
I32 - CR Projet Naval World

CHAREYRE Adam

Table des matières

Introduction	2
Base de données	3
Mise en place d'un environnement de travail	3
Les tables	3
Triggers	7
Application	9
Introduction	9
Pré-requis	9
Menu Principal	10
Bouton 'View Country'	11
Bouton 'Add new country'	13
Bouton 'Create contract'	14
Bouton 'View contracts'	18
Programmation de l'application	21
Introduction	21
cProfile, pstats et snakeviz	21
Polygons	21
Connection à la DDB	24
Intermédiaire entre la base de donnée et l'application	24
Bouton tkinter custom	24
Scrollbar	24
Utils maths	24
PathFinding	26
Fonctions prix en fonction de l'affection	26
Intéractions entre les frames et la DDB	27
CountrySelectionFrame	27
PresentCountryWindows	27
CountryAddingFrame	28
CountryMakeContractFrame	29
ViewContractsFrame	31
BoatViewRealTimeFrame	34
WalkerShip	35
Conclusion	36

Introduction

Vous trouverez dans ce compte rendu l'ensemble des concepts qui couvrent l'application présentée. L'application est basée sur la vente et l'achat de bateaux militaires de pays dans le monde. Elle propose de visualiser les données d'un pays, de renseigner de nouveaux pays, de mettre un bateau en vente ainsi que d'en acheter.

Vous trouverez joints dans ce compte rendu une liste d'insertions de base permettant d'ajouter les plus grandes forces militaires mondiales comptant seize pays avec des ports militaires qui leurs sont assignés. Chaque port compte des bateaux militaires. Il y en a 584 de base.

Base de données

Mise en place d'un environnement de travail

Pour créer un environnement de travail utilisable, j'ai mis le projet en public et j'ai créé un schéma.

```
01 | SET search_path TO naval_world, public ;
02 |
03 | CREATE SCHEMA NAVAL_WORLD;
04 | SET search_path TO NAVAL_WORLD, public;
```

Les tables

La table 'Country' définit les différents pays qui sont renseignés (vous trouverez dans l'annexe la définition de tous les pays au premier lancement de l'application). Au premier lancement de l'application vous trouverez renseignées les seize plus grandes puissances militaires navales. Les colonnes sont :

- country_id : représente l'identifiant unique du pays
- country_name : correspond au nom en anglais du pays
- country_money : représente l'argent du pays en million de dollars
- country_power : représente une valeur calculée sur la somme des puissances de tous les bateaux du pays
- surface : représente la surface du pays en m²

Schéma relationnel de la table :

Country(country_id, country_name, country_money, country_power, surface)

```
01 | CREATE TABLE Country (
02 |     country_id INT PRIMARY KEY,
03 |     country_name VARCHAR(42),
04 |     country_money INT,
05 |     country_power INT,
06 |     surface INT
07 | );
```

La table 'Country_Affection' définit l'affection d'un pays vers un autre. Un script python (nommé randomAffinities.py) permet de modifier toutes les valeurs d'affection pour chaque pays pour des valeurs aléatoires entre 1 et 100. Un trigger initialise la valeur d'affection entre tous les pays et un nouveau pays créé, et réciproquement. La raison de la présence de cette table est la suivante : un pays n'a pas forcément la même affection pour un autre pays que ce pays pour le premier. Vous trouverez ci-dessous la liste de toutes les colonnes :

- country_id_in : représente l’identifiant du pays qui juge l’affection
- country_id_out : représente l’identifiant du pays qui est jugé
- score : représente le niveau d’affection entre les deux pays de 0 à 100.

Schéma relationnel de la table :

Country_Affection(#country_id_in, #country_id_out, score)

```

01 | CREATE TABLE Country_Affection (
02 |     country_id_in INT REFERENCES Country(country_id),
03 |     country_id_out INT REFERENCES Country(country_id),
04 |     score FLOAT,
05 |
06 |     PRIMARY KEY(country_id_in, country_id_out)
07 | );

```

La table ’Region’ définit la liste des régions (ports militaires) de tous les pays (vous trouverez dans l’annexe la définition de toutes les regions au premier lancement de l’application). Plusieurs régions peuvent être renseignées pour un même pays. Les coordonnées d’une région sont totalement indépendantes de celles d’un pays. Les colonnes sont :

- region_id : représente l’identifiant unique de la région
- region_name : correspond au nom en anglais de la région
- latitude : représente la latitude de la region
- longitude : représente la longitude de la région

Schéma relationnel de la table :

Region(region_id, region_name, latitude, longitude)

```

01 | CREATE TABLE Region (
02 |     region_id INT PRIMARY KEY,
03 |     region_name VARCHAR(200),
04 |     latitude FLOAT,
05 |     longitude FLOAT
06 | );

```

La table ’Country_Region’ fait le lien entre les pays et les régions (vous trouverez dans l’annexe la définition de toutes les valeurs par défaut). Les colonnes sont :

- country_id : représente l’identifiant unique du pays
- region_id : représente l’identifiant unique de la region

Schéma relationnel de la table :

Country_Region(#country_id, #region_id)

```

01 | CREATE TABLE Country_Region (
02 |     country_id INT REFERENCES Country(country_id),
03 |     region_id INT REFERENCES Region(region_id),
04 |
05 |     PRIMARY KEY(country_id, region_id)

```

```
06 | );
```

La table 'Batiment_type' définit les différents types de batiment existants (vous trouverez dans l'annexe la définition de toutes les valeurs par défaut). Il y en a 9 renseignés. Les colonnes sont :

- batiment_type_id : représente l'identifiant unique d'un type de bateau
- batiment_type_name : représente le nom d'un type de batiment
- batiment_power : représente la puissance du type de bateau
- max_health : représente la santé d'un batiment militaire
- costOfProductionInMillions : représente le coût de production d'un batiment de ce type en millions de dollars

Schéma relationnel de la table :

Batiment_type(batiment_type_id, batiment_type_name, batiment_power, max_health, costOfProductionInMillions)

```
01 | CREATE TABLE Batiment_type (
02 |     batiment_type_id INT PRIMARY KEY,
03 |     batiment_type_name VARCHAR(50),
04 |     batiment_power INT,
05 |     max_health INT,
06 |     costOfProductionInMillions INT
07 | );
```

La table 'Batiment' définit tous les bateaux du monde (vous trouverez dans l'annexe la définition de toutes les valeurs par défaut). Les colonnes sont :

- batiment_id : représente l'identifiant unique du bateau
- region_id : représente l'identifiant unique de la région à laquelle le bateau est assigné
- batiment_type_id : représente l'identifiant du type de batiment
- batiment_name : représente le surnom du bateau
- latitude : représente la position latitude du batiment en projection mercator
- longitude : représente la position longitude du batiment en projection mercator
- health : représente santé du bateau
- comment : un commentaire sur le batiment peut être écrit ici
- state : 0 = actif et 1=en vente (inactif)

Schéma relationnel de la table :

Batiment(batiment_id, #region_id, #batiment_type_id, batiment_name, latitude, longitude, health, comment, state)

```
01 | CREATE TABLE Batiment (
```

```

02 |     batiment_id INT PRIMARY KEY,
03 |     region_id INT REFERENCES Region(region_id),
04 |     batiment_type_id INT REFERENCES Batiment_type(batiment_type_id),
05 |     batiment_name VARCHAR(30),
06 |     latitude FLOAT,
07 |     longitude FLOAT,
08 |     health INT,
09 |     comment VARCHAR(2048),
10 |     state INT -- 0: actif 1: en vente (inactif)
11 |   );

