

NATIONAL ADVANCED SCHOOL OF ENGINEERING
YAOUNDE

COURS DE SYSTÈME D'EXPLOITATION

OPERATING SYSTEM COURSE

BIKE PROJECT

Supervisors :

Pr Thomas DJOTIO NDIE

Mr Juslin KUTCHE NEGOUE

December 7, 2023

Contents

1	Introduction	3
2	Description of the Problem	4
3	Our solution	5
3.1	Random Process Generator(RPG)	5
3.1.1	Client-RPG	5
3.1.2	Bike-RPG	5
3.2	Client module	6
3.3	Bike module	6
3.4	Central Scheduler	7
3.5	simulation:User Interface	8
4	Implementation	9
4.1	Differents libraries,structures and data structures	9
4.2	RPG (Random Process Generator)	12
4.3	Clients	14
4.4	Bike	16
4.5	Central Scheduler	19
5	User Interface	20
5.1	Client and bike	21
5.2	Scheduler	22
5.3	UI	23
6	Test and evaluation	25
6.1	without interface	25
6.1.1	scheduler	25
6.1.2	scheduler with RPG	26
6.2	with interface	28
7	Conclusion	34

List of members

1. BEKONO BINDUGA Florian Donald (21P301)
2. BENGONO AMVELA Nathan (21P091)
3. DONCHI Tresor Leroy (21P107)
4. FEZEU YOUNDJIE Fredy Clinton (23P751)
5. KENGALI FEGUE Pacome (21P027)
6. KOGHENE LADZOU Eric (23P752)
7. KOUDJOU TIEMIGNI Vicrand (21P190)
8. MBASSI EWOLO Loic Aron (21P340) (Project Manager)
9. MBEYA NDONGO Joel Hyacinthe (21P341)
10. MEKIAGE Olivier (21P369)
11. MOGOU Igor Green (21P246)
12. NGANTA YATCHOUA Ange Pacifique (21P012)
13. NGO BASSOM Anne Rosalie (21P089)
14. NOMO BODIANGA Gabriel Nasaire junoir (23P753)
15. NTYE EBO'O Nina Laissa (21P223)
16. VUIDE OUENDEU Franck Jordan (21P018)
17. WOTCHOKO NGATCHEU Yohan (21P228)

1 Introduction

The transportation of customers is an essential activity for transportation companies, especially for motorcycle taxi providers. It is indeed about guaranteeing customer satisfaction by offering them fast and reliable transportation. However, customer transportation poses a challenge, as it requires effectively managing a multitude of parameters, such as customer location, motorcycle availability, and delivery times. In this context, the motorcycle project we are carrying out aims to develop a customer transportation algorithm that respects a certain policy, while simulating the natural operation of a motorcycle in reality. The customer transportation policy consists of defining the selection criteria for customers to be transported, as well as the order of priority of requests. The customer transportation algorithm must be able to respect the defined policy, while optimizing the use of available resources. The operation of a motorcycle in reality includes the behavior and characteristics of a real motorcycle, such as: looking for a customer when the motorcycle is empty, going to a given destination when the motorcycle has a customer...etc. Thus, we ask ourselves the question of what problem does this problem situation raise? What description can we make of this problem? What modeling will best simulate reality? and how to implement this modeling to have the most optimal solution? Answering these questions in the following will clarify the terms of this project while defining a policy and an appropriate mode of operation for our motorcycle.

2 Description of the Problem

Our challenge is to develop a communication management solution between motorcycles and users, initially focusing on simplicity and speed. The goal is to design a program capable of efficiently gathering data from motorcycles and customers, transmitting them to the appropriate drivers, and returning optimized responses. This initial approach aims to enhance coordination between the two parties, allowing customers to save valuable time during their city journeys. It will also benefit motorcycle drivers by reducing the time needed to select their customers.

In a later phase, our challenge is to consider expanding the program's scope to include the management of taxis, buses, and all road transport means. This will require a more advanced optimization of the system, with the ability to integrate various modes of transport. The ultimate goal is to create a comprehensive and efficient transportation management ecosystem, providing a holistic solution for city transport needs.

Accurate data collection on motorcycles, clients, and traffic conditions will be very crucial for the success of this project. We will work to establish a smooth mechanism for information transmission, thus promoting optimal communication among all system stakeholders. In summary, our challenge is to transform the way transportation is managed in the city, offering a technologically advanced solution that benefits both users and transport service providers.

Our perspectives for further optimization in the future include:

- Implementing a remote ordering system for clients, enabling them to request a nearby motorcycle from the comfort of their homes.
- Incorporating real-time notifications to keep clients informed about the imminent arrival of their motorcycle and alerting drivers to new ride requests.
- Integrating secure electronic payment options, diversifying methods such as credit cards and digital wallets.
- Establishing a robust rating and feedback system for users and drivers, enhancing service quality and building trust.
- Introducing a loyalty program to reward regular users and dedicated drivers, fostering platform loyalty.
- Integrating weather data to anticipate challenging driving conditions and inform users in advance.
- Implementing real-time traffic management for drivers to optimize routes and avoid congestion.
- Offering personalized preferences for users, such as music selection and driver style.

3 Our solution

To tackle this issue we will be subdividing the problem in to 5 different modules which include:

- Random Process Generator
- Client module
- Bike module
- Central Scheduler
- Graphical Interface

Each part with their various specifications which are elaborated as follows

3.1 Random Process Generator(RPG)

The RPG (Random Process Generator) serves as a crucial tool in simulating the behavior of real-world customers and bikers within a ride command system.

This RPG as stated above is used for customers and bike generation and is further explained as follows:

3.1.1 Client-RPG

The RPG (Random Process Generator) is a crucial tool designed to simulate the behavior of real-world customers within a ride command system. In parallel, it generates ride requests for bikes at random intervals, replicating the unpredictable patterns of user behavior in a live environment.

Specifically designed for this context, the RPG generates ride requests for bikes and customers at random intervals, replicating the unpredictable patterns of human behavior in a live environment.

3.1.2 Bike-RPG

In parallel, another component of the system creates bikes to fulfill these randomly generated ride requests. This dual simulation process mimics the dynamics of the system's response in providing the necessary bikes to meet those requests within the framework.

This approach allows developers to assess how well the system handles the variability in user demands, ensuring its ability to efficiently respond to spontaneous ride requests and allocate bikes accordingly. This thorough testing contributes to enhancing the overall reliability and effectiveness of the ride command system in meeting the dynamic needs of its users.

3.2 Client module

For this section, we have decided to use shared memory segments. The following provides an overview of how the client operates.

On creation of a motorcycle the following steps are followed:

1. **Information Generation:** by using four random values, the first determining its starting point, the second defining its final point and the third being its process id and the time waiting time limit, thus establishing all the necessary informations.
2. **Shared Memory Segment Creation:** attaching to the client a shared memory in read-write mode.
3. **Signal Transmission to Scheduler:** It then sends a signal to the scheduler informing the completion of the initial part execution.
4. **Signal Awaiting:** It waits for the scheduler's signal which will continue until it reaches its limit. If the time allocated is over happens, the process is then terminated and send a signal to the scheduler and is removed from the client list.
5. **Signals Reception:** It receives a signal from the the bike informing them that they have been carried and. At the end of the ride it receives a signal from the bike stating the end of the ride and gets terminated.
6. **Signals Emission:** It finally sends a signal from the the scheduler stating the end of the procedure.

3.3 Bike module

For this section as the one above, we have decided to use shared memory segments. The following provides an overview of how the motorcycle operates.

On creation of a motorcycle the following steps are followed:

1. **Route Creation:** by using two random values, the first determining its starting point and the second defining its "radius" thus establishing its itinerary.
2. **Shared Memory Segment Creation:** attaching to the bike a shared memory in read-write mode.
3. **Signal Transmission to Scheduler:** It then sends a signal to the scheduler informing the completion of the initial part execution.
4. **Signal Awaiting:** It waits for the scheduler's signal which is reading the list clients stored in the shared memory segment and since we decided to implement the FIFO policy, it selects the first client in the list.

5. **Signal Reception:** It receives a signal from the scheduler assigning it a client to serve.
6. **Signal Transmission to Client:** It then sends a signal to the client informing them that they have been carried.
7. **Waiting:** It then sends a signal to the scheduler indicating the transportation of a client. Which in our case is synonymous to going into the waiting state.
8. **End signal:** It sends a signal to the client indicating to end of it's process.
9. **Restart:** The procedure then restarts from the beginning.

As such we will have completed the total transport of a client and managed potential deadlock situations.

3.4 Central Scheduler

When creating the central scheduler the following steps are followed:

1. **Data initialization:** Here we start by creating two deques one for the clients and another one for the bikes which are present. And then arm the different signal handlers.
2. **Listening Mode:** The Scheduler now listens to the various signals from bikes and the clients for a set time t .
3. **Masking of signals:** After the said time, the signals sent as of now are not taken into consideration for the following scheduler operations
4. **Saving Clients and Bikes information:** It saves the different clients into the deque saving it using the FIFO policy and same is done for the different bikes using the other deque.
5. **Assigning Clients to Bikes:** Using the first fit protocole, the various clients are assigned to the available bikes and for this to happen, a signal is sent to the chosen bike and the chosen client is removed from client deque.
6. **Restart:** The procedure then restarts from the listening period and the masked items are added to the appropriate deques.

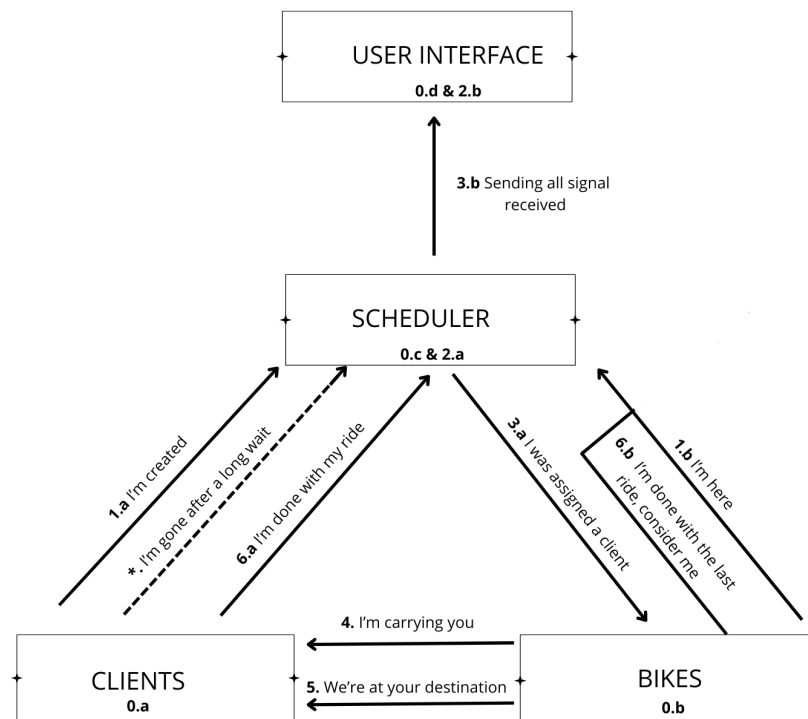
3.5 simulation:User Interface

Using OpenGL, we now create the different representations for the clients represented by a **blue circle**, the bikes represented by a **green square** and quarters differentiated by various colours and shapes.

