**Project Subject**

For this project, we aim to place “communication objects” in a wide area: the field. These communicating objects are either servers or mobile clients (drones).

We consider that servers are linked by a wired network which connects each server to its neighbors. Two servers are neighbors if they reach a common border.

We have also drones which are linked to the closest server by a wireless network. They are mobile, autonomous, they detect close obstacles.

**Our Approach**

**Classes Used**

Field

FieldDraw (extends GlutWindow class)

Server

Drone

Triangle

Polygon

Vector2D

**Algorithms Used**

Simple triangulation

Graham’s scan

Delaunay Triangulation

Derive Voronoi Diagram from Delaunay Mesh

**Organization**

Kwabena BAMFO – assigned to drawing the field

Patrice GADEGBE – assigned to physics of drone movements and collision detection

Rabah GILES – assigned to dynamic distribution of drones in the field

Zihan SHEN – assigned to ensure clean merging of code on GitHub

**Planning**

We developed with JetBrains CLion, because of its convenience when programming in C++. For version control, we made use of GitHub. This also allowed us to work independently and bring our code together at the end rather than waiting for one implementation to be completed before moving to another. We also used Google Doc for breaking down the project subject into smaller units that can be shared and assessing the difficulty of all the unit tasks of the project subject.

**Our Solution**

**Server Part**

First, we compute the convex hull of a given set of severs using Graham’s scan algorithm.

A picture containing sky, skiing, outdoor, snow

Description automatically generated

We then triangulate with this convex hull.

A close up of an umbrella

Description automatically generated

After this, we add the interior points, and form a new triangulation with the interior points.

A picture containing accessory, umbrella

Description automatically generated

After this normal triangulation, we apply the Delaunay triangulation algorithm on our list of triangles.

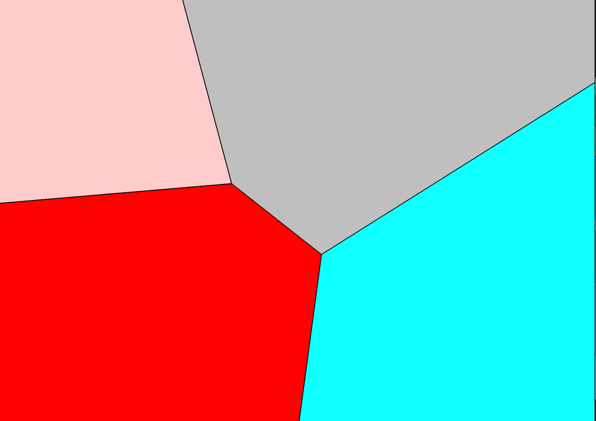
A picture containing accessory, envelope, stationary

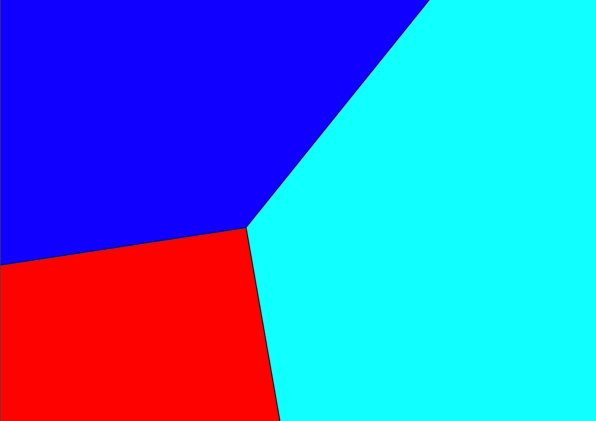
Description automatically generated

Given our Delaunay triangles, we draw the Voronoi diagram using the algorithm proposed in Figure A.

A picture containing accessory

Description automatically generated



****

**Figure A**

A screenshot of text

Description automatically generated

**Drone Part**

The drones in autonomous movement and move on the map and through the zones of each city independently. Each drone has dimensions which are specified at the level of its manufacturer which is represented by the attribute: “radius”.

each drone has a function to update its destination server. The drone continuously checks the server area in which it is located and automatically connects to the server in this area. once connected the server sends it its next destination using the onboard function in each drone.

