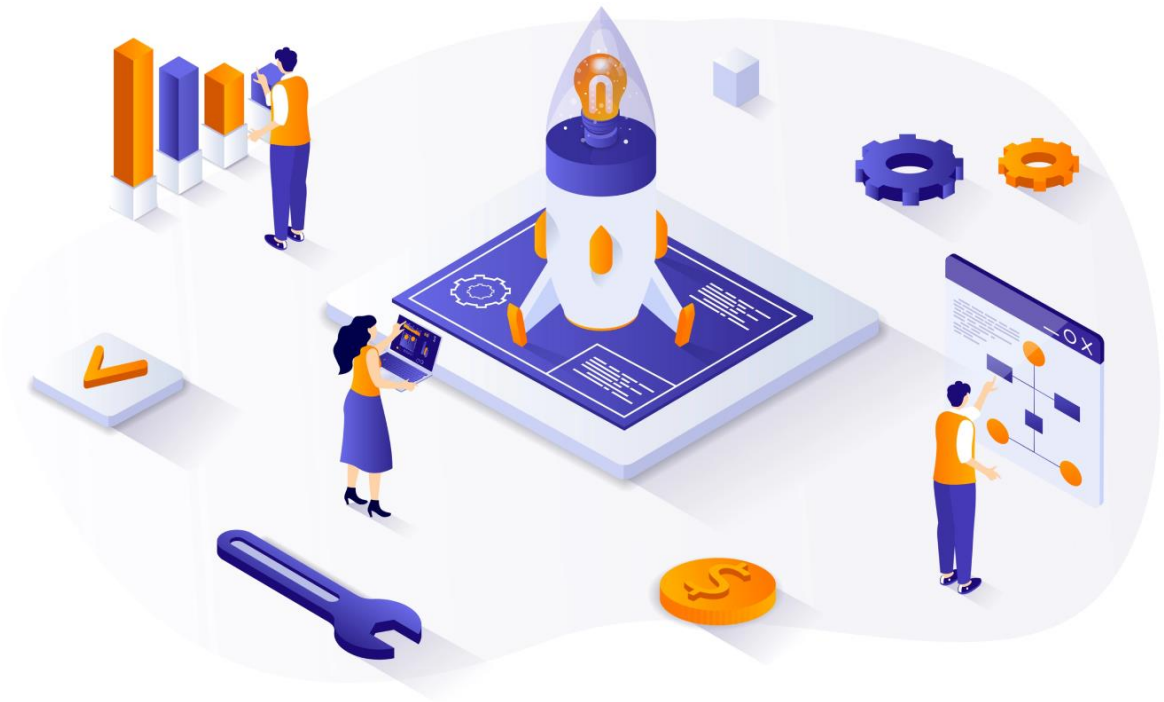
Technical Design



Technical Design

Table of Contents

Introduction

This is the Technical Design Document or the addition to the Functional Design Document. It will contain all the content not covered by the Functional Design. The focus of the Functional Design document is to grasp a better understand of the overall software, the Technical Design Document will instead focus on in depth details. These in-depth details refer to the overall documentation of the software.

Within this documentation the general layout of the software will be covered. This will include the file layout of the project and the architecture of the code. Communication between all the different nodes using the Sasa Communications library will also be explained and why these design choices were taken.

The Database design and its connection to the software will be expanded on and diagrams will be shown to give the reader a better understanding of the software.

