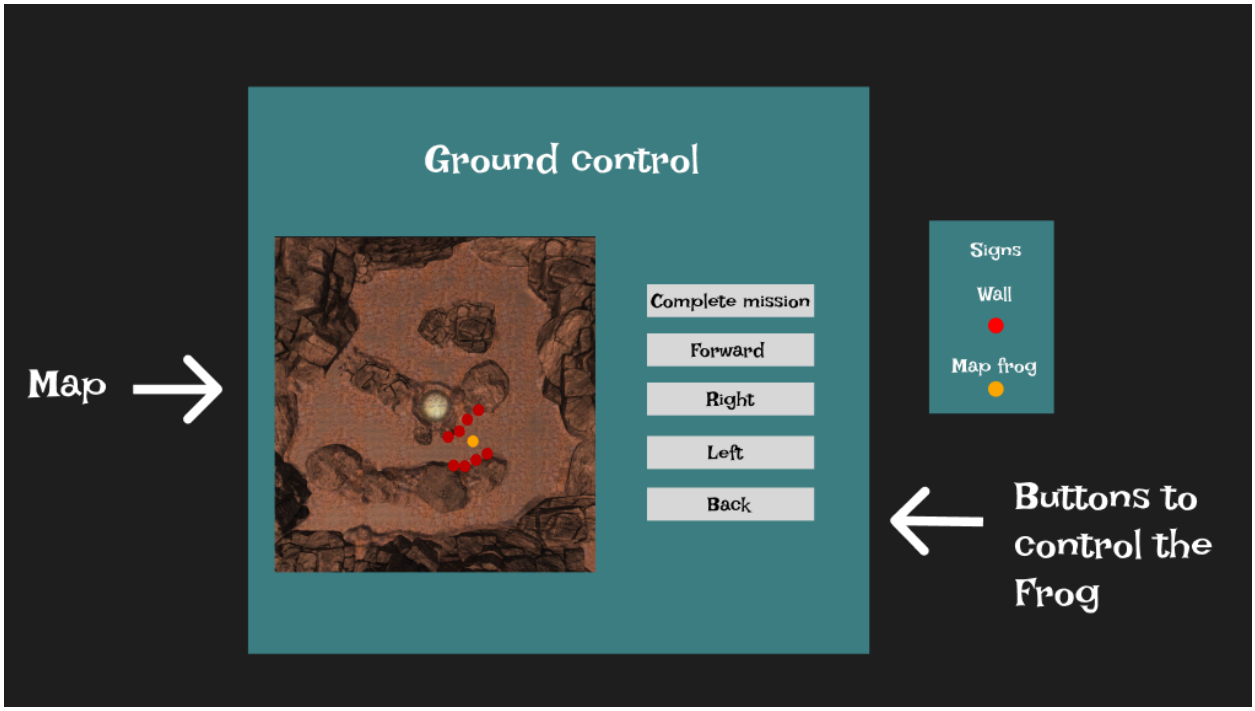
The wireframes were created to visually plan how the user interface of the applications must look. Below is the wireframe for the Pilot application.

On the wireframe, the map and the radar view for the Frog are represented. There is also a terminal window for the application. It is evident that in the actual application, the implementation is different. First, for each feature it was decided to implement a separate window. So, there is a separate screen launched for the radar map and a separate one for the terminal. Second, the radar view (the one in the top of the wireframe) was decided to not be added to the application, as the radar map already covers its functionality, Third, the terminal was included in the Ground Control application rather then in the Pilot, as the team misunderstood the requirement at first.



The ground control wireframe resembles the application designed more closely. There is a map with buttons to move as well as one to complete the mission. In the application designed, there are three mission buttons for each step. Also, the map shows

the trace of the frog and the chosen destination point.



3. How to extend code

To extend the code of “the Frog”, a developer needs to understand the difference in the functionality in the Ground Control module and Pilot module. The Pilot module is responsible for the algorithmic aspect of the Frog. So if the code has to be extended in terms of pathfinding and obstacle avoidance, a developer must work with the Pilot module. To get the data from the radar of the Frog, a developer can refer to the set called “processedRadarPoints” in the class RadarPanelController. If a developer needs to create a new class that interacts with the Frog, they have two options. The first option is to initialize a new SasaCommunicator in the new class. This allows the class to send commands to the Frog. The second option is to use the Pilot instance in the newly created class. As the Pilot already is initializing a SasaCommunicator, the class can connect to the Frog through it.

The classes in the Ground Control module cover mostly GUI functionality. This is the module, which makes the ground control map interactive and adds buttons to it. This module works closely with FXMl, so if a developer needs to enhance any of the application windows, they need to initialize visual objects with annotation @FXML and generate an fxml module either by using scene builder or manually. To control a graphic window, the developer needs to create a controller class that is linked to fxml module. Most importantly, to make the application run together with the other ones, the developer has to add a start script to the Main class in the ground control module. Then, when the Main class is run, the 3 already existing windows with the new ones will open.