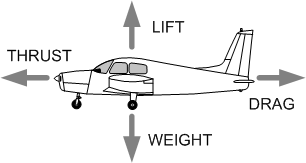
another time a drone knows its destination it generates a propelling force which is directed in the direction of its destination. This propulsion power depends on the distance from its destination where it can be maximized in order to allow the drones to reach a maximum speed and gradually reduce the time to reach the area of ​​the destination server.

This propelling force is converted into an acceleration vector using Newton's second law of motion

A close up of a logo

Description automatically generated

where F is the sum of all the external forces applied to the drone.



**Lift**

Lift is the force operating perpendicular to the relative airflow. In normal level flight Lift workings against Weight/Gravity

**Weight**

Weight is the force of gravity equal to the total mass of the drone

**Drag**

As an object moves through the air, the air resists the motion and this force is known as Drag. Drag is along and opposed to the direction of flight.

**Thrust**

Thrust opposes drag. The engine creates thrust and moves the plane forward.

a is the acceleration to generate.

m is the actual mass of the drone.

In order to limit worries and maximize the speed we use a damping coefficient k <1:

A close up of a clock

Description automatically generated

If there is an acceleration here is the evolution of the speed:

the speed is maximized a: *vmax* = *ka*/ (1 - *k*)

A screenshot of a cell phone

Description automatically generated

If there is no more force, no more acceleration, here is the evolution of the speed which descends towards 0:

vmax = 0

A close up of a mans face

Description automatically generated

location = new Vector2D (50, 50);

speed = new Vector2D ();

acceleration = new Vector2D ();

start = new Vector2D ();

server = new Server ("", new Vector2D (), "");

// average weight of a drone in kg

weight = 0.8;

// drone size radius

radius = 20;

// legal max speed of 100 mph

maxSpeed ​​= 44.704;

// distance of collision activation

dmax = 0 + 96;

// distance of max collision force activation

r = radius + 48;

// the thrust force strength

thrustForceStrength = 0.5;

// the collison max force

collisionForceStrength = 1;

// smooth damping

smoothDamping = 0.9;

In the event of a collision, the drone has generated a collision force which is added to the propulsion force of the drone in order to obtain a new speed vector.

A screenshot of a cell phone

Description automatically generated

As shown in the figure below

A close up of a map

Description automatically generated

Vector2D\* acceleration; : The acceleration is a Vector2D and is evaluate according to the total force applied on the drone

Vector2D\* speed : The speed is a Vector2D and is evaluate according to the acceleration

Vector2D\* location: the drone send his location over an interface

Server\* server: a drone receive the target server in order to evaluate the direction of his thrust force

Vector2D\* start: if a drone change direction or recieve a new target server, he save the last location where he change direction

**Challenges**

* Adding corner points: In some cases, it was necessary to add corner border points, which was a bit tricky. For example, if Qi(x) = 0 and Qi+1(x) = width of field.
* Collision detection: Finding the right force to apply for smooth collision detection was a bit of a challenge
* Different Operating Systems: We had challenges with CMake Files, since it was not added to git ignore.
* Sequential distribution: We faced difficulties in merging our code that distributes drones neighbour by neighbour with the rest of our code

**Solutions**

* To solve the problem with corner points, we tested out the scenarios where we need to add two border points to find out how (in what order) to add the points. We still envisage that there may be scenarios in which there is still a problem with attaining corner points
* To solve the different OS problem, we each kept a local CMake and used that for compiling and running whenever we pulled from origin

**Existing Issues (Limits of our Solution)**

* Voronoi diagram is buggy for some combination of server locations, particularly because of minor issues with finding and adding the right corner points.
* Collision detection force looks smooth for some server areas and cumbersome for others.
* Distribution of drones in the field not optimized as per the project subject (that is, servers can only reach neighbours). Our implementation is close enough, but not ideal. We however have an ideal solution separate from the master branch, as demonstrated during our presentation.