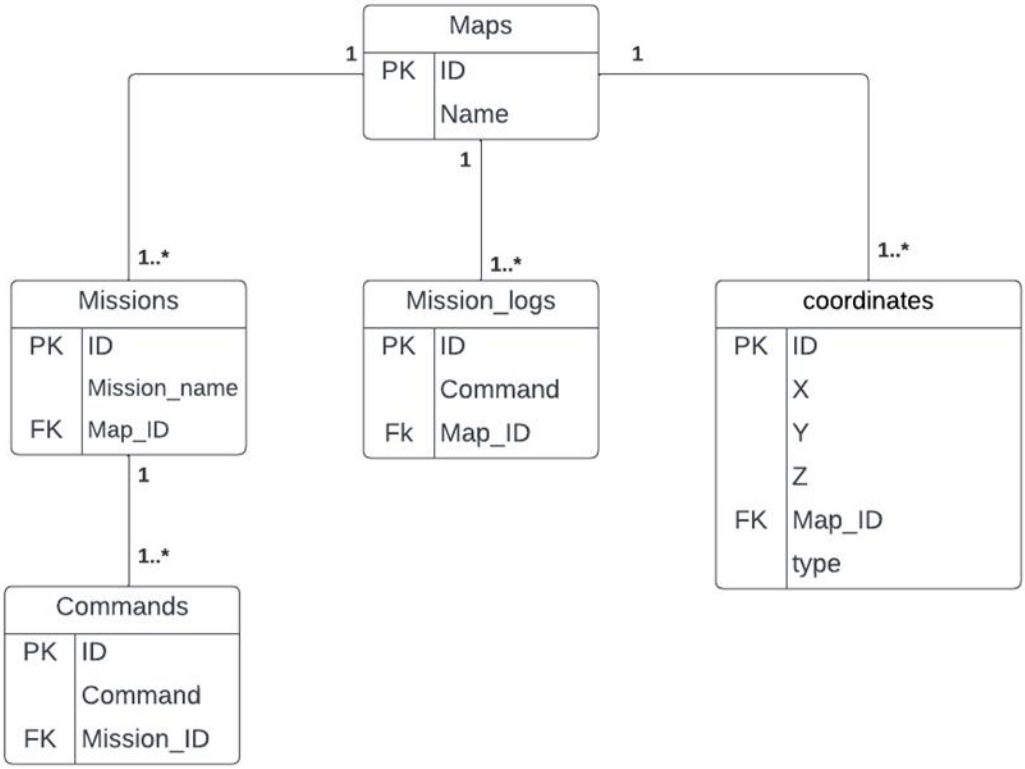
The applications to be built in this project are **The Pilot** and **Ground Control**, which aim to satisfy the needs of the Saxion Space Academy (SaSA) in carrying out their lunar mission with The Rather Rudimentary Operated Lunar Vehicle, also known as The Frog.

Since The Frog does not have a software to make autonomous decisions, The Pilot will be created to help navigate The Frog automatically based on radar data and commands from the ground station on the earth. This application will also deal with storing and sending data back to the ground station.

Since the ground station does not currently have a way to communicate and control The Frog remotely, Ground Control will be created. This application will provide the interface to control The Frog remotely, as well as path tracking and displaying of the obstacles detected via radar.

For our code structure we used the MVC model in order to better organize our code since just after the first commits to the repository were made our code base was a mess and nobody could understand what was happening in there.