```

La table 'Action_Batiment' définit le besoin qu'a un batiment à se déplacer de son origine à un emplacement latitude longitude. Les colonnes sont :

- action_id : représente l'identifiant unique d'un déplacement
- batiment_id : correspond au batiment visé
- goalLat : représente l'objectif en latitude
- goalLong : représente l'objectif en longitude

Schéma relationnel de la table :

Action_Batiment(action_id, #batiment_id, goalLat, goalLong)

```

01 | CREATE TABLE Action_Batiment (
02 |     action_id SERIAL PRIMARY KEY,
03 |     batiment_id INT REFERENCES Batiment(batiment_id),
04 |     goalLat FLOAT,
05 |     goalLong FLOAT
06 |   );

```

La table 'Contract' définit la liste des contrats. Ils sont plutôt complexes et seront définis plus loin. Les colonnes sont :

- contract_id : représente l'identifiant d'un contrat
- region_id_seller : correspond à l'identifiant de la région qui vend le bateau
- batiment_id : représente l'id du batiment en vente
- minCost : représente la prix minimal (affection à 100) d'un contrat en million de dollars
- maxCost : représente la prix maximal (affection à 0) d'un contrat en million de dollars
- tau : représente une variable gerant le prix de vente
- r : représente une variable gérant le prix de vente
- date_creating : donne la date de création d'un contrat
- active : informe si le contrat est toujours actif ou a été vendu

Schéma relationnel de la table :

Contract(contract_id, #region_id_seller, #batiment_id, minCost, maxCost, tau, r, date_creating, active)

```

01 | CREATE TABLE Contract (
02 |     contract_id SERIAL PRIMARY KEY,
03 |     region_id_seller INT REFERENCES Region(region_id),
04 |     batiment_id INT REFERENCES Batiment(batiment_id),
05 |     minCost INT,
06 |     maxCost INT,
07 |     tau INT,
08 |     r INT,
09 |     date_creating DATE,
10 |     active INT
11 | );

```

La table 'Command' définit la liste des commandes enregistrées. Suite à la signature d'un contrat, une commande apparaît et peut être utilisée comme des logs. Les colonnes sont :

- command_id : représente l'identifiant unique de la commande
- contract_id : correspond à l'identifiant unique d'un contrat
- region_id_buyer : correspond à l'identifiant unique de la région qui a acheté le contrat
- date_buy : informe sur la date d'achat du contrat

Schéma relationnel de la table :

Command(command_id, #contract_id, #region_id_buyer, date_buy, price)

```

01 | CREATE TABLE Command (
02 |     command_id SERIAL PRIMARY KEY,
03 |     contract_id SERIAL REFERENCES Contract(contract_id),
04 |     region_id_buyer INT REFERENCES Region(region_id),
05 |     date_buy DATE,
06 |     price INT
07 | );

```

Triggers

Un trigger a été rajouté. Il permet, lorsque qu'un pays est créé, de rajouter un lien entre ce pays et réciproquement.

```

01 | CREATE OR REPLACE FUNCTION SetDefaultCountryAffection()
02 | RETURNS TRIGGER
03 | AS $$
04 | DECLARE
05 |     country Country%rowtype;
06 | BEGIN
07 |     FOR country IN (
08 |         SELECT *
09 |         FROM Country
10 |     ) LOOP
11 |         INSERT INTO Country_Affection(country_id_in, country_id_out, score)
12 |             VALUES (new.country_id, country.country_id, 40)
13 |             ON CONFLICT (country_id_in, country_id_out)
14 |                 DO NOTHING;
15 |

```

```

16 |     INSERT INTO Country_Affection(country_id_in, country_id_out, score)
17 |         VALUES (country.country_id, new.country_id, 60)
18 |     ON CONFLICT (country_id_in, country_id_out)
19 |     DO NOTHING;
20 |
21 |     END LOOP;
22 |     RETURN new;
23 |   END;
24 | $$ LANGUAGE PLPGSQL;
25 |
26 | CREATE TRIGGER SetDefaultCountryAffection
27 | AFTER INSERT ON Country
28 | FOR EACH ROW
29 | EXECUTE PROCEDURE SetDefaultCountryAffection();

```

Projet I32 - MEA

Chareyre Adam | December 17, 2021

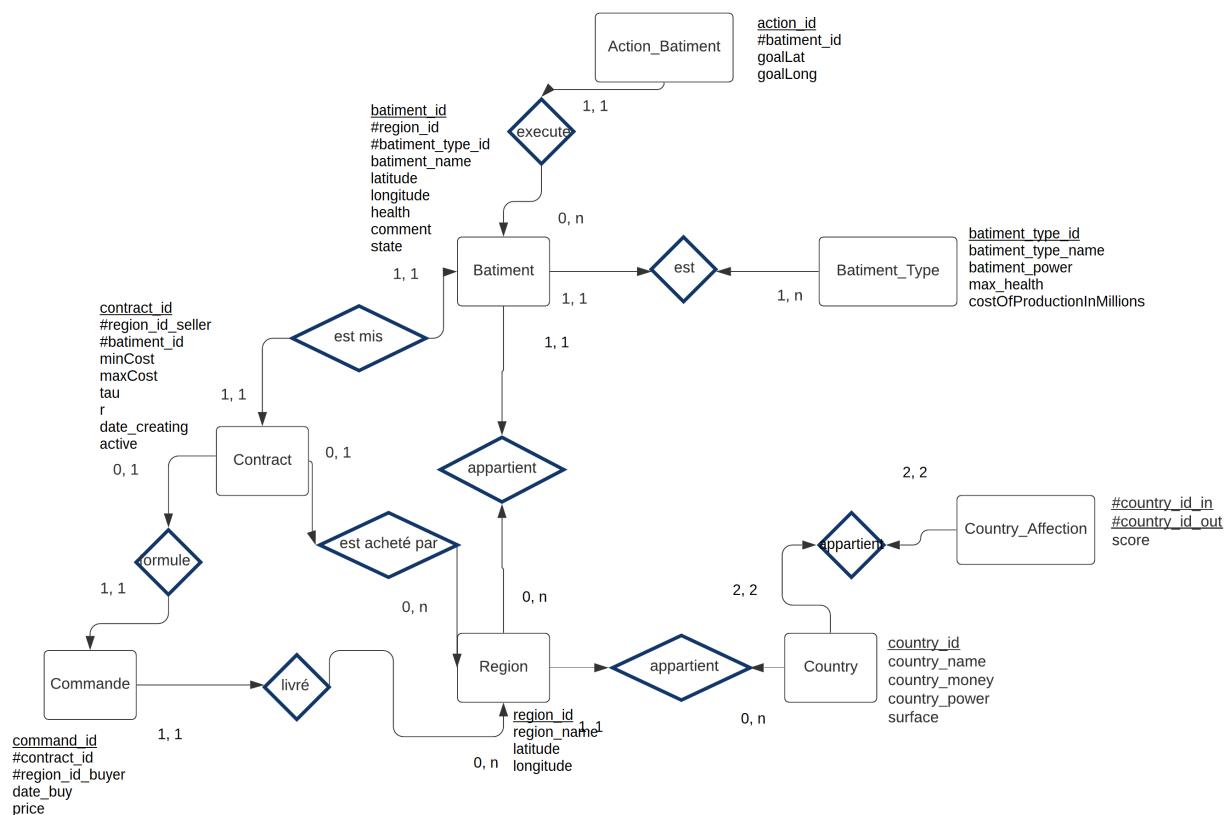


FIGURE 1 – MEA Projet I32

Application

Introduction

Le moyen d'interaction avec la base de donnée est une application de type GUI programmée en Python utilisant l'API Tkinter qui est une bibliothèque graphique. Un thème noir a été appliqué pour réduire l'effort de lecture.