Both bikes and clients only appear when they have been matched.

The following figure gives a brief summary of all what has been explicated above. The numbered sections 0.a, 0.b, 0.c, 0.d, 2.a and 2.b which couldn't be further explained on the figure respectively represent the following:

1. **Step 0.a:** This represents the information generation and the shared memory segment creation.
2. **Step 0.b:** This represents the route and the shared memory segment creation.
3. **Step 0.c:** This part is concerned with the data initialization and listening mode from the scheduler
4. **Step 0.d:** This part is concerned with the initialization of the user interface
5. **Step 2.a:** This part is concerned with the different saving of the different bikes and clients information.
6. **Step 2.b:** This part is concerned with the the treatment of all the various signals by the user interface.



4 Implementation

In this section, we will briefly outline the implementation of our solution using the C programming language.

We will start by presenting the different libraries, structures and data structures which will be used. To continue, discussing the **RPG**, then delve into the **clients**, **motorcycles**, and finally, the **scheduler**.

4.1 Different libraries, structures and data structures

During the implementation of our program, different libraries, structures and data structures were used among which we can cite :

libraries

The libraries we have utilized are:

- **stdio.h** : Standard Input/Output operations.
- **stdlib.h** : General utilities, including memory allocation and conversion functions.
- **sys/types.h** : Definitions for various data types used in system calls.
- **sys/ipc.h** : Definitions for interprocess communication using IPC.
- **sys/shm.h** : Definitions and functions for shared memory operations.
- **sys/signal.h** : Definitions for signal handling.
- **sys/wait.h** : Declarations for wait functions.
- **math.h** : Mathematical functions.
- **unistd.h** : Standard symbolic constants and types, including functions like `fork()`.
- **string.h** : String manipulation functions.
- **signal.h** : Signal handling functions.
- **stdbool.h** : Boolean data type and values.
- **sys/stat.h** : Declarations for functions and macros related to file status.
- **time.h** : Time-related functions.
- **glut.h** to make the user interface sing opengl functions

structures and data structures

Listing 1: quarter

```
1
2     typedef enum QUARTER_YAOUNDE
3 {
4     OMNISPORT ,
5     POLYTECH ,
6     TRAVAUX ,
7     .
8     .
9     .
10    .
11    ESSOS ,
12    NGOUSSO_1 ,
13    QUARTIER_GENERAL ,
14    ECOLE_NORMALE ,
15    NKOA_BANG
16 } quarter;
```

Listing 2: client and bike

```
1     typedef struct client
2 {
3     /*
4      * Definition of a client type, which will references our process:
5      * one client is a process
6      * pid is the process id of the client (as we said, it's a process)
7      * start and dest are quarters, element of an enumeration of all
8      * quarters of our town
9      * price in an integer which represents the price that client will
10     * pay if there is a bic that drive him to his dest
11     */
12     pid_t pid;
13     quarter start;
14     quarter dest;
15     unsigned int price;
16     time_t wait_time;
17 } Client;
```

Listing 3: bike

```

1  /*
2  Definition of a bike type, which will references our processes:
3      one bi is a process
4      pid is the process id of the bike (as we said, it's a process)
5      itinerary is a table of quarters, element of an enumeration of
6      all quarters of our town
7
8  */typedef struct Bike
9  {
10     pid_t pid;
11     int *itinerary;
12 }Bike;
13 //type deque
14 typedef struct deque
15 {
16     struct block *start;
17     struct block *end;
18 }deque;

```

- The structure **quartier**: who does the correspondance between each quater and its Pid in the data structure
- The structure **deque** :A deque, or double-ended queue, is a type of queue in which insertion and removal of elements can be performed from both the front and the rear. This means it does not follow the FIFO (First In First Out) rule.A deque provides more flexibility by allowing operations at both ends of the queue.
- The strucure **Bike** : is one of the main structure of our program, the main ressource used in the program. This structure is defined as a type and its variables are :
 - It's **PID** which is a unique number attributed to each bike after it's creation
 - It's **Itinerary**: who defines the starting point of the bike and it's ending point.And we should note that an itinernary is a set of many destinations.
- The data structure **Quartier** : whose's role is to contain the name of all the quaters we will use to generate all the itinerary for our bike

4.2 RPG (Random Process Generator)

To implement this RPG, we will need enter an infinite loop, representing the total duration of the program. At random intervals, we will simultaneously create processes representing motorcycles and customers. These processes will be modified in their operation using the `exec()` function.

The RPG, as defined earlier, is a program in which processes are created, representing either customers or motorcycles. After the creation of these processes, they will be directed to execute a binary file containing specific instructions that they will carry out. Here is the C code of the RPG code:

Listing 4: code of RPG

```

1 #include <time.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <stdlib.h>
5 #include <sys/types.h>
6 #define MAX_SLEEP_TIME 2
7 #define CLIENT_GENERATION_FREQ 2
8
9 int main(int argc, char *argv[])
10 {
11     srand(time(NULL));
12     int balance = 0;
13     while (1)
14     {
15         pid_t pid = fork();
16         if (pid == -1)
17             fprintf(stderr, "Error, cannot start neither bike or client process\n");
18
19         if (pid == 0)
20         {
21             if (balance % 2)
22             {
23                 char* args[] = {"../client/bin/client.out", argv[1],
24                                 NULL};
25                 execvp("../client/bin/client.out", args);
26             }
27             else
28             {
29                 char* args[] = {"../moto/bin/bike.out", argv[1], NULL};
30                 execvp("../moto/bin/bike.out", args);
31             }
32
33             balance++;
34             sleep(5);
35         }
36
37         return 0;
38     }

```

This program continuously creates processes at random intervals less than a given time T, which is a compilation parameter. Subsequently, it modifies their instructions using the **execvp()** function.using the variable balance,it create clients and bikes.

4.3 Clients

To implement the client, let's describe how to carry out each task. The following provides an overview of how the client operates.

When creating a motorcycle, the following steps are taken:

1. **Information Generation:** Here, we use the **random()**; function to generate the client's starting and ending destinations. The PID is retrieved using the **getpid()**; function.

Listing 5: random and pid functions

```
1 \\random function()
2 #include <stdlib.h>
3 long int random(void);
4
5 \\getpid()
6 #include <sys/types.h>
7 #include <unistd.h>
8 pid_t getpid(void);
```

2. **Shared Memory Segment Creation:** We create using the **shmget()**; function and attach using the **shmat()**; function.

Listing 6: shmget and shmat functions

```
1 \\shmget function()
2 #include <sys/shm.h>
3 int shmget(key_t key, size_t size, int shmflg);
4
5 \\shmat()
6 #include <sys/shm.h>
7 void *shmat(int shmid, const void *shmaddr, int shmflg);
```

3. **Signal Transmission to Scheduler:** It then sends a signal to the scheduler informing the completion of the initial part execution, this using the **kill()**; function.

Listing 7: kill function

```
1 \\kill fuction()
2 #include <signal.h>
3 int kill(pid_t pid, int sig);
```

4. **Signal Waiting:** It waits for the scheduler's signal, which continues until it reaches its limit using the **sleep()**; function. If the allocated time is exceeded, the process is then terminated, a signal is sent to the scheduler, and it is removed from the list of clients using the **kill()**; and **SIGNAL(int,void*);** functions.

Listing 8: sleep and signal functions

```
1  \\sleep fuction()  
2  #include <unistd.h>  
3  unsigned int sleep(unsigned int seconds);  
4  \\SIGNAL()  
5  void (*signal(int sig, void (*func) (int))) (int);
```

5. **Signals Reception by the Client:** It receives a signal from the bike informing that it has been transported. At the end of the ride, it receives a signal from the bike indicating the end of the ride and terminates. This is done using the **SIGNAL** function, and it concludes with **exit(0)** if the client is dissatisfied and **exit()**; if satisfied.

4.4 Bike

The bike entity is defined by a data structure of **type def struct bike** which has its pid and itinerary as data. Its itinerary is itself an array formed of five consecutive elements from the list of neighborhoods chosen randomly.

To implement the Bike, we will follow the steps below:

1. **Bike Creation** :In this step we start by allocating memory space to create the bike using the **malloc();** function. Then, the bike is initialized from a start point randomly chosen from the list of neighborhoods and incrementing the list of neighborhoods until we have five neighborhoods. The pid of the bike is retrieved using the **getpid();**function. All this bike creation is done using a function called **Bike-Creation();** of type void and returns an element of type bike.
2. **Shared Memory Segment Creation** : In this segment, the bike creates a shared memory segment using the **shmget();** function and attaches it using the **shmat();** function. In this segment, the bike writes its itinerary using the **memcpy();** function.
3. **Signal Transmission to Scheduler**: The bike then sends a bike created signal to the scheduler using the **kill();** function.
4. **Signal Waiting** : After sending its signal, the bike waits for a while using the **sleep();** function. A handler will be in charge of transmitting the Central Scheduler signal when the time comes.
5. **Signal Transmission to Client** : Here, the bike reads the client's pid in the shared memory segment and sends a signal to the client to say "I have accepted you" using the **kill();** function. Then, the process starts over.