Database design



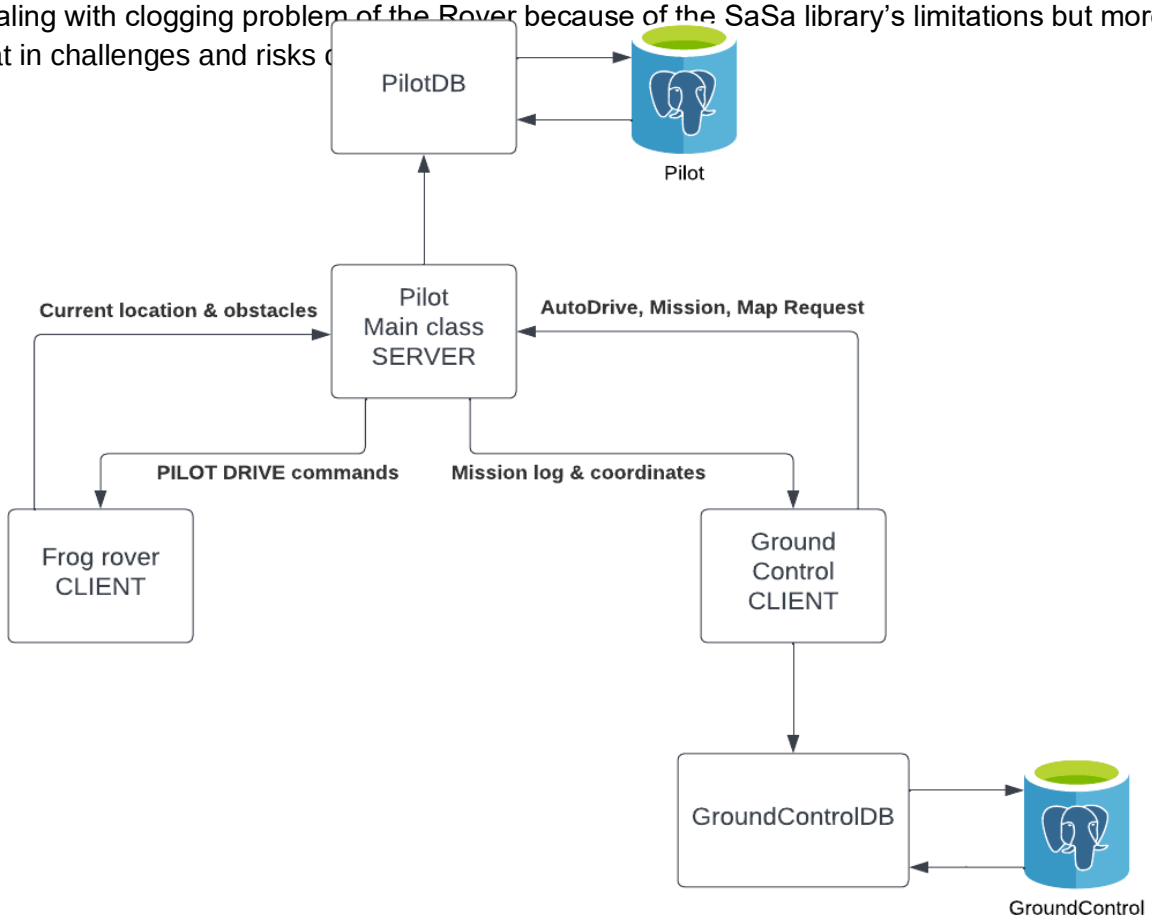
The diagram for both DB is almost identical with one exemption. For the Database to be normalized we decided to save all coordinates in the same table. There are 3 types of coordinates: “location” coordinates are the current locations of the frog; and they are relative to the map; “collision” coordinates are the collision point the frog detects and they are relative to the frog itself: and “endpoint” coordinates which are the destinations given by the operator the frog to go to automatically. The reason why we design it this way is to be more efficient since for all 3 types of coordinates we are storing the same thing ((x, y, z) coordinate, double precision). As far as why the diagrams are identical, well we put a lot of thought into this, and this is the only way I can see to design the DB’s and meet the requirements. The structures might look the same but the information they will contain will be different. Our DB is normalized to NF3 because the structure of the DB will not allow duplicate data to be stored. We have map table even though we only have one map to work with this project when designing we thought to include this table because it would be nice if we ever decide to scale up the application to work on multiple maps to already have it in our structure.

For installing both Databases user needs to create 2 DB in PgAdmin and called them Pilot and GroundControl. The server needs to be on port: 5432; the username is postgres and password is 1234

SaSa communication library API specification

The diagram below illustrates how the general communication between the different pasts of the project is happening. Between the 2 applications we have a DB class that stores all the queries that need to be executed and makes the connection with the DB using JDBC.

In the beginning of the project, we had a bit of more complicated communication setup between the pilot and the Ground control where both pilot and Ground Control are Servers however later in the development of the project we decided to drop that down because it overcomplicated the communication, however the older structure did prove beneficial for for dealing with clogging problem of the Rover because of the SaSa library's limitations but more on that in challenges and risks c



The Pilot API specifications

The Pilot working as a server in our implementation is listening from both The Frog (Rover) and the Ground Control for messages and responding accordingly (communication does not necessarily always start from one side) here we will list the messages it is listening for:

1. **AUTODRIVE {X} {Y} {Z}:** This is a message that will come from the Ground Control. Basically, with this message we send coordinates for the automatic driving algorithm to go to.
2. **FROG POINT {X} {Y} {Z}:** This message is sent by the rover, and it is sending coordinates of a collision point.
3. **FROG POSITION {X} {Y} {Z}:** This message is sent by the rover, and it is the current position of the Frog.
4. **MISSION: {MISSION_NAME}; {MISSION_COMMANDS}:** This is a long sting message that is being sent by the Ground control and it is basically a newly created mission where first parameter will always be the name and everything else will be commands and they are separated by a ";" sign.
5. **MAP REQUEST:** This message is sent by Ground Control when user wants to open the map the response to this message is the current location of the rover and its collision points.