L'application démarre avec une page constituée de quatre boutons qui permettent différentes actions qui seront développées plus loin.

Le schéma suivant met en exergue quelles tables sont utilisées en fonction du bouton qui est utilisé.

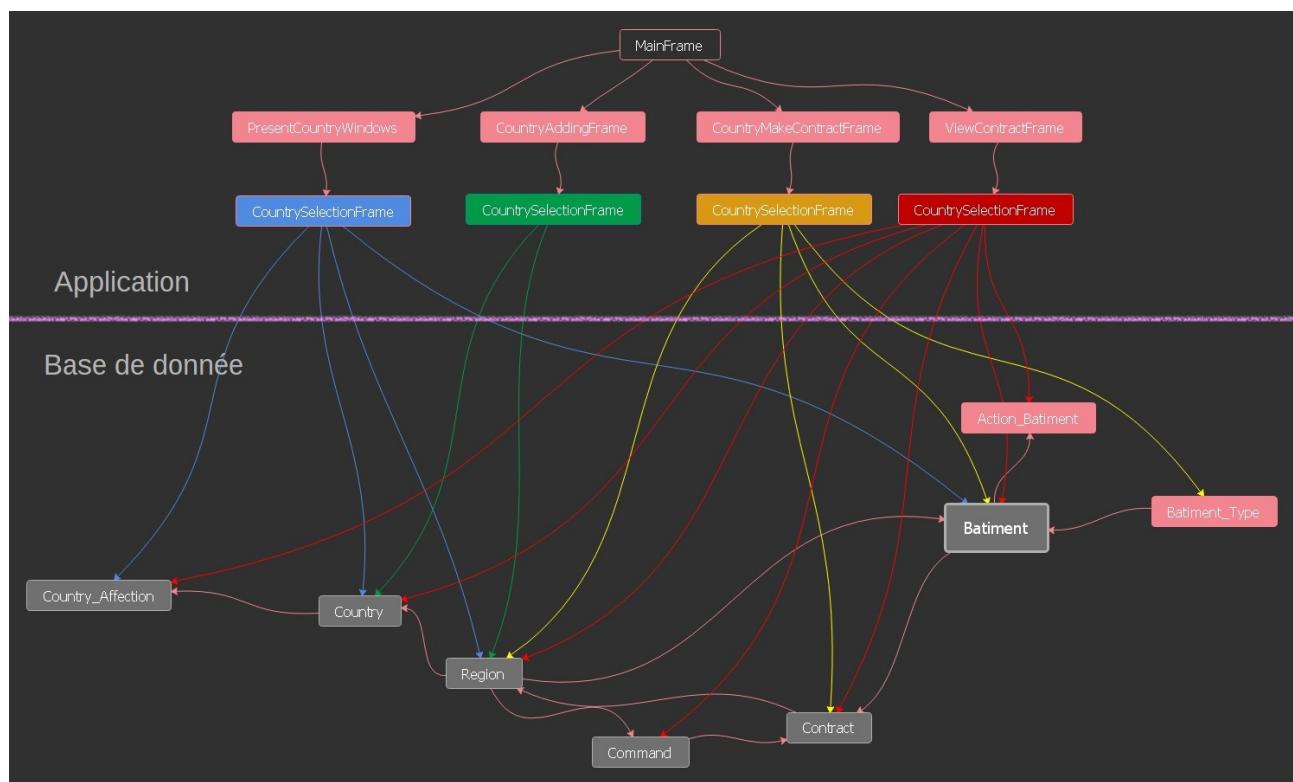


FIGURE 2 – Schéma explicatif Projet I32

Pré-requis

Certains pré-requis sont nécessaires pour que l'application fonctionne correctement.

Tout d'abord, il est nécessaire d'avoir une version de Python assez récente. Cette application a été développée dans la version 3.9 de Python, cela peut-être une référence pour vous en cas de coquille.

De plus, certaines librairies doivent être installées si elles ne sont pas déjà présentes sur la machine de l'utilisateur. Elles sont les suivantes :

- Tkinter : permet d'obtenir une interface graphique interactive
- numpy : permet de travailler avec des images
- shapely : permet de travailler avec des polygones plus facilement
- psycopg2 : permet d'intéragir avec la base de donnée postgresql
- json : permet de travailler avec des fichiers json (ici geojson)
- cProfile et pstats : permet d'obtenir une analyse des performances de la dernière exécution de l'application
- snakeviz : Permet de visualiser l'analyse de performance (attention, ce n'est pas une librairie python mais bien un programme à installer)
- matplotlib : permet un affichage de graphiques

Vous devez créer la base de donnée à l'aide du script sql nommé 'create_all.sql'. Une fois cela fait, vous devrez mettre des données de base dans la base, pour cela exécutez le script sql nommé 'insert_data.sql'.

Le rôle de ces scripts est de créer les tables et de les remplir. Il y a notamment plus de 500 bateaux qui ont été renseignés. Cependant ils ont été enregistrés à la position exacte des régions. Pour les déployer, vous devez lancer le script python 'spread-Ships.py'. Un autre script python qui peut vous être utile est 'randomAffinities.py', il permet de mettre des valeurs d'affinités complètement aléatoire (bien qu'il ne soit aucunement obligatoire, il permet de bien mettre en exergue la variation des prix des bateaux en fonction du pays choisi).

A partir de maintenant vous pouvez lancer l'application avec le script python 'main.py'.

Menu Principal

Le menu principal se compose de quatre boutons :

- View Country : Permet de visuliser les informations d'un pays
- Add new country : Permet d'ajouter un nouveau pays et de saisir ses informations
- Create contract : Permet de créer un contrat
- View contrat : Permet à un pays de signer un contrat