Listing 9: Bike creation

```

1 #include "../headers/defines.h"
2 #include "../headers/signal.h"
3 #include "../headers/initialisation_bikes.h"
4
5 char *quartier[111] = {"Centre_commercial", "Elig_Essono", "Etoa_MekiI", "
    Nlongkak", ... , "Minkoameyos", "Nkolso"
6 };
7 // Return a bike filled with his parameters
8 Bike *create_bike(){
9     Bike *bike = (Bike*)malloc(sizeof(Bike));
10
11     // if alloc fails
12     if(!bike) return NULL;
13
14     // Generating a itinerary filled with a random of 5 consecutives
        quarters
15     int start = random_integer(0, NB_QUARTER - RADIUS);
16     int* itinerary = (int*)malloc(sizeof(int) * RADIUS);
17
18     // Filling consecutives quarters
19     for (int i = 0; i < RADIUS; i++)
20         itinerary[i] = start++;
21
22     // Filling the new bike structure
23     bike->pid = getpid(); // Getting pid process
24     bike->itinerary = itinerary;
25
26     return bike;
27 }
28 void write_infos_bike(Bike *bike){
29     memcpy(shm_zone, bike, sizeof(Bike));
30 }

```

Listing 10: Shared memory segment creation

```

1 void create_segment(pid_t pid){
2     shm_id = shmget((key_t)pid, SEG_SIZE, IPC_CREAT | IPC_EXCL | S_IRUSR
        | S_IWUSR);
3     shm_zone = shmat(shm_id, NULL, 0);
4 }

```

Listing 11: signals (transmission to scheduler , waiting , transmission to client)

```

1
2 #include "../headers/defines.h"
3 #include "../headers/signal.h"
4 #include "../headers/initialisation_bikes.h"
5
6 // Generate a random integer between a and b included
7 int random_integer(int a, int b){
8     return (rand() % (b - a + 1)) + a;
9 }
10 void starting_course(pid_t client_pid){
11     kill(client_pid, STARTED_COURSE);
12 }
13 void ended_course(pid_t client_pid){
14     kill(client_pid, ENDED_COURSE);
15 }
16
17 // Receive a signal of client and read in the shm the pid of client and
18 // change a statut of moto
19 void bike_handler(int signum){
20     if(signum != SEGMENT_READY)
21         return;
22
23     pid_t client_pid = *((pid_t*)shm_zone);
24     // Send a signal to precise that we are carrying a client
25     starting_course(client_pid);
26     // Time interfal for the course
27     sleep(random_integer(TASK_TIME_MIN, TASK_TIME_MAX));
28     // Send a signal to precise that we have ended the course
29     ended_course(client_pid);
30     // Write again bike's infos
31     write_infos_bike(bike);
32     // Resend the signal to the CS
33     send_signal(oc_pid, BIKE_CREATED);
34 }
35 // Atttach functions to signal
36 void set_handler(){
37     signal(SEGMENT_READY, &bike_handler);
38 }
39 // Send signal to CS
40 void send_signal(pid_t oc_pid, int sig){
41     kill(oc_pid, sig);
42 }

```

4.5 Central Scheduler

To implement our scheduler, we are going to present step by step, the following ideas:

1. **Data initialization:** Here we start by creating two deques one for the clients and another one for the bikes which are present. And then arm the different signal handlers.

Listing 12: creation of deques

```

1 // client_deque and bike deque
2 /* @brief Create a deque object
3  * @return deque* deque created
4  */
5 deque *create_deque()
6 {
7     deque *new_deque = (deque *)malloc(sizeof(deque));
8     if (new_deque == NULL)
9     {
10         perror("Error, cannot create deque\n");
11         exit(EXIT_FAILURE);
12     }
13     new_deque->start = NULL;
14     new_deque->end = NULL;
15     return new_deque;
16 }
17 void init_deques()
18 {
19     bikes_deque = create_deque();
20     clients_deque = create_deque();
21 }
22 //type deque
23 typedef struct deque
24 {
25     struct block *start;
26     struct block *end;
27 } deque;

```

2. **Listening Mode:** The Scheduler now listens to various signals from bikes and clients for a set time t .
The different signals awaited are: **CLIENT-CREATED** and **BIKE-CREATED**. We perform the waiting using the **signal()** function. We use handlers to process these different signals with **sig-actions**
3. **Masking of signals:** After the said time, the signals sent as of now are not taken into consideration for the following scheduler operations.
4. **Saving Clients:** It saves the different clients into the deque saving it using the FIFO policy and same is done for the different bikes using the other deque.
5. **Assigning Clients to Bikes:** Using the first fit protocol, the various clients are assigned to the available bikes and for this to happen, a signal is sent (with the function **kill()**;) to the chosen bike and the chosen client is removed from client deque.
6. **Restart:** The procedure then restarts from the listening period and the masked items are added to the appropriate deques.

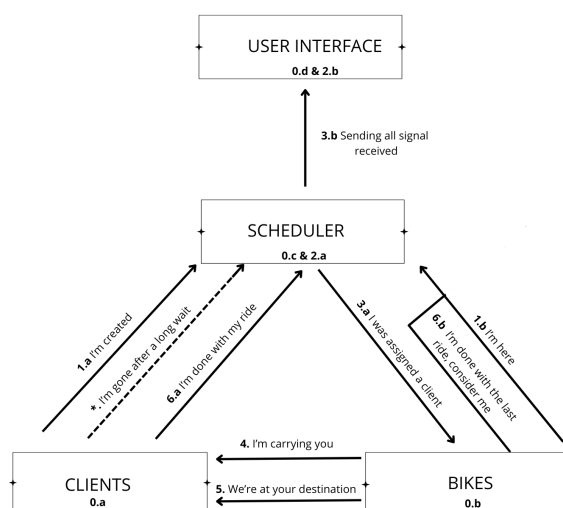
5 User Interface

To implement the interface of this project, we have opted to utilize the **OpenGL** library in the C programming language. This library proves to be highly advantageous for crafting interactive interfaces. In order to carry out this task systematically, we have chosen to break down the responsibilities into several key modules: the **Central Scheduler (CS)**, the **User Interface (UI)**, and the **Client** and **Bike** modules. This modular approach ensures effective management of each aspect of the interface, ensuring a robust design and smooth development.

Furthermore, we will facilitate interaction among these different modules through interprocess communication and the use of shared memory segments. In the following sections, we will provide a detailed explanation of the work accomplished in each module and illustrate the results.

The following figure gives a brief summary of all what has been explicated above. The numbered sections 0.a, 0.b, 0.c, 0.d, 2.a and 2.b which couldn't be further explained on the figure respectively represent the following:

1. **Step 0.a:** This represents the information generation and the shared memory segment creation.
2. **Step 0.b:** This represents the route and the shared memory segment creation.
3. **Step 0.c:** This part is concerned with the data initialization and listening mode from the scheduler
4. **Step 0.d:** This part is concerned with the initialization of the user interface
5. **Step 2.a:** This part is concerned with the different saving of the different bikes and clients information.
6. **Step 2.b:** This part is concerned with the the treatment of all the various signals by the user interface.



5.1 Client and bike

Graphical clients and motorcycles are simply defined using G-client and G-bike data structures described below.

Listing 13: structures for user interface

```

1 #include <unistd.h>
2
3 // 2D point structure
4 typedef struct Point
5 {
6     float x;
7     float y;
8 }Point;
9
10 //client's structure
11 typedef struct G_client
12 {
13     float radius;
14     Point* start;
15     Point* dest;
16     Point* center;
17 }G_client;
18
19 //bike's structure
20 typedef struct G_bike
21 {
22     float side;
23     Point* start;
24     Point* center;
25 }G_bike;
26
27
28 typedef struct G_quarters
29 {
30     int id;
31     Point *loc;
32 } G_quarters;
33
34
35 typedef struct G_data
36 {
37     pid_t bike_pid, client_pid;
38     int bike_start;
39     int client_start, client_dest;
40
41     struct G_data* next;
42 }G_data;

```

We also define G-quarters structures to describe the graphic area and G-data to link a motorcycle to a client it must go to. This structure will be useful to ensure the simultaneous movement of our motorcycle and client on the interface.

5.2 Scheduler

The central scheduler operates in this section regarding communication with the background of our project. In fact, it is responsible for informing the interface process about the pairs (*client*, *motorcycle*) that need to be moved. It writes this information into a shared memory segment obtained through the **shmget()** function and sends a signal to the UI using the **kill()** function. The UI will then read the information from the shared memory segment and use it to simultaneously move both the motorcycle and the client it carries. The information written includes:

1. the client's process ID (pid)
2. the motorcycle's process ID (pid)
3. the starting and destination points of the client

The UI will then use this data to carry out the movement.

5.3 UI

Just as in the scheduler, the UI needs a few data structures to store the information of the clients and motorcycles it needs to display.

- First, it maps each neighborhood from the backend, stored as an integer, to a graphical neighborhood defined by the G-quar structure mentioned above.
- Then, it needs a named list of pairs list-t (*client*, *moto*) during display.

The UI also needs certain functions to perform its task:

The function **init-list()** creates the couple's list (*client*, *moto*) for data storage.

The function **init-quarters-list()** is in charge of creating a correspondence between the quarters back-end and the quarters on the interface and then store it in the quarter-list data structure.

The function **get-quarter-pos** which allows obtaining the graphical position of a neighborhood based solely on its PID. In fact, it will search the neighborhood list to accomplish this. It returns a graphical point.