Ground Control API specification

1. **MLD:** This message is sent to Ground Control right before sending new mission log update so that the previous mission log can be cleaned from the DB
2. **MLR {COMMAND}:** This message is sending a single command from the Pilot DB
3. **MAP RESPONSE: {CURRENT_POSITION}; {COLLISION_POINTS}:** This command sends to Ground Control all the necessary coordinates for the maps in one log string;

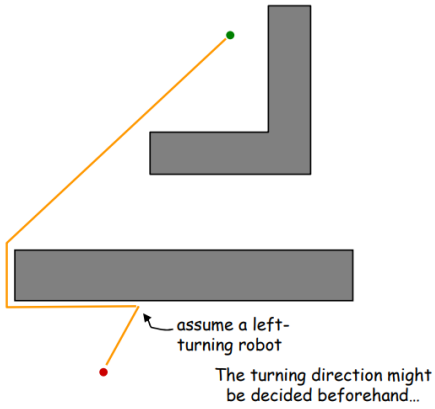
Frog API specification

The Frog API specification can be found in the original document provided to us for development of this project however we decided to list the commands we used.

1. **PILOT DRIVE {MOTOR_POWER} {ANGLE} {TIME}:** This is the only command/message we decided to use for controlling the frog, we didn't use any other commands for the radar ect...

Path finding algorithm

Initially we intended using the Bug 2 algorithm, but there are p challenges and risks with implementing this algorithm, so we used BU basically does is it faced its destination goal, heads towards it in a stra obstacle on its way, it will follow the obstacle wall until it can resume t



We simplified the wall-following action. There is a danger-zone and a safe-zone. Danger-zone is defined from danger-distance. If the rover is closer than danger-distance from the wall, it moves for its safety. (Stops or go backward). Safe-zone is defined from safe-distance and danger-distance. In this zone the rover acts to avoid obstacles. It is a little different from the original bug algorithm. However, the rover has no side or back sensors/radars so when turning while avoiding obstacles it may hit its wheels because of the lack of orientational knowledge it has on its sides and back. The Rover turns about 80 degrees and goes forward a little (about 1m) and used big precision for angle adjustment. For faster simulation we used big mortar power as 500~1000, however it is not the most accurate. For accuracy, the values can be small, but it takes a lot longer to reach its destination. The details are implemented in the code. Users can review the code comments.

The Bug algorithms are easy to implement and do not require heavy hardware. But it has problems. The main is that the turning direction is determined beforehand. The direction can be left or right. If the environment is good for left turning it succeeds. But turning right may not work. For our test we used mostly the left turning method. If users change the start point and destination at random, the test may fail. If the test environment is complex that requires left and right turning both, the rover can circle or stuck repetitive action. To avoid this, users should set temporary goals in the path.