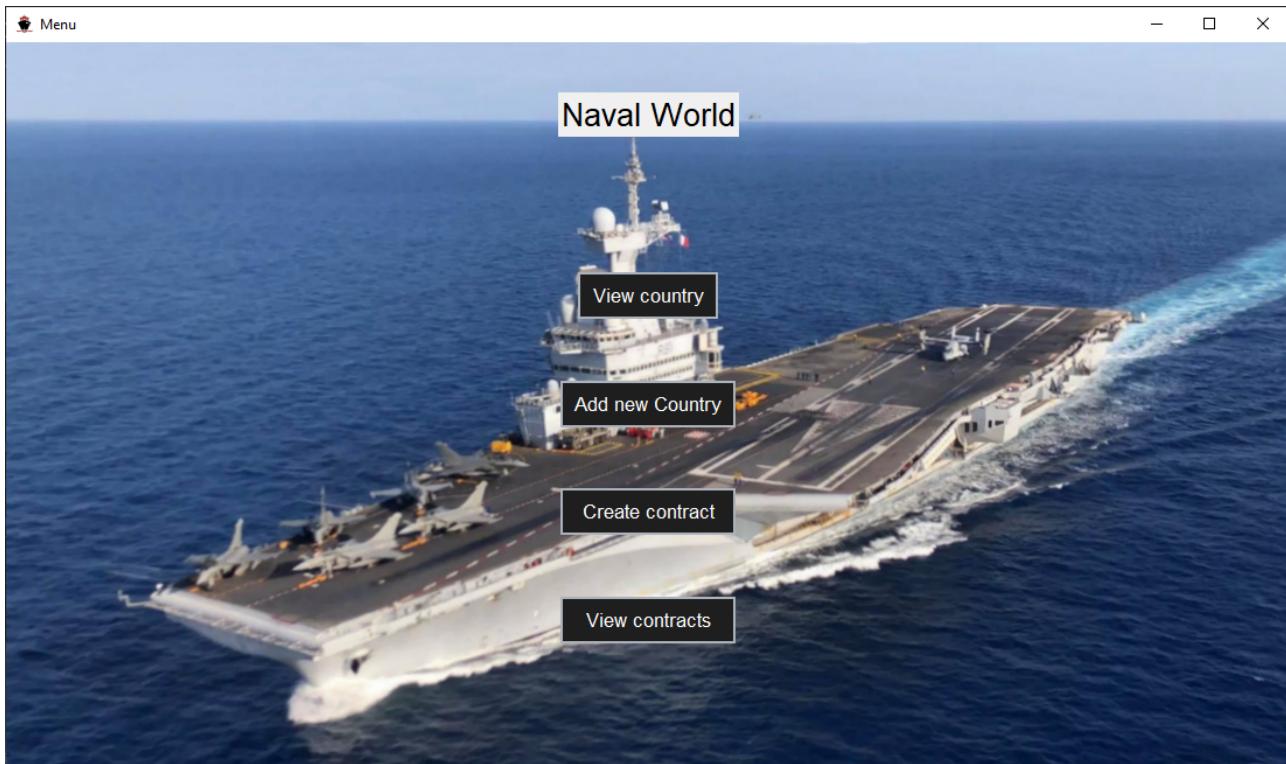


FIGURE 3 – Menu

Bouton 'View Country'

Après avoir cliqué sur un pays, l’application affichera les informations du pays notamment son nom, son budget militaire, sa surface, ses affinités avec tous les pays renseignés. Une fois un port militaire sélectionné, vous serez en mesure de connaître également les informations d’un des bateaux du port. Il existe également un bouton ‘view ships in real time’ qui permet de visualiser la position actuelle des bateaux du port militaire choisi. Chaque bateau est représentés par un petit point blanc (cf figure 5).