A handler **handle-oc-scheduled** which is used after each scheduling by the scheduler. It allows the UI to read the shared memory segment with the scheduler and retrieve the necessary information to move a pair. (*client*, *moto*)

The UI also requires specific functions to draw the clients and motorcycles, which we do not consider necessary to list here. Just note that the motorcycles will be represented as squares, while the clients will be represented as circles on the screen. Here are the function signatures for these functions:

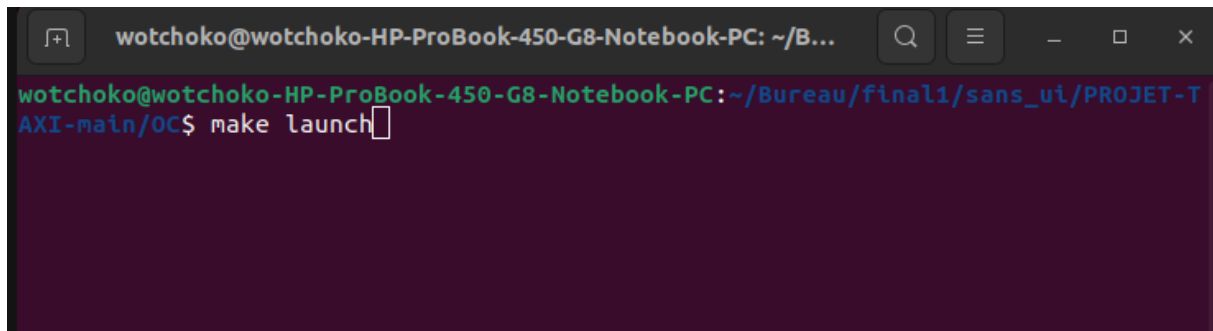
Listing 14: functions of UI

```
1 /**
2  * @brief initialize bikes and clients tuple list
3  *
4  */
5 void init_list(void)
6 {
7     bikes_n_clients_list = create_list();
8 }
9 /**
10 * @brief Function to create many points for display and store them in a
11 *    list_quarter.
12 *
13 */
14 void init_quaters_list()
15 /**
16 * @brief Function to find the coordinates of a quarter based on its ID.
17 * @param id(int)
18 * @return point
19 */
20 Point* get_quater_pos(int id)
21 /**
22 * @brief handle is used to gather information about couple client-bike
23 *    which now moves together.
24 * @param signum(int) is a number of signals used to call this handler
25 * @param *info(siginfo_t) helps to gather information for the the
26 *    handle
27 * @param *context is of type void
28 */
29 void handle_oc_scheduled(int signum, siginfo_t *info, void *context)
```

6 Test and evaluation

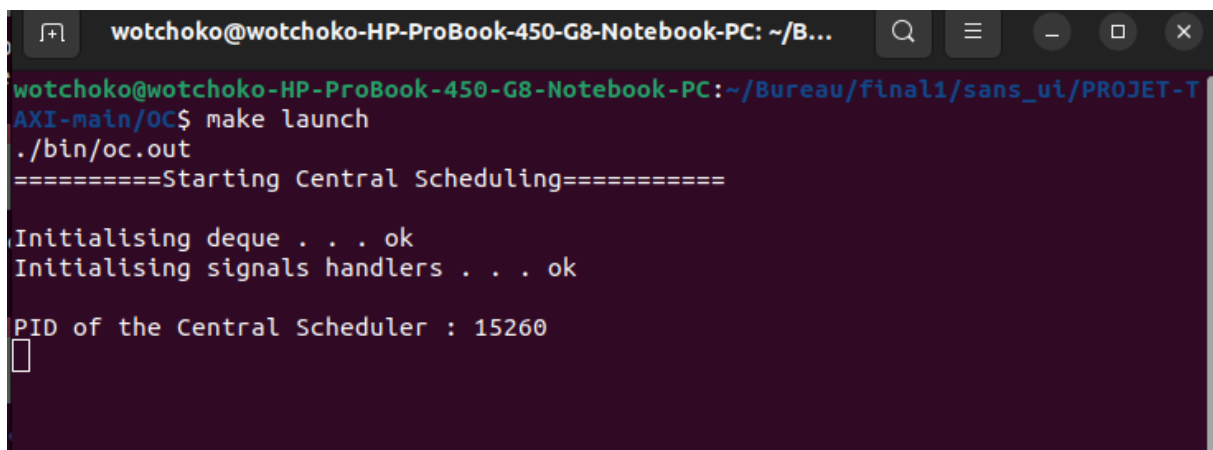
6.1 without interface

6.1.1 scheduler

A terminal window with a dark background and light-colored text. The window title is 'wotchoko@wotchoko-HP-ProBook-450-G8-Notebook-PC: ~/B...'. The prompt is 'wotchoko@wotchoko-HP-ProBook-450-G8-Notebook-PC:~/Bureau/final1/sans_ui/PROJET-TAXI-main/OC\$'. The command 'make launch' has been entered, and the cursor is at the end of the line.

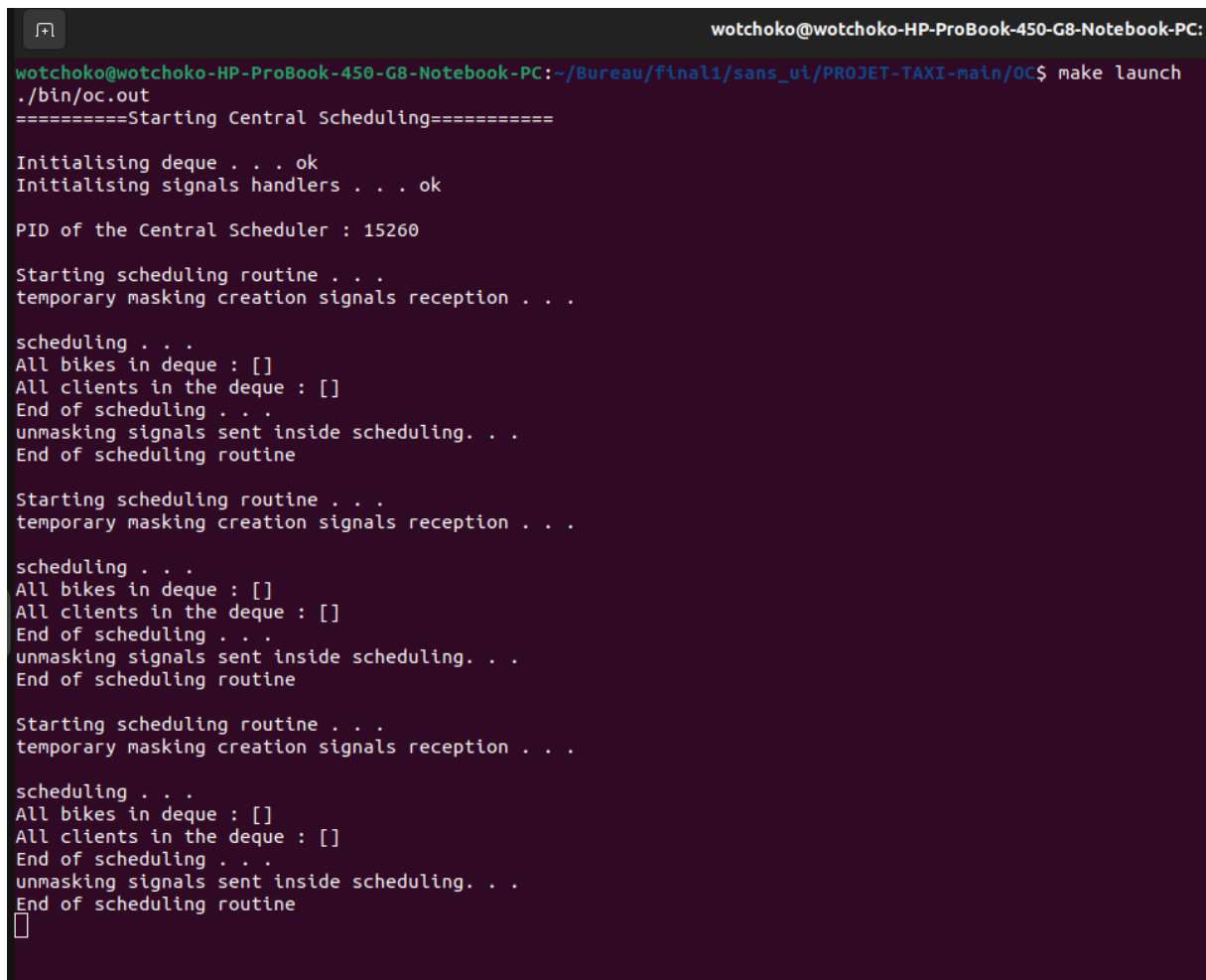
```
wotchoko@wotchoko-HP-ProBook-450-G8-Notebook-PC: ~/B...
wotchoko@wotchoko-HP-ProBook-450-G8-Notebook-PC:~/Bureau/final1/sans_ui/PROJET-TAXI-main/OC$ make launch
```

Figure 1: compiling scheduler only

A terminal window with a dark background and light-colored text. The window title is 'wotchoko@wotchoko-HP-ProBook-450-G8-Notebook-PC: ~/B...'. The prompt is 'wotchoko@wotchoko-HP-ProBook-450-G8-Notebook-PC:~/Bureau/final1/sans_ui/PROJET-TAXI-main/OC\$'. The command 'make launch' has been executed, and the output is displayed. The output includes the path to the executable, a separator line, and initialization messages for the deque and signals handlers, followed by the PID of the Central Scheduler.

```
wotchoko@wotchoko-HP-ProBook-450-G8-Notebook-PC:~/Bureau/final1/sans_ui/PROJET-TAXI-main/OC$ make launch
./bin/oc.out
=====Starting Central Scheduling=====
Initialising deque . . . ok
Initialising signals handlers . . . ok
PID of the Central Scheduler : 15260
```

Figure 2: result



```
wotchoko@wotchoko-HP-ProBook-450-G8-Notebook-PC:~/Bureau/final1/sans_ui/PROJET-TAXI-main/OC$ make launch
./bin/oc.out
=====Starting Central Scheduling=====

Initialising deque . . . ok
Initialising signals handlers . . . ok

PID of the Central Scheduler : 15260

Starting scheduling routine . . .
temporary masking creation signals reception . . .

scheduling . . .
All bikes in deque : []
All clients in the deque : []
End of scheduling . . .
unmasking signals sent inside scheduling. . .
End of scheduling routine

Starting scheduling routine . . .
temporary masking creation signals reception . . .

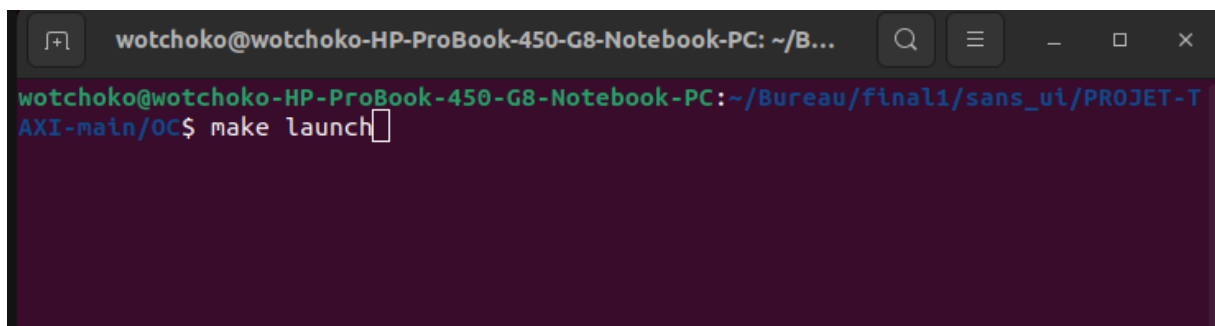
scheduling . . .
All bikes in deque : []
All clients in the deque : []
End of scheduling . . .
unmasking signals sent inside scheduling. . .
End of scheduling routine

Starting scheduling routine . . .
temporary masking creation signals reception . . .

scheduling . . .
All bikes in deque : []
All clients in the deque : []
End of scheduling . . .
unmasking signals sent inside scheduling. . .
End of scheduling routine
□
```