Added methods

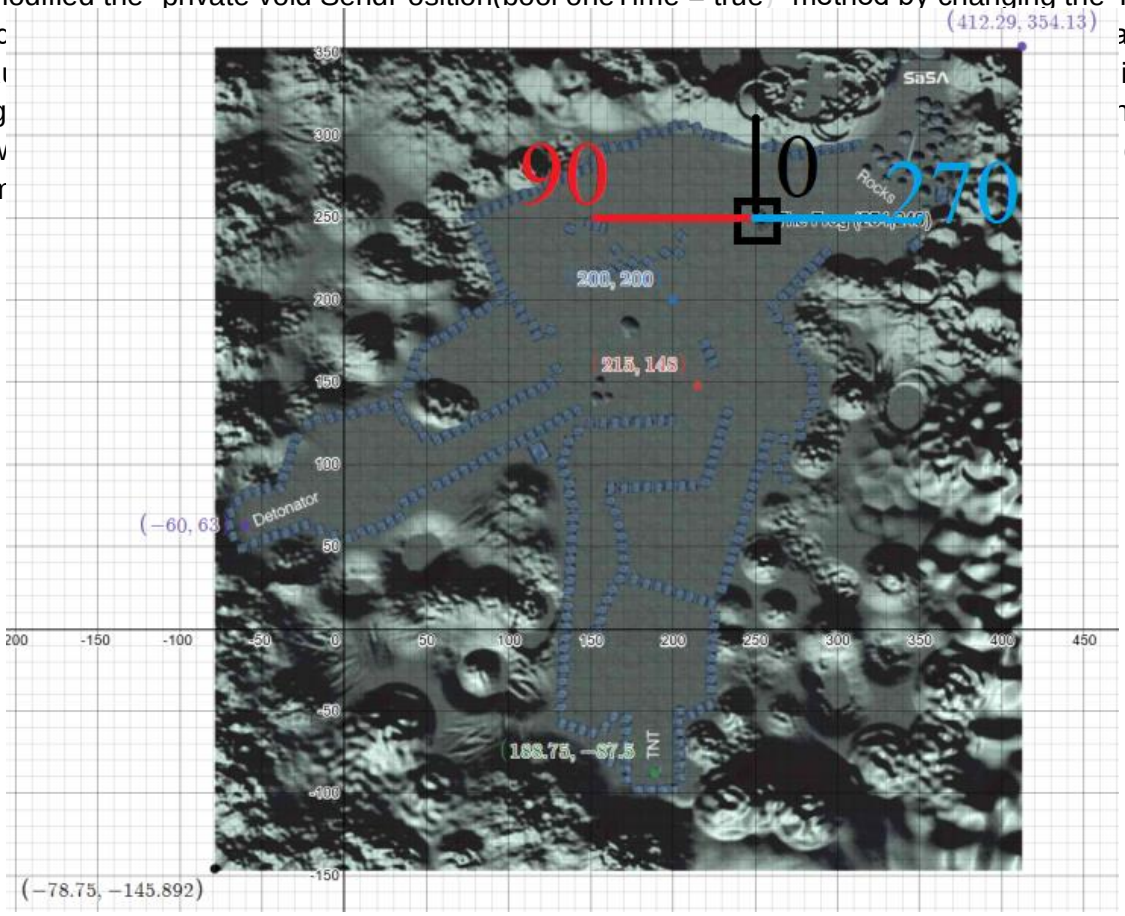
1. **getObsInDistance:** This function gets the obstacle list in each distance.
2. **Drive:** Runs the rover for 1 action, this function works after the rover's current position is decided, it may execute several commands, and it requires correct current position and obstacle data, so sleep is required before or after the function execution.
3. **action_danger:** Command the rover to move backward, users can change the motor force and duration
4. **Auto:** This function provides automatic drive. Caution: TimeUnit.SECONDS.sleep(n) function is often used for correctly receiving rover and Obstacles' positions.

5. **getAngle:** Gets the angle of line between a and b the result is (-180,180) degrees.
6. **goTo:** Drive to destination (only used when there are no obstacles in between). Turn to the GOAL_POINT direction and go straightthe direction tolerance is ANGLE_PRECISION
7. **avoidOb:** This function avoids the nearest obstacle. Implemented bug action.

Frog/Rover Unity project modifications

As mentioned in the Challenges and risks chapter during the development of the path finding algorithm there were many issues that arose but the main problem, we discovered was the orientation of the rover. There was an attempt to solve this issue by comparing the previous position and the new position and determining the global orientation based on that data, however that was very challenging because a small drive forward commands needed to be performed after each big command and it was almost impossible to not hit obstacle if 90 degree turn is to be performed in the maze so we decided to modify the unity project so we can receive the global orientation.

What we did was using an open-source program called ILSpy to decompile the "Assembly-CSharp.dll" file into a C# code. Then we used IDA to patch the code with hex view we modified the "private void SendPosition(bool oneTime = true) " method by changing the Y coorc at is becau isn't flying n and the w on our m



The Pilot – Sasa Communication Library (Sasa Server)

Tech stack: Java. No alternative is possible for this tech stack, as it was specified by a “Must” Business Requirement.

The Pilot is responsible for processing messages/tasks from Ground Control and passing the appropriate commands to The Frog and in return, while, in the opposite direction, also listening messages from The Frog and pass it back to Ground Control. The Pilot uses a Sasa Server to facilitate communication-related tasks – this design decision is based on the medium role of The Pilot in communication between Ground Control and The Frog. In addition, the manual control of the frog is implemented in the Pilot.

The Pilot contains a UI instance (of “UserInterfaceTP” class), a database manager instance (of “PilotDB” class) and an Automatic Driving instance (of “AutomaticDriving” class).