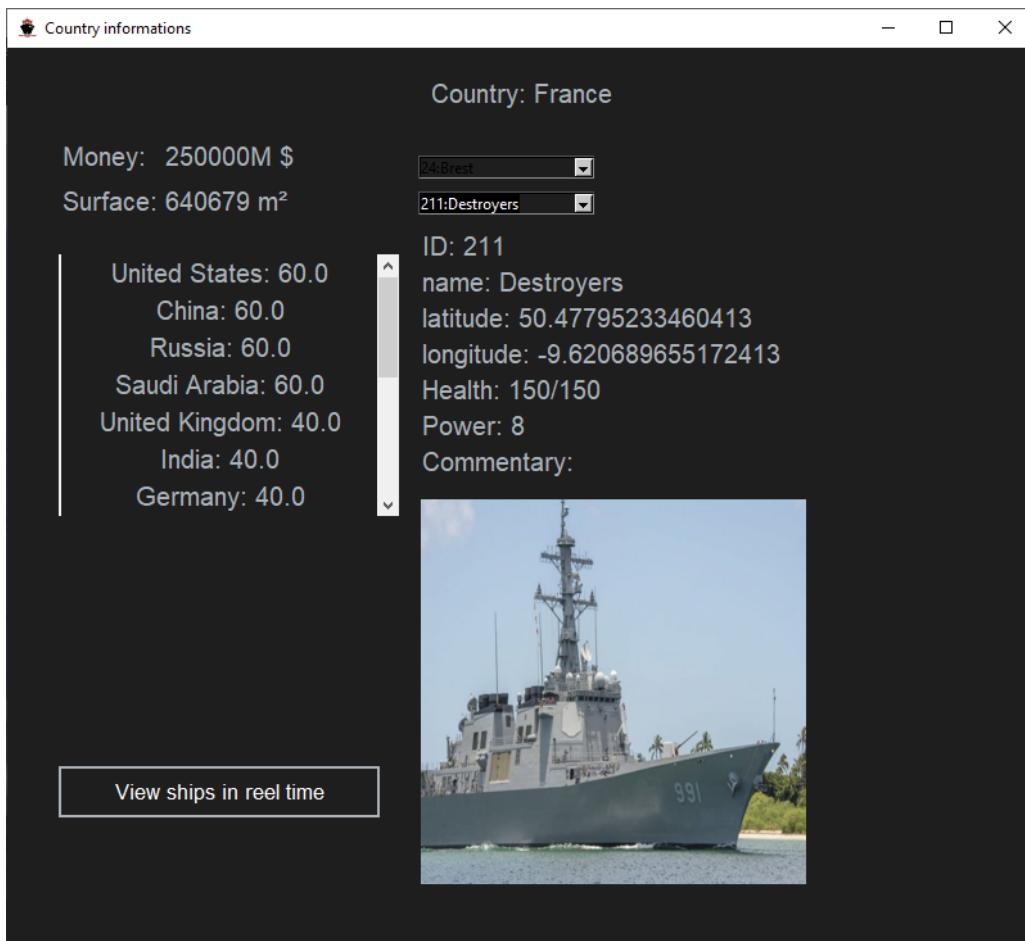


FIGURE 4 – Informations d'un pays

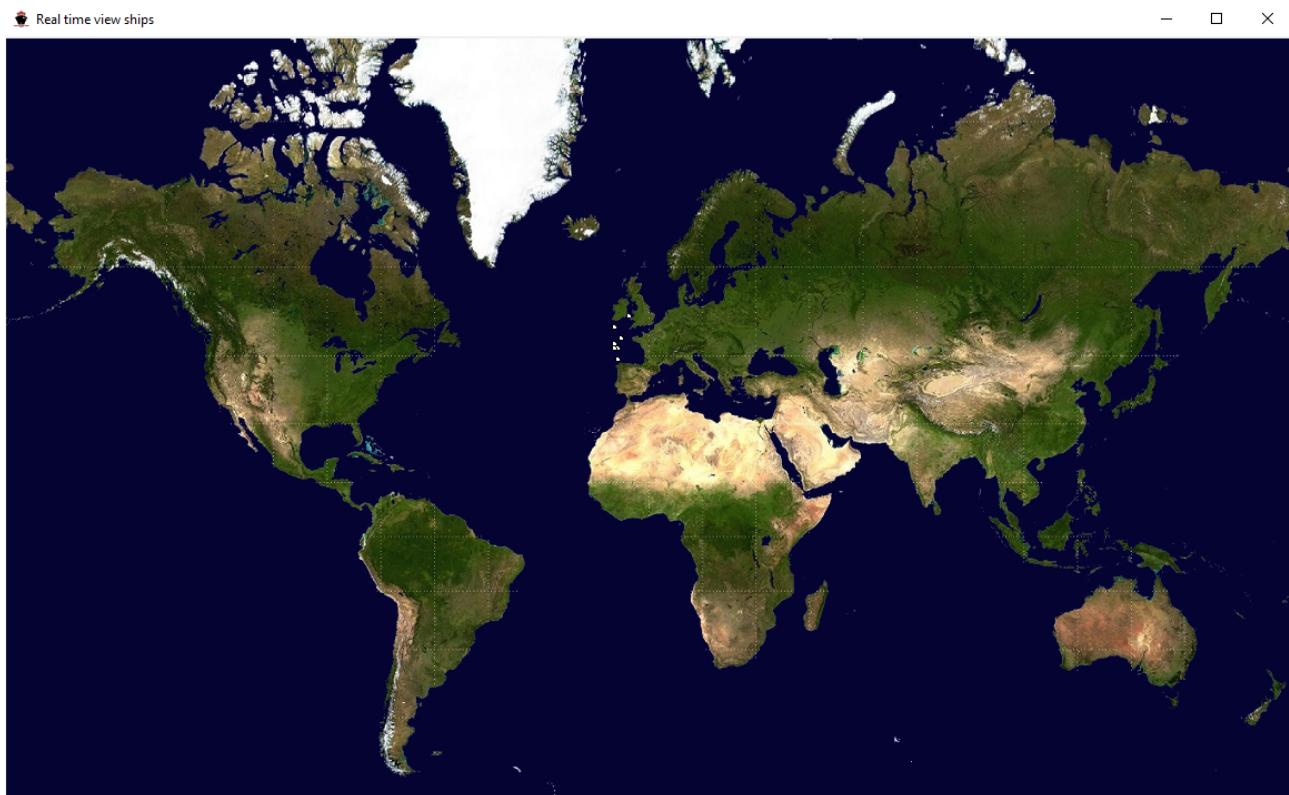


FIGURE 5 – Vue en temps réel des bateaux d'une région, ici Brest

Bouton 'Add new country'

Comme dit précédemment, seulement seize des plus grandes forces militaires sont renseignés dans la base de donnée. Cependant vous pouvez ajouter de nouveaux pays grâce à ce bouton. Après avoir cliqué sur le pays de votre choix, une nouvelle interface s'ouvrira avec des informations à saisir. Si une donnée au mauvais format est saisie, un message d'erreur apparaîtra vous informant du problème. (cf figure 7 et 8)