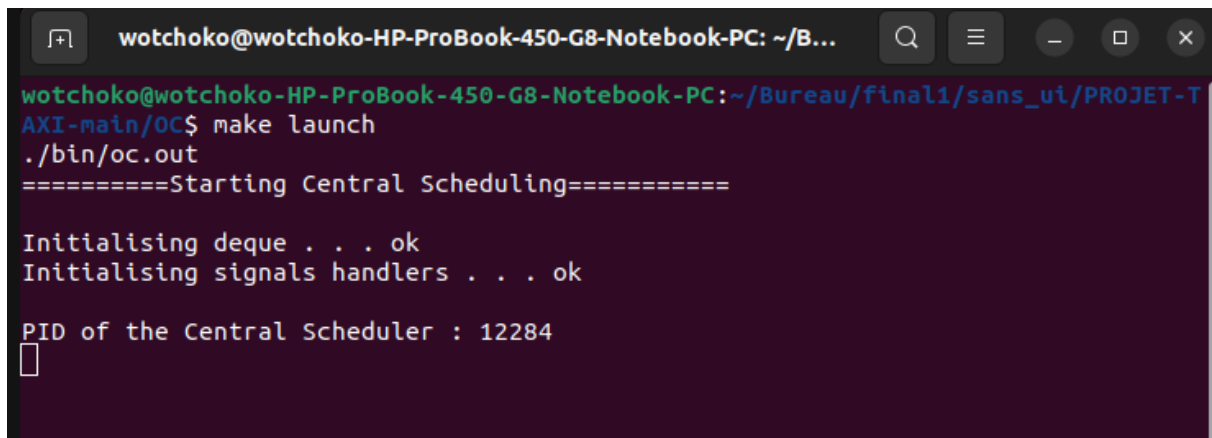
Figure 3: we see that deques are empty because clients and bikes are not created. Now we are going to compile it with RPG

6.1.2 scheduler with RPG



```
wotchoko@wotchoko-HP-ProBook-450-G8-Notebook-PC: ~/B...
wotchoko@wotchoko-HP-ProBook-450-G8-Notebook-PC:~/Bureau/final1/sans_ui/PROJET-TAXI-main/OC$ make launch□
```

Figure 4: to compile OC again

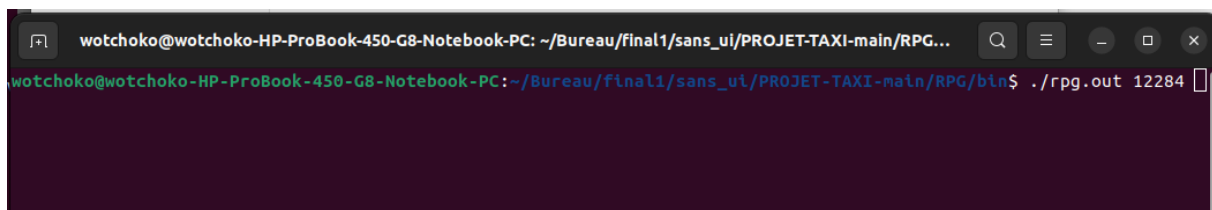


```
wotchoko@wotchoko-HP-ProBook-450-G8-Notebook-PC: ~/B...
wotchoko@wotchoko-HP-ProBook-450-G8-Notebook-PC:~/Bureau/final1/sans_ui/PROJET-TAXI-main/OC$ make launch
./bin/oc.out
=====Starting Central Scheduling=====

Initialising deque . . . ok
Initialising signals handlers . . . ok

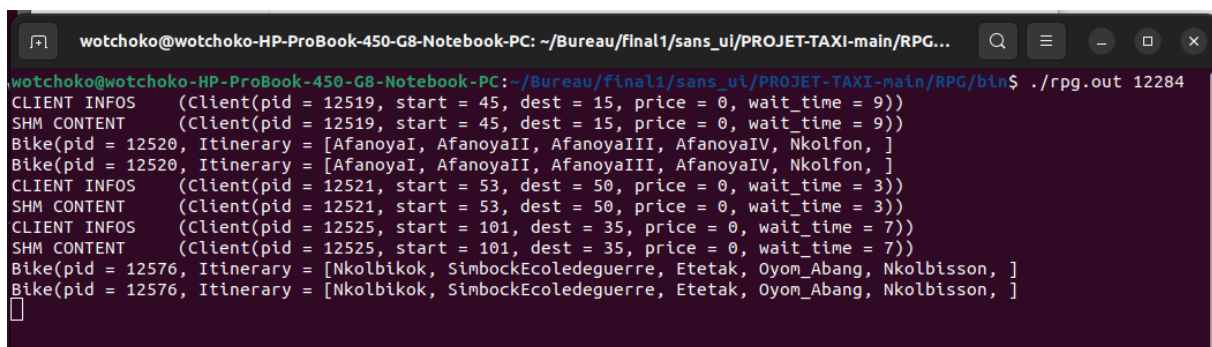
PID of the Central Scheduler : 12284
█
```

Figure 5: new result before RPG



```
wotchoko@wotchoko-HP-ProBook-450-G8-Notebook-PC: ~/Bureau/final1/sans_ui/PROJET-TAXI-main/RPG...
wotchoko@wotchoko-HP-ProBook-450-G8-Notebook-PC:~/Bureau/final1/sans_ui/PROJET-TAXI-main/RPG/bin$ ./rpg.out 12284 █
```

Figure 6: to compile RPG with pid of scheduler



```
wotchoko@wotchoko-HP-ProBook-450-G8-Notebook-PC: ~/Bureau/final1/sans_ui/PROJET-TAXI-main/RPG...
wotchoko@wotchoko-HP-ProBook-450-G8-Notebook-PC:~/Bureau/final1/sans_ui/PROJET-TAXI-main/RPG/bin$ ./rpg.out 12284
CLIENT INFOS (Client(pid = 12519, start = 45, dest = 15, price = 0, wait_time = 9))
SHM CONTENT (Client(pid = 12519, start = 45, dest = 15, price = 0, wait_time = 9))
Bike(pid = 12520, Itinerary = [AfanoyaI, AfanoyaII, AfanoyaIII, AfanoyaIV, Nkolfon, ])
Bike(pid = 12520, Itinerary = [AfanoyaI, AfanoyaII, AfanoyaIII, AfanoyaIV, Nkolfon, ])
CLIENT INFOS (Client(pid = 12521, start = 53, dest = 50, price = 0, wait_time = 3))
SHM CONTENT (Client(pid = 12521, start = 53, dest = 50, price = 0, wait_time = 3))
CLIENT INFOS (Client(pid = 12525, start = 101, dest = 35, price = 0, wait_time = 7))
SHM CONTENT (Client(pid = 12525, start = 101, dest = 35, price = 0, wait_time = 7))
Bike(pid = 12576, Itinerary = [Nkolbikok, SimbockEcoledeguerre, Etetak, Oyom_Abang, Nkolbisson, ])
Bike(pid = 12576, Itinerary = [Nkolbikok, SimbockEcoledeguerre, Etetak, Oyom_Abang, Nkolbisson, ])
█
```

Figure 7: result in RPG

```
End of scheduling routine

Starting scheduling routine . . .
temporary masking creation signals reception . . .

scheduling . . .
All bikes in deque : []
All clients in the deque : []
End of scheduling . . .
unmasking signals sent inside scheduling. . .
End of scheduling routine

Starting scheduling routine . . .
temporary masking creation signals reception . . .

scheduling . . .
All bikes in deque : []
All clients in the deque : []
End of scheduling . . .
unmasking signals sent inside scheduling. . .
End of scheduling routine
process with pid %d has been inserted in the bikes list
process with pid %d has been inserted in the bikes list
process with pid %d has been inserted in the bikes list
process with pid %d has been inserted in the clients list
process with pid %d has been inserted in the clients list
process with pid %d has been inserted in the clients list
process with pid %d has been inserted in the clients list
process with pid %d has been inserted in the bikes list
process with pid %d has been inserted in the bikes list

Starting scheduling routine . . .
temporary masking creation signals reception . . .

scheduling . . .
All bikes in deque : [15869,15870,15876,15927,15930,]
All clients in the deque : [15923,15924,15925,15926,]
█
```