Current functionalities:

1. **Startup of The Frog**: the executable Frog simulator is automatically booted when starting up The Pilot
2. **Piloting of The Frog:** prepare a drive command message and forward this command to The Frog for it to drive. The source of the drive command can be manual driving from Pilot UI, pre-programmed mission, or automatic collision avoidance algorithm. Also dealing with storing this command to The Pilot’s database
3. **Sending a mission log to Ground Control:** retrieve the currently stored drive commands from database, process these data and send them to Ground Control
4. **serverSendMessage method**: a workaround that allows the Pilot UI to send message using a Sasa Server (to be elaborated in the “Challenges/Risks” chapter)
5. **Storing data:** It stores and processes any upcoming data from both clients

(The Pilot) User Interface

Tech stack: Java, Swing. No alternative is possible for this tech stack, as it was specified by a “Must” Business Requirement.

User interface of The Pilot is created to fulfill the Business Requirement RRB-04: “The application has a (preferably Java Swing) graphical user interface”. Components of this UI are defined in the “UserInterfaceTP” class, and an instance of the interface is generated when starting up The Pilot.

Current functionalities:

1. **Manual driving of The Frog**: using the 6 direction buttons and a braking button in the GUI, the operator can manually control The Frog. Each button, when clicked, send a suitable corresponding drive command to The Frog for it to drive.
2. **Display the mission log:** It displays the mission log The Pilot database to a panel on the GUI. This is automatically done once at startup of the UI.
3. **Refresh the mission log pane:** This updates the screen with the current mission log that exist in The Pilot database.
4. **Send mission log to Ground Control**: When clicking this button on the GUI, commands in the “mission_log” table of The Pilot database are retrieved and sent to the Ground Control as Sasa messages, using an encoding format (prefix “MLR”) as explained in the “API specification” section.
5. **Pre-programmed missions:** This Panel gets whatever preprogrammed missions have been sent to the Pilot in a list. The user then can select a mission and execute its commands.

(The Pilot) Database – JDBC

Tech stack: Java, Postgres. No alternative is possible for the use of Java, as it was specified by a “Must” Business Requirement. The use of Postgres is motivated by its familiarity within the organization (used in a previous database course) and its nature of being free & open source. Possible alternatives for Postgres are MySQL, SQLite, Oracle Database, etc.

Database-related activities of The Pilot is handled by “PilotDB” class. This is where connections are made, and SQL queries are sent to the database.

Current functionalities:

1. **Update mission log:** adds a command to the mission log table
2. **Storing Drive to location coordinates:** It stores any coordinates received from the Ground Control for the algorithm to reach.
3. **Storing a received pre-programmed mission:** The mission and its command are stored to The Pilot database. Information about the missions (e.g., mission name and mission ID) is stored in the “missions” table, the commands of the missions are stored in the “commands” table
4. **Read mission log:** It prints out the contents of the mission log.
5. **Connect:** It connects to the DB

Ground Control – Sasa Communication Library (Sasa Client)

Tech stack: Java. No alternative is possible for this tech stack, as it was specified by a “Must” Business Requirement.

The Ground Control is responsible for processing messages from The Pilot and perform tasks based on the message received. Ground Control contains a GUI (“UserInterfaceGC” class) instance to handle inputs/tasks from user and send these inputs to The Pilot using a Sasa Client instance, it also contains a database manager instance (of “GroundControlDB” class).

Current functionalities

1. **Client send message:** It relays messages from the UI to the Ground control

(Ground Control) User Interface

Tech stack: Java, Swing. No alternative is possible for this tech stack, as it was specified by a “Must” Business Requirement.

User interface of The Pilot is created to fulfill the Business Requirement RRB-04: “The application has a (preferably Java Swing) graphical user interface”. Components of this UI are defined in the “UserInterfaceTP” class, and an instance of the interface is generated when starting up The Pilot.

Current functionalities:

1. **File chooser options**: You can choose to either import/export a mission/collision points
2. **Drive to location**: User can enter coordinates to be relayed to the automatic driving algorithm to execute
3. **Map**: User can display 2 maps. One is the global location, and the other is the current collision points location

4. **Create mission**: User can create mission and add commands to that mission. Set mission will be relayed to the Pilot.
5. **Mission log**: User can update the mission log in the Ground Control application.

(Ground Control) Database – JDBC

Tech stack: Java, Postgres. No alternative is possible for the use of Java, as it was specified by a “Must” Business Requirement. The use of Postgres is motivated by its familiarity within the organization (used in a previous database course) and its nature of being free & open source. Possible alternatives for Postgres are MySQL, SQLite, Oracle Database, etc.