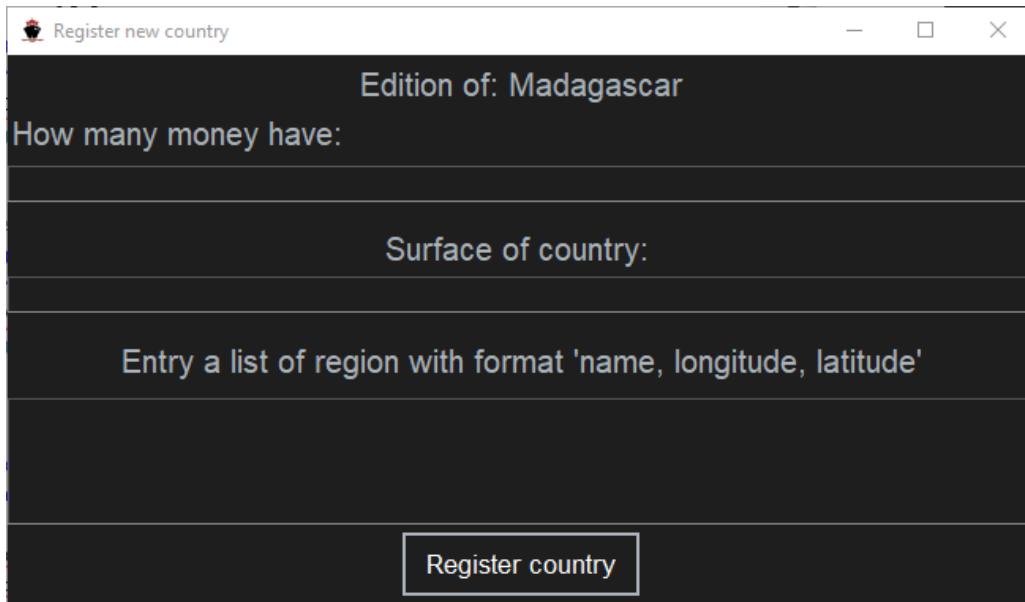


FIGURE 6 – Saisie de donnée pour l’ajout d’un nouveau pays

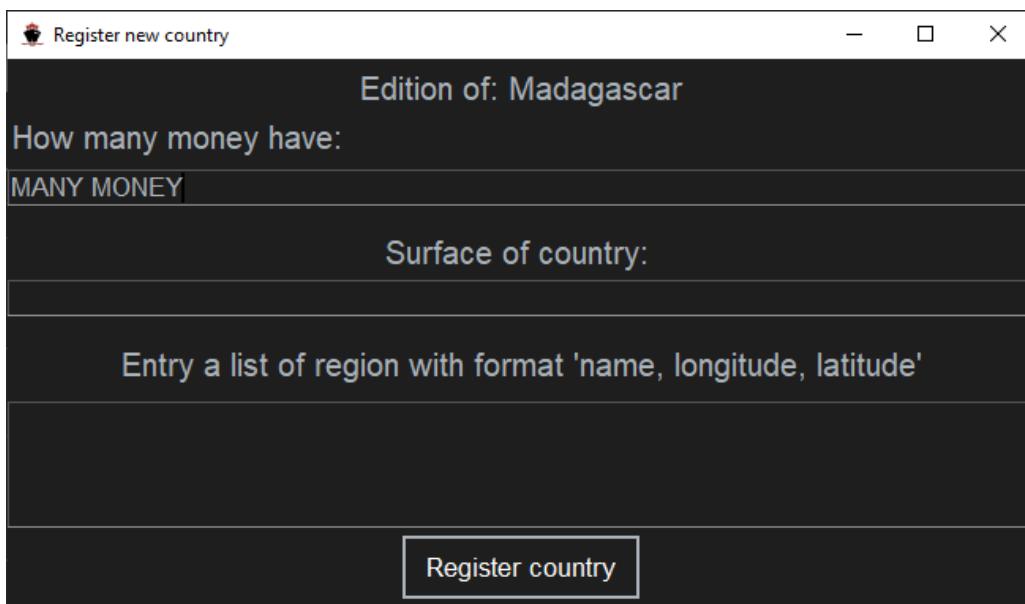


FIGURE 7 – Exemple de mauvaise saisie

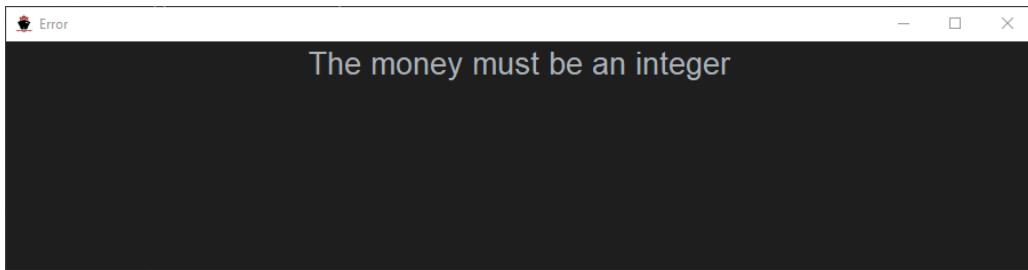


FIGURE 8 – Message d’erreur lors d’une mauvaise saisie

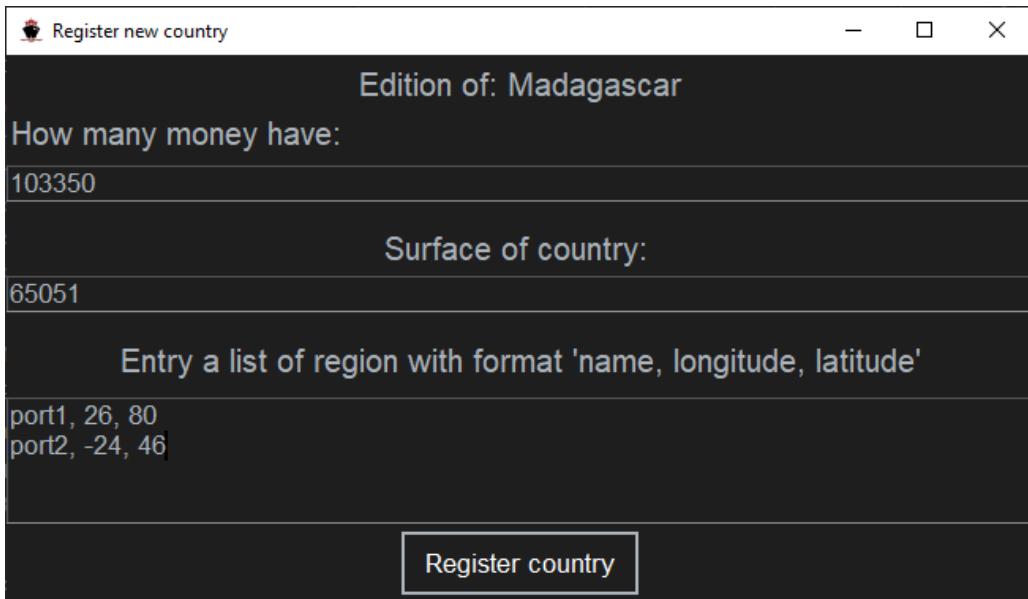


FIGURE 9 – Exemple de saisie correctement rédigée

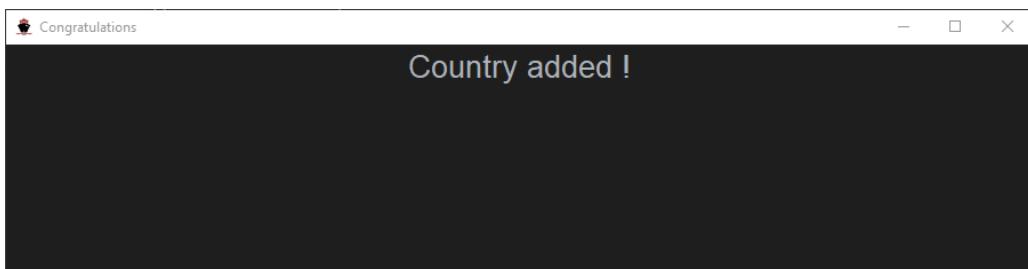


FIGURE 10 – Ajout avec succès d’un nouveaux pays

Bouton 'Create contract'

Vous pouvez aussi créer un contrat. Cela consiste à mettre en vente un des bateaux d'un pays. Après avoir sélectionné le pays qui vendra un bâtiment, vous arriverez sur une nouvelle interface qui vous proposera de choisir l'une des régions disponibles ainsi que le bateau de cette région.

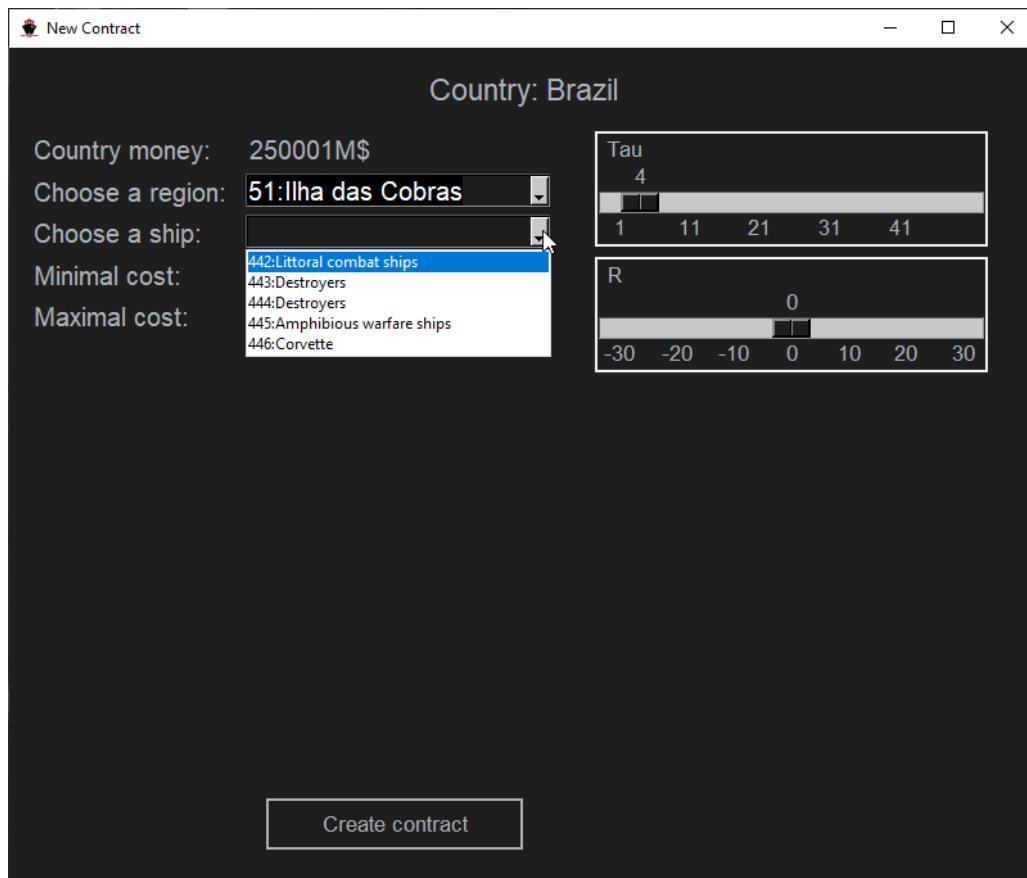


FIGURE 11 – Exemple de contrat

Une fois cela effectué, il sera nécessaire de renseigner le coût minimal et le coût maximal du bateau de votre choix. De base, le prix minimal se fixe à 110% du prix de construction du bateau et le prix maximal à 200% du prix de construction.