Figure 8: scheduling

6.2 with interface

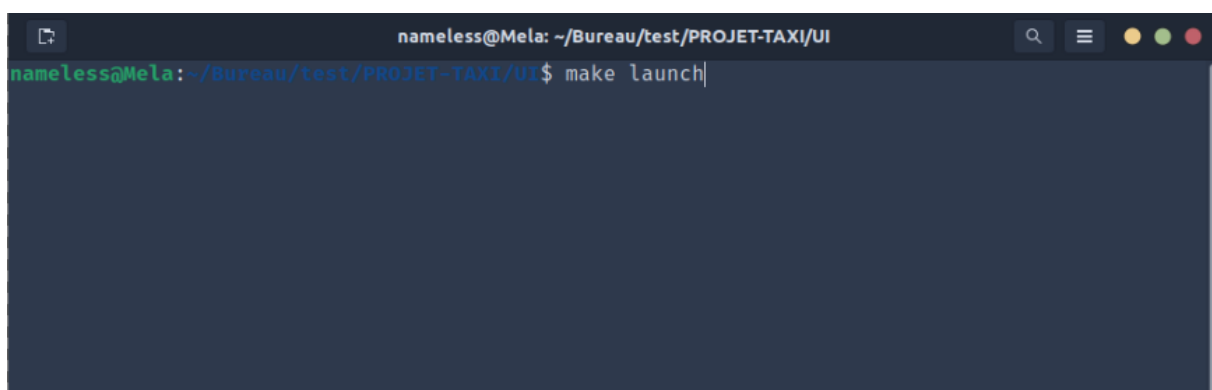


Figure 9: compiling of UI

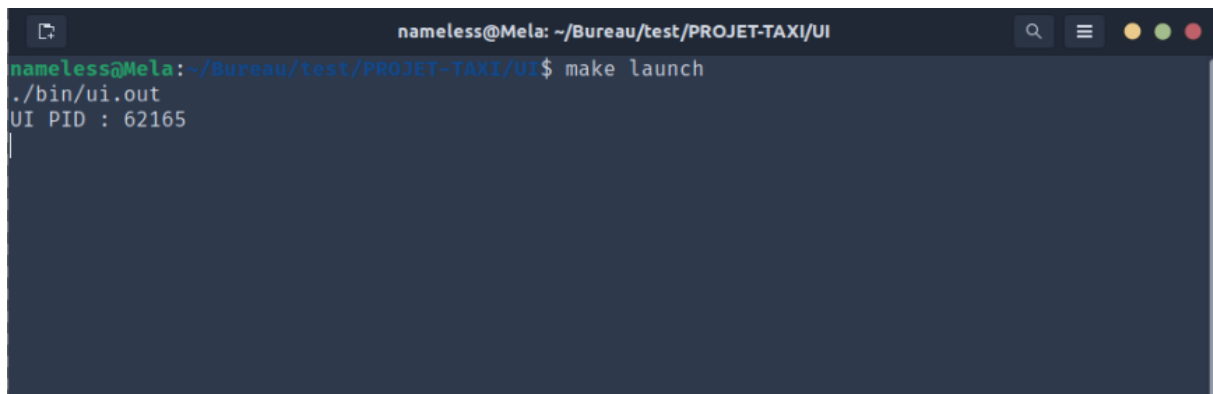


Figure 10: result of compiling UI in terminal

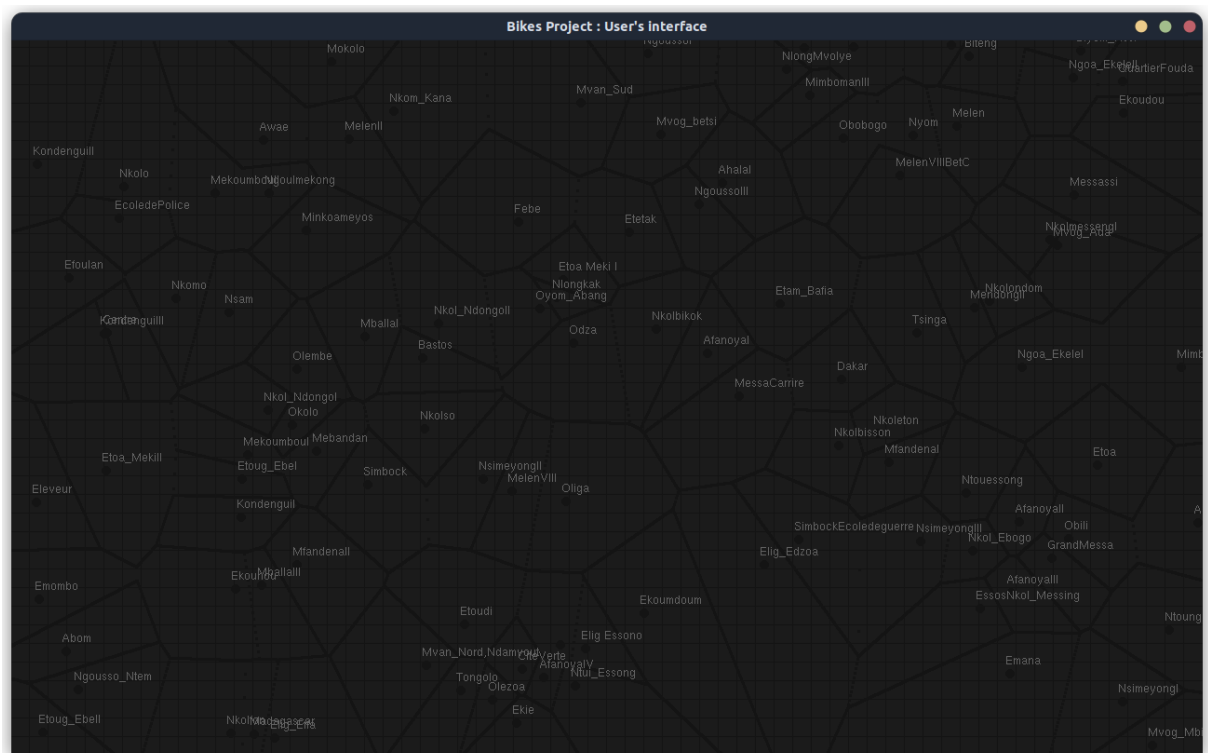


Figure 11: result of compiling UI

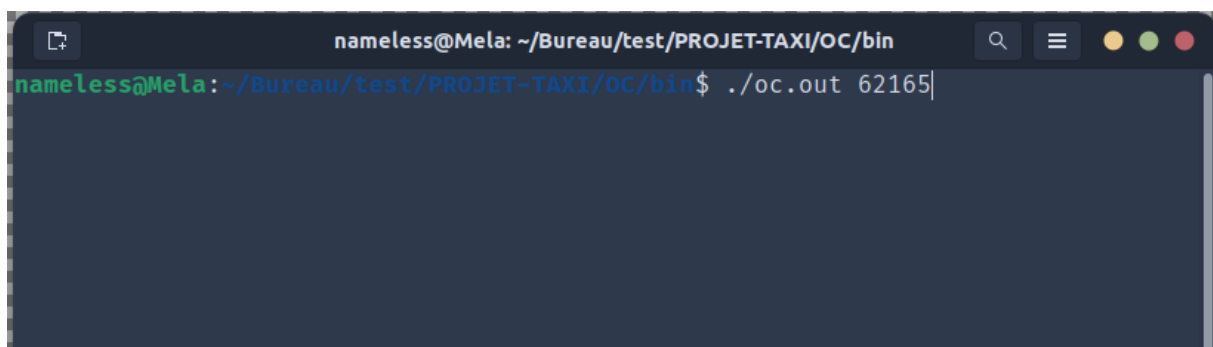
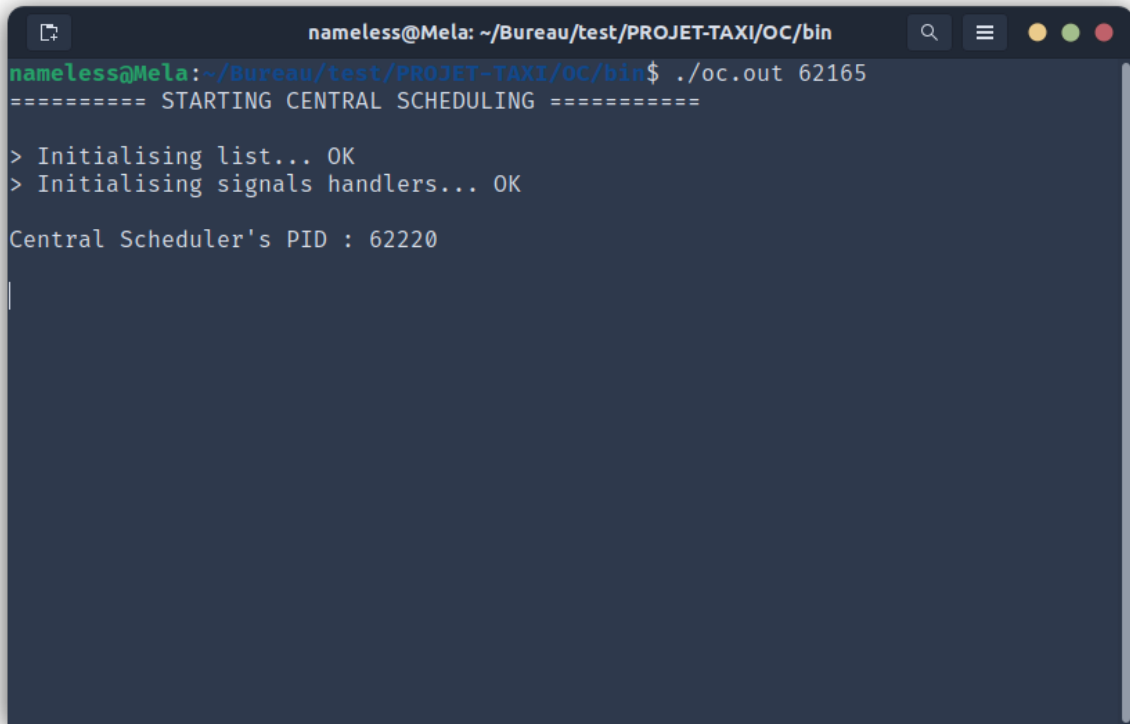
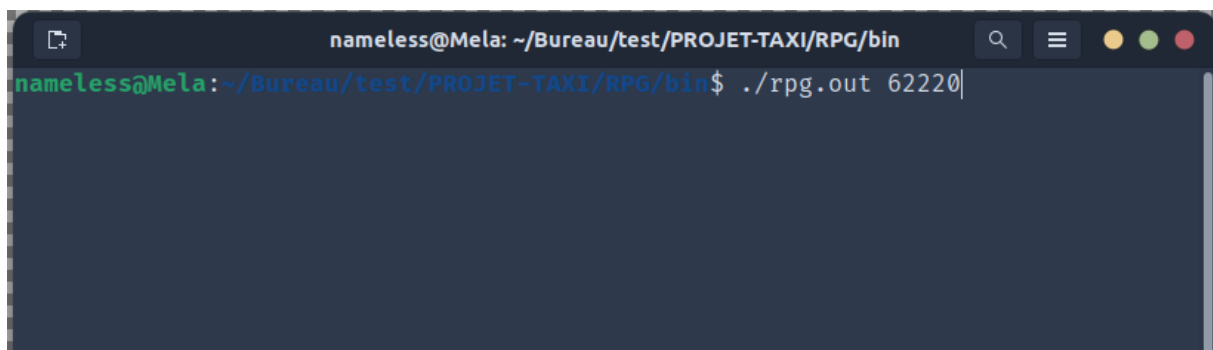