Database-related activities of Ground Control is handled by “GroundControlDB” class. This is where connections are made, and SQL queries are sent to the database.

Current functionalities:

1. **Connect:** Connects to the Ground Control DB
2. **Import missions:** Imports mission from an SCV file to the DB
3. **Export mission:** Exports missions to SCV file
4. **Import collisions:** Imports collision points to the DB
5. **Export collisions:** Exports collision points to an SCV file
6. **Read mission log:** Gets all the stored data in the mission log table
7. **Insert mission log row:** Inserts a single command to the mission log
8. **Delete mission log:** Deletes the mission log table’s contents. It is necessary so there wont be any duplicate commands
9. **Add mission:** Adds a single mission to the DB
10. **Add drive to location:** Adds coordinate to the DB

Styling Utilities

The 2 “StyleUtils” classes in the Ground Control and The Pilot packages provide styling to the components of the GUIs, for aesthetic purposes and improvement of user experience.

Current functionalities:

1. **Style a button** based on a preset of criteria (background color, font, etc.)
2. **Style a writable** text field based on a preset of criteria (background color, font, etc.)
3. **Style a panel** within the GUI and its objects based on a preset of criteria (background color, font, etc.).

Challenges and Risks

Our biggest challenges and risks came with the development of the path finding algorithm. The main problems we encountered were related to the were coming from the limitations of the environment provided to us.

I want to start with the SaSa communication library because it proved quite challenging to control the Frog with its current design. There were many bugs we encountered while working with it, the main one I can recall was the PILOT DRIVE command given with 0 as a time integer. Practically it became impossible to give a new command after that, so we decided to never use this option. Another problem came up when we were working on the map. It became very clear that if we tried to immediately relay messages to Ground Control when received by the Pilot from the rover the rover will get clogged with messages. This is the results of the slow I/O speed between server and client. This is also the reason why in automatic driving and in map display we put a timer of between 2 to 6 seconds so all the information can arrive.

When it comes to the unity project there are so many issues that could probably fit into a phone book. I will start with the dimensions of the rover which weren't provided to us and we had to use map glitches to figure them out, the physical parameters of the ground, motor power and speed can not be properly tested which means that any continuous planning or control is impossible to implement here, the control of the frog can only happen once it has executed a command so testing any path finding algorithm takes very long time. The radar system is broken in my opinion the rover should scan in 360 degrees and provide the server with real time data. The rover should also send its orientation on the global map, and it should be controlled in real time. Because of these reasons the biggest risks in our project are in the automatic driving. When rover is circling around an obstacle because of the radar's blind spots it might hit its tires on the corner of the obstacle.

Quality Assurances

One of the ways that we assured quality assurance is a test class. This was created as an example of how we would communicate between the nodes (Specifically the client and the server). This is located in the file _TestConnection within the project. Each team member followed this design pattern to assure quality of communication between all different components.

Another method followed to insure that our code was of a certain standard in quality is following an MVC Design pattern. This was done to gain a clear understanding of a separation of concerns within the project. The Model directory is to handle any and all interactions with the Database and to not have these database calls scattered throughout the project. Similar with the View and Controllers directory. Everything to do with the UI was located within the View directory to again insure a clean format and understanding of our directory and code structure.

Git was used to version manage. This insured that we would always have a working and running version of the code. Our main working directory was called “develop”, when a team member had completed their feature on a branch specific to them (usually following the naming convention of branch/<team_member>), their branch would be merged back into develop for everyone to continue.

References [1, 2]

[1] C. M. University, "BUG 0," [Online]. Available: https://www.cs.cmu.edu/~motionplanning/lecture/Chap2-Bug-Alg_howie.pdf.

[2] R. p. f. d. p. f. algorithms. [Online]. Available: https://pdf.sciencedirectassets.com/280203/1-s2.0-S1877050921X00026/1-s2.0-S1877050921000399/main.pdf?X-Amz-Security-Token=IQoJb3JpZ2luX2VjEBQaCXVzLWVhc3QtMSJHMEUCIQCkOmXy9%2F0C%2FgCDS5p3%2BbYBThKq5Y3l2mFWR3EbjxEbPwIgUayzWTHHapgEV3acvYPPwU3giq%2F%2F3ZxFrB.