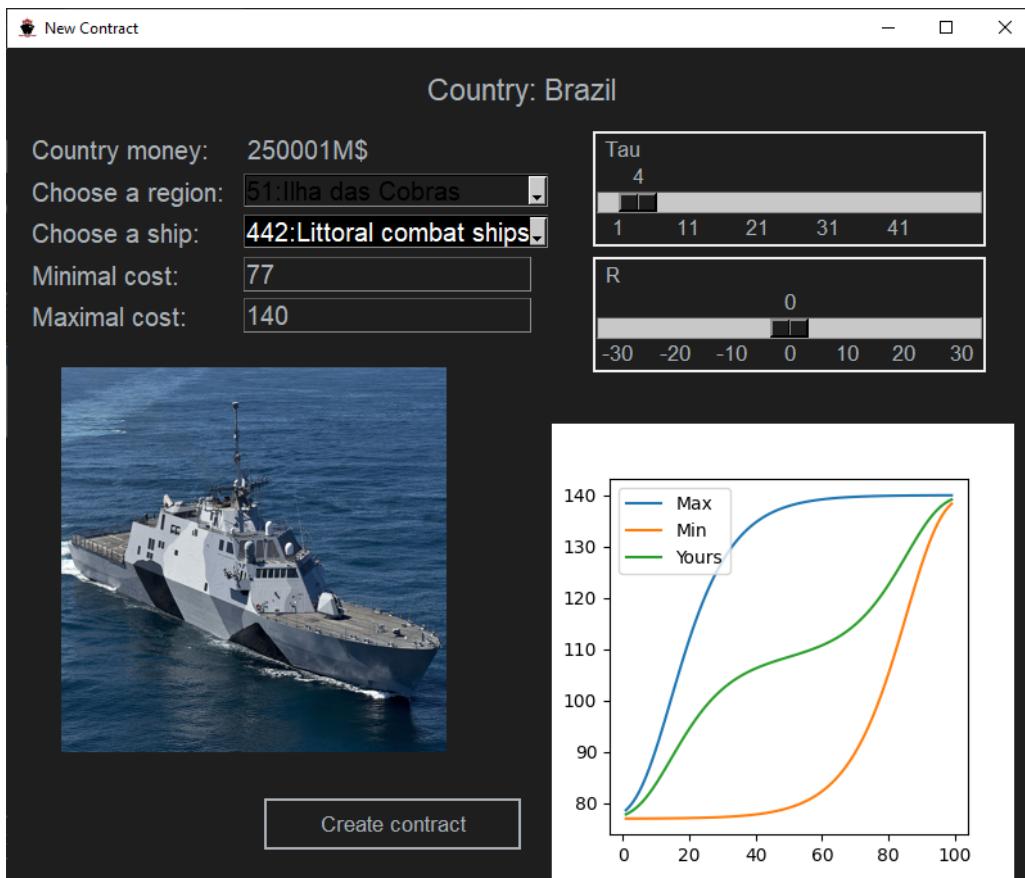


FIGURE 12 – Exemple de contrat

Une fois ces informations saisies, vous verrez un graphique apparaître. Il permet de voir quelle sera l'évolution du prix en fonction du niveau d'affection entre le pays qui crée le contrat et le pays qui le signe. Ces fluctuations sont expliquées par trois courbes :

- une bleu : représente l'évolution la plus précipitée possible
- une orange : représente l'évolution la plus retardée possible
- une verte : C'est la courbe que va suivre votre contrat

Grâce au paramètre 'R', vous pourrez faire tendre votre courbe vers la bleu ou la orange en fonction de votre choix personnel. Grâce au paramètre 'Tau', vous pourrez changer la forme des courbes bleu et orange à votre guise.

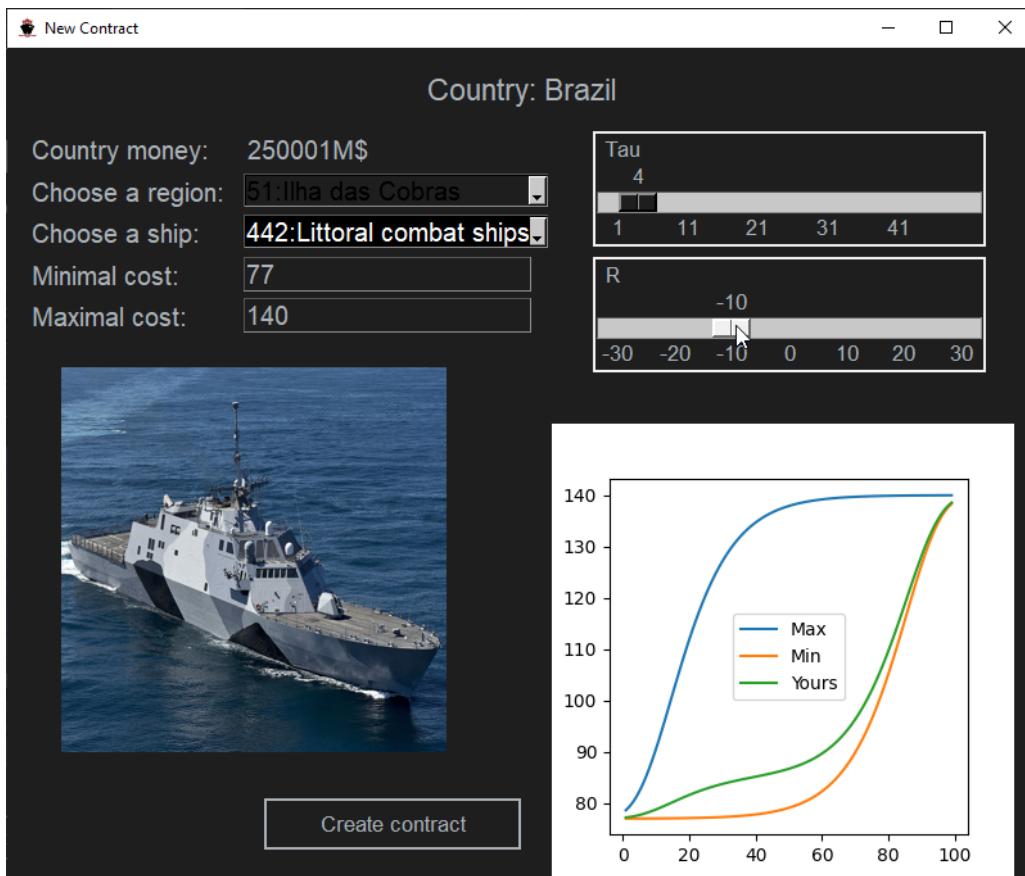


FIGURE 13 – Variation de R (négatif)



FIGURE 14 – Variation de R (positif)