Figure 12: compiling OC with pid of UI



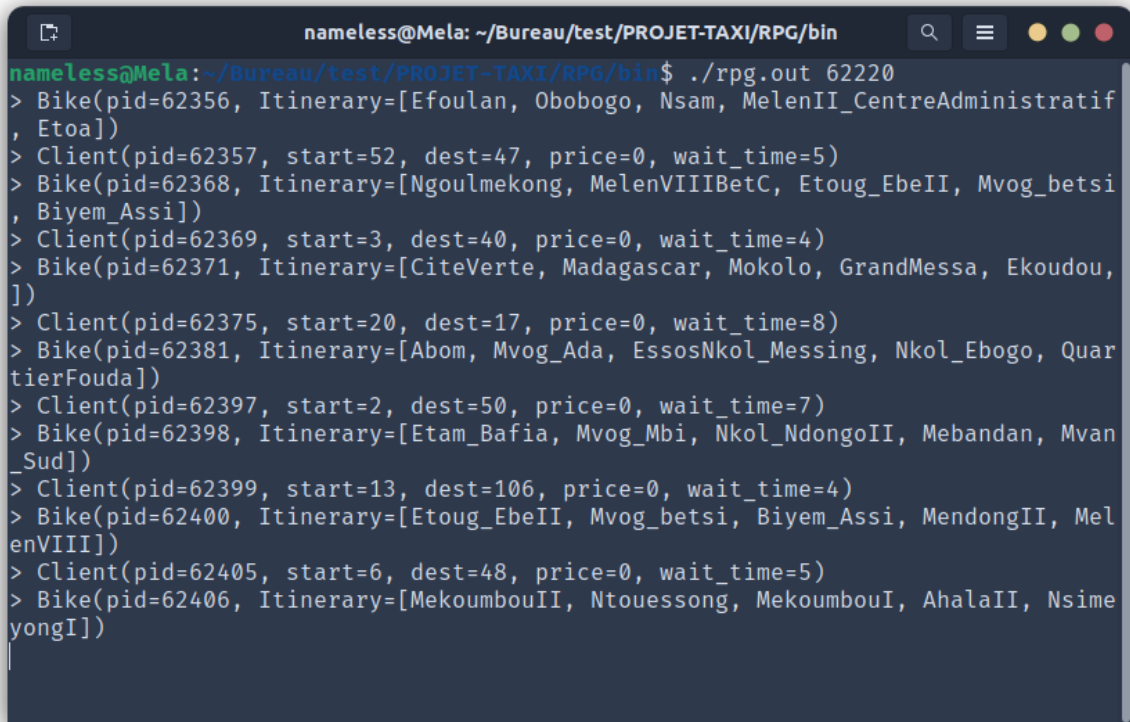
```
nameless@Mela: ~/Bureau/test/PROJET-TAXI/OC/bin
nameless@Mela:~/Bureau/test/PROJET-TAXI/OC/bin$ ./oc.out 62165
===== STARTING CENTRAL SCHEDULING =====
> Initialising list... OK
> Initialising signals handlers... OK
Central Scheduler's PID : 62220
```

Figure 13: result before the compilation of RPG



```
nameless@Mela: ~/Bureau/test/PROJET-TAXI/RPG/bin
nameless@Mela:~/Bureau/test/PROJET-TAXI/RPG/bin$ ./rpg.out 62220
```

Figure 14: compiling RPG with pid of OC

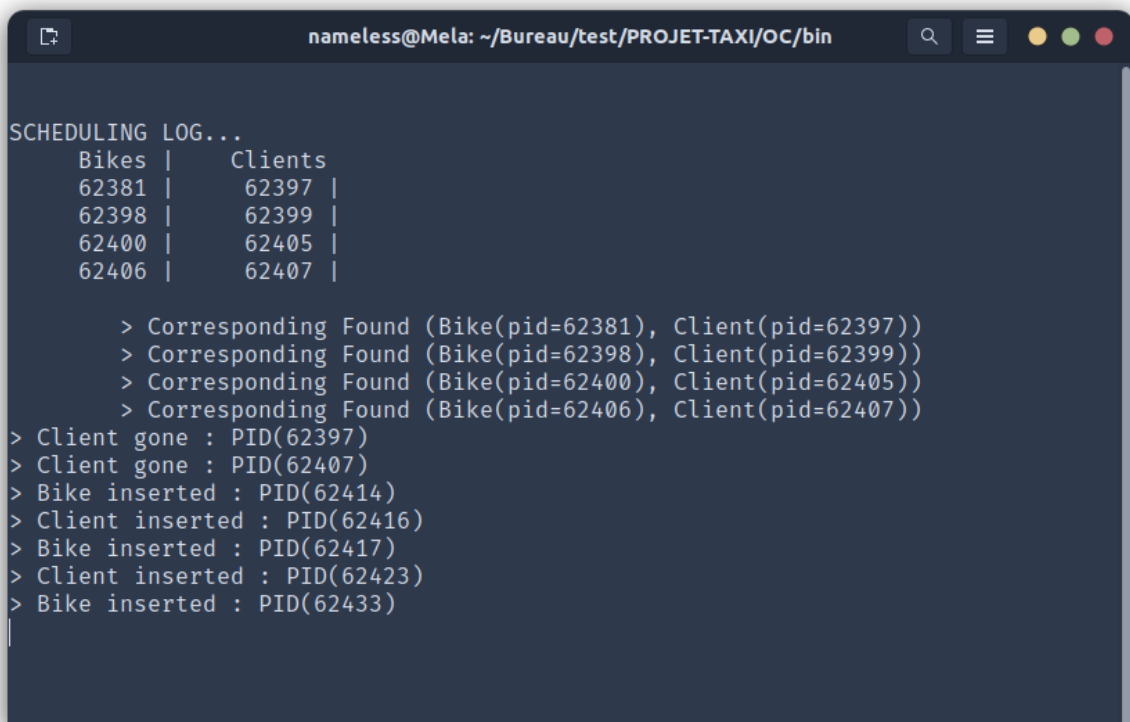


```

nameless@Mela: ~/Bureau/test/PROJET-TAXI/RPG/bin
nameless@Mela:~/Bureau/test/PROJET-TAXI/RPG/bin$ ./rpg.out 62220
> Bike(pid=62356, Itinerary=[Efoulan, Obobogo, Nsam, MelenII_CentreAdministratif, Etoa])
> Client(pid=62357, start=52, dest=47, price=0, wait_time=5)
> Bike(pid=62368, Itinerary=[Ngoulmekong, MelenVIIIBetC, Etoug_EbeII, Mvog_betsi, Biyem_Assi])
> Client(pid=62369, start=3, dest=40, price=0, wait_time=4)
> Bike(pid=62371, Itinerary=[CiteVerte, Madagascar, Mokolo, GrandMessa, Ekoudou, ])
> Client(pid=62375, start=20, dest=17, price=0, wait_time=8)
> Bike(pid=62381, Itinerary=[Abom, Mvog_Ada, EssosNkol_Messing, Nkol_Ebogo, QuartierFouda])
> Client(pid=62397, start=2, dest=50, price=0, wait_time=7)
> Bike(pid=62398, Itinerary=[Etam_Bafia, Mvog_Mbi, Nkol_NdongoII, Mebandan, Mvan_Sud])
> Client(pid=62399, start=13, dest=106, price=0, wait_time=4)
> Bike(pid=62400, Itinerary=[Etoug_EbeII, Mvog_betsi, Biyem_Assi, MendongII, MelenVIII])
> Client(pid=62405, start=6, dest=48, price=0, wait_time=5)
> Bike(pid=62406, Itinerary=[MekoumbouII, Ntouessong, MekoumbouI, AhalaII, NsimeyongI])

```

Figure 15: RPG result



```

nameless@Mela: ~/Bureau/test/PROJET-TAXI/OC/bin
SCHEDULING LOG...
  Bikes | Clients
  62381 | 62397 |
  62398 | 62399 |
  62400 | 62405 |
  62406 | 62407 |

  > Corresponding Found (Bike(pid=62381), Client(pid=62397))
  > Corresponding Found (Bike(pid=62398), Client(pid=62399))
  > Corresponding Found (Bike(pid=62400), Client(pid=62405))
  > Corresponding Found (Bike(pid=62406), Client(pid=62407))
> Client gone : PID(62397)
> Client gone : PID(62407)
> Bike inserted : PID(62414)
> Client inserted : PID(62416)
> Bike inserted : PID(62417)
> Client inserted : PID(62423)
> Bike inserted : PID(62433)

```

Figure 16: OC result

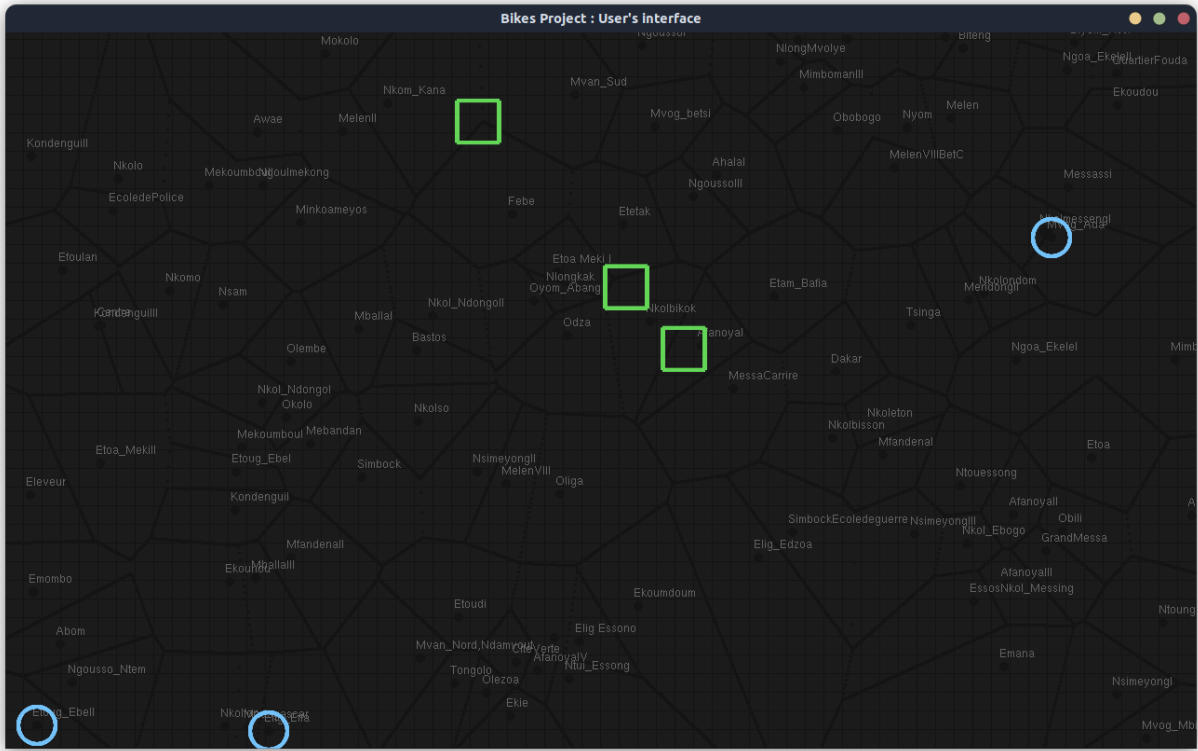


Figure 17: interface result

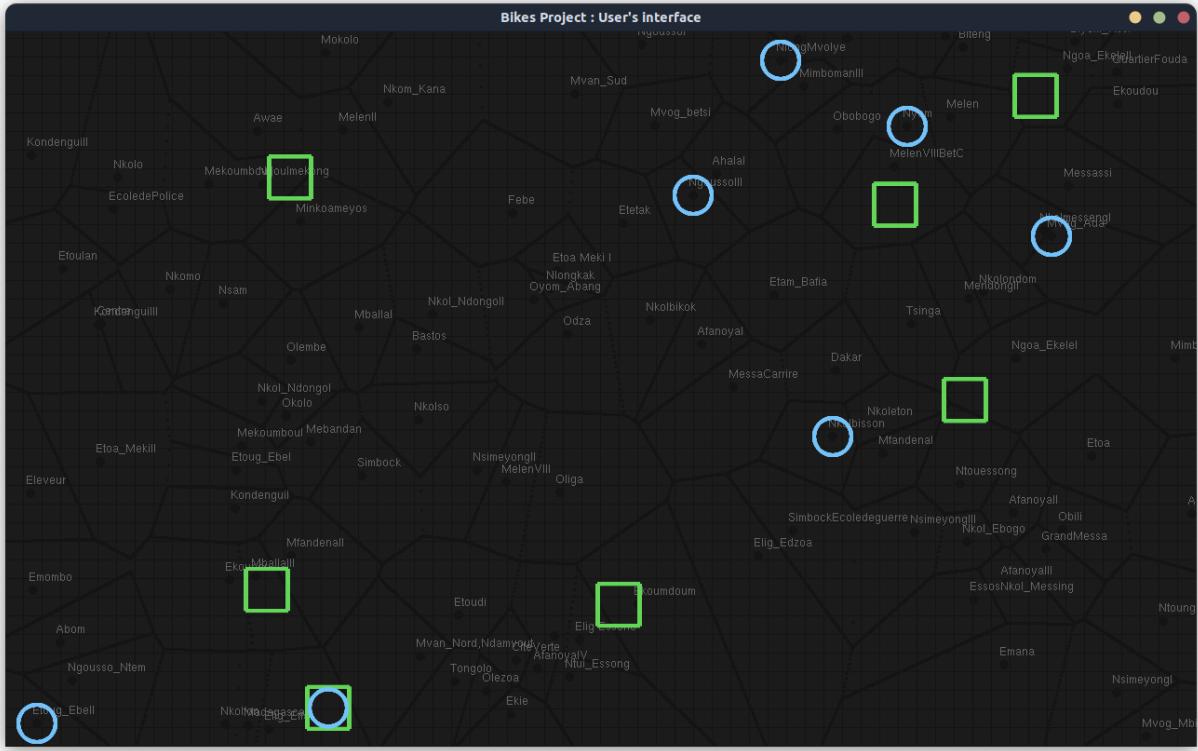


Figure 18: interface result

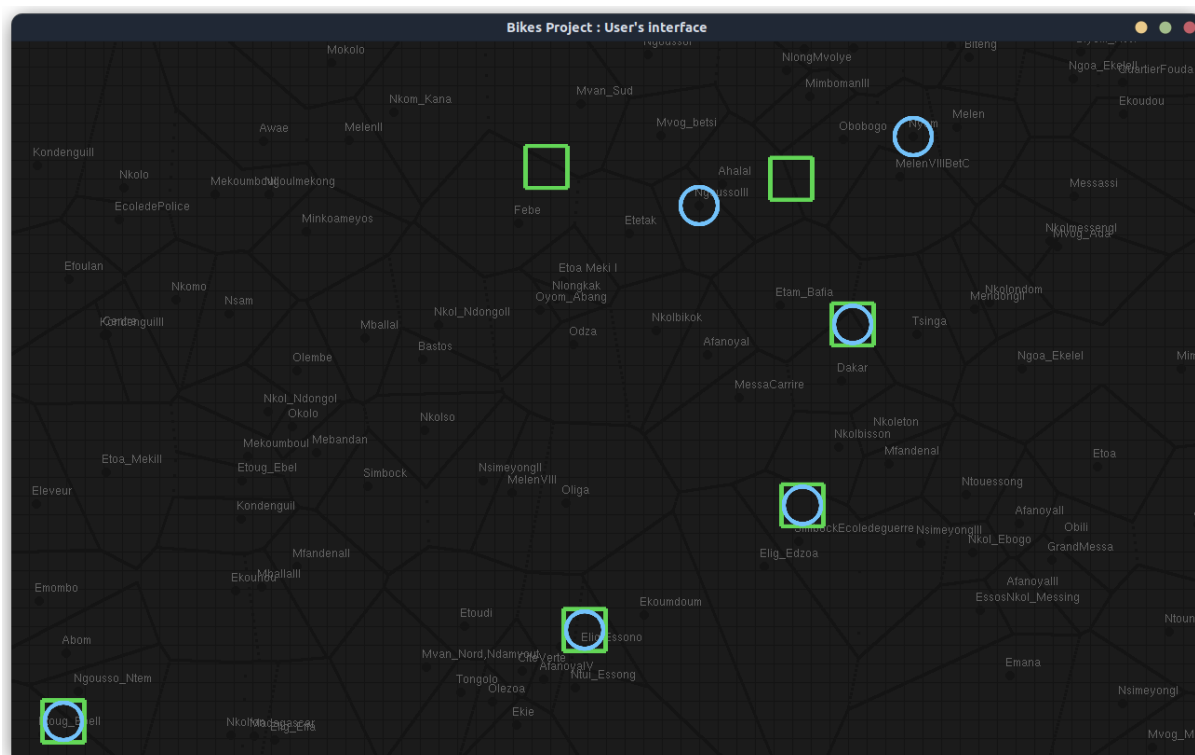


Figure 19: interface result

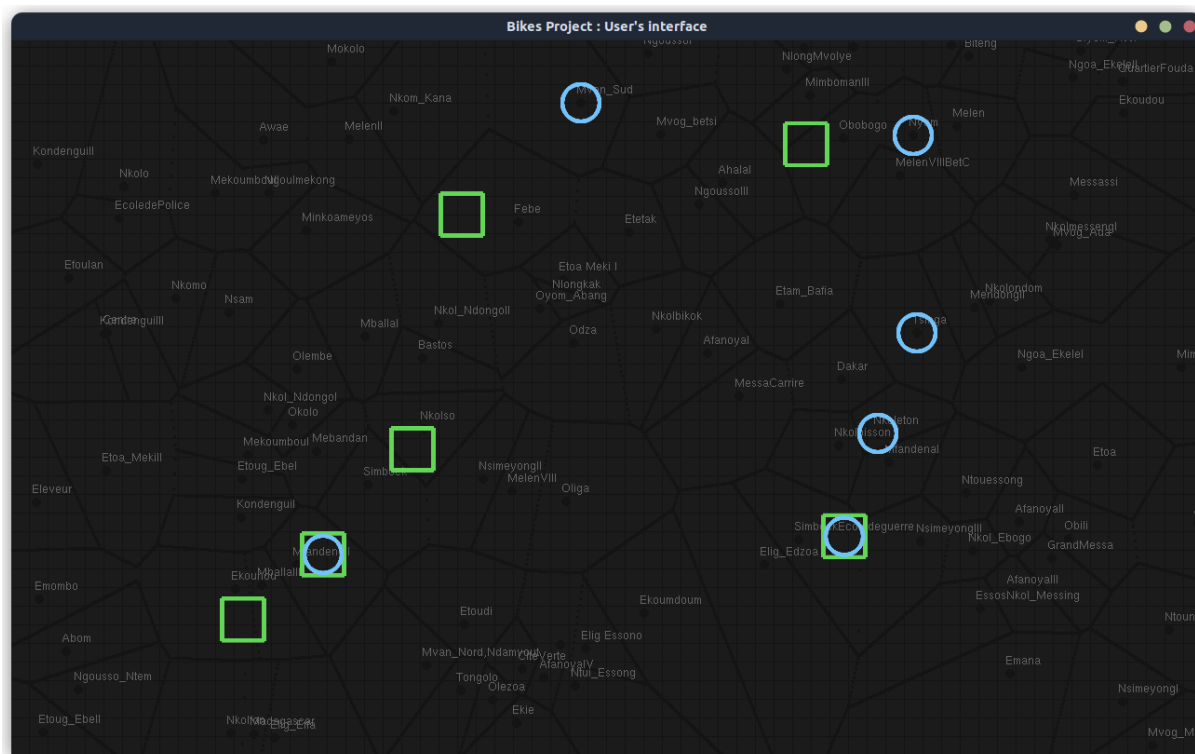


Figure 20: interface result

7 Conclusion

In conclusion, the motorcycle project has addressed a significant challenge in the transportation domain, particularly for motorcycle taxis: the efficient pickup of clients. We recognized the importance of ensuring customer satisfaction by providing fast and reliable transport, and we have taken measures to achieve this.

We identified challenges associated with client pickups, including managing various parameters such as client locations and motorcycle availability. To address these challenges, we proposed developing a client pickup algorithm that adheres to a certain policy while simulating the operation of a motorcycle in real life.

Throughout this project, we posed crucial questions that guided our work and helped us define an appropriate policy and mode of operation for our motorcycle.

Ultimately, we believe that this project has the potential to transform the transportation sector. By offering fast and reliable service, we can not only meet the needs of our clients but also set new standards in the industry. We look forward to seeing how this project will evolve and contribute to enhancing the transportation experience for our clients. Thank you for accompanying us on this exciting journey.