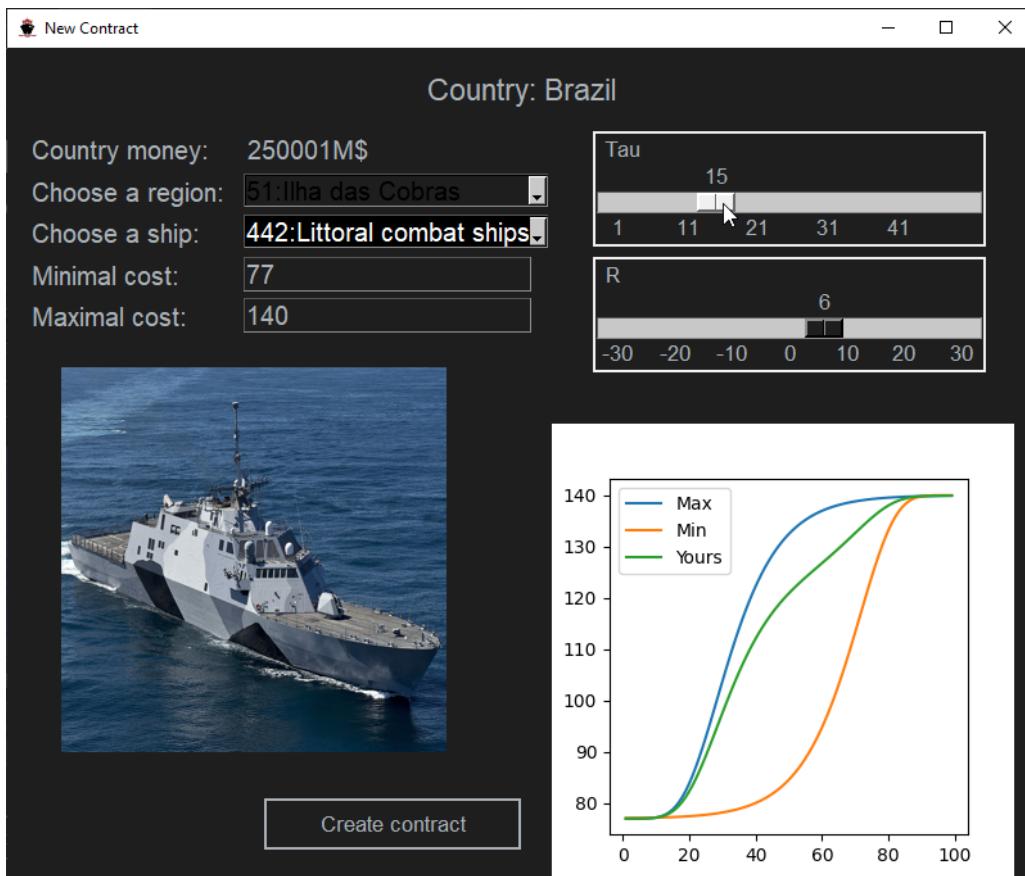


FIGURE 15 – Variation de Tau

Une fois avoir défini les différents paramètres de votre contrat, vous pourrez appuyer sur le bouton 'Create contract'.

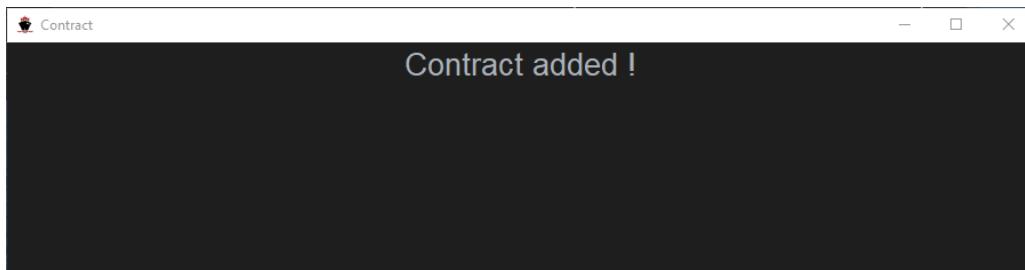


FIGURE 16 – Publication du contrat réussi

Bouton 'View contracts'

Ce bouton va vous permettre de signer des contrats. Après avoir sélectionné le pays acheteur, vous pourrez choisir le contrat de votre choix ainsi que l'un des ports où le bateau sera livré. Si le pays sélectionné possède assez d'argent, il pourra cliquer sur le bouton 'create contract' qui signera le contrat et lancera le déplacement du bateau qui sera visible à partir du bouton 'view ships in real time' (disponible après avoir cliqué sur le bouton 'View country' depuis le menu principal). Si l'appli-

cation se ferme pendant le trajet, lorsque l'application redémarrera, le déplacement reprendra là où il en était.

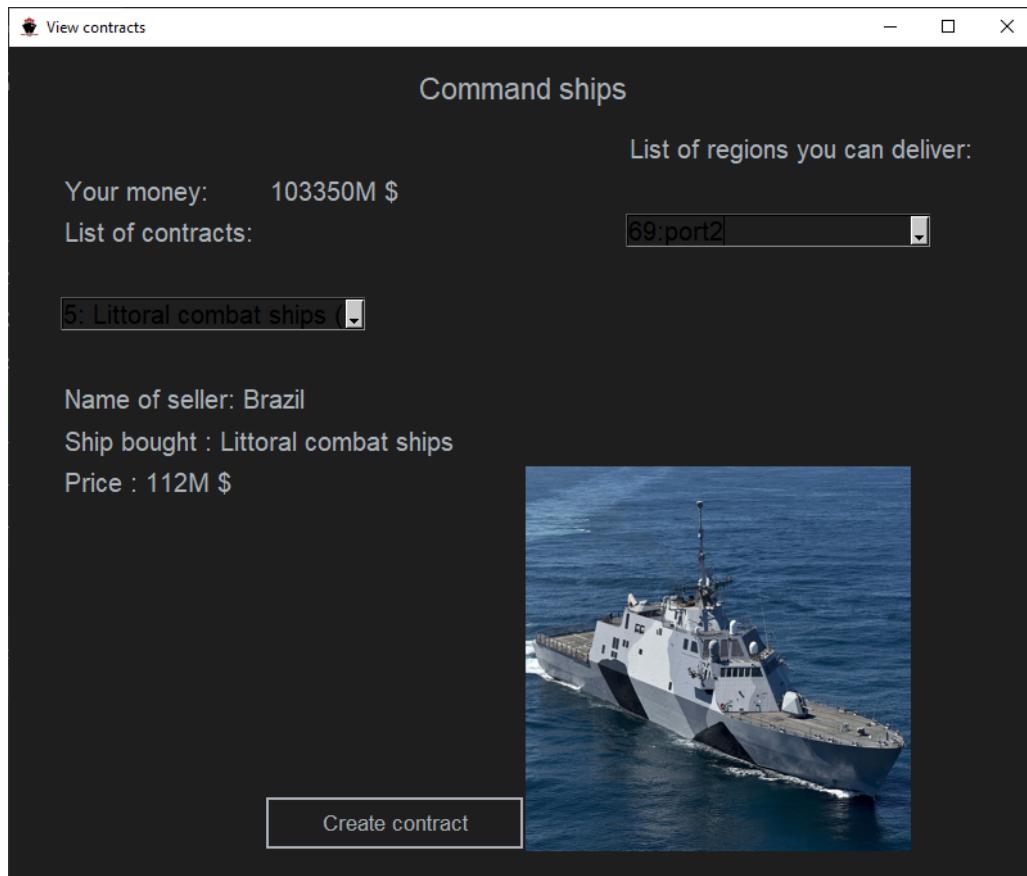


FIGURE 17 – Observation du nouveau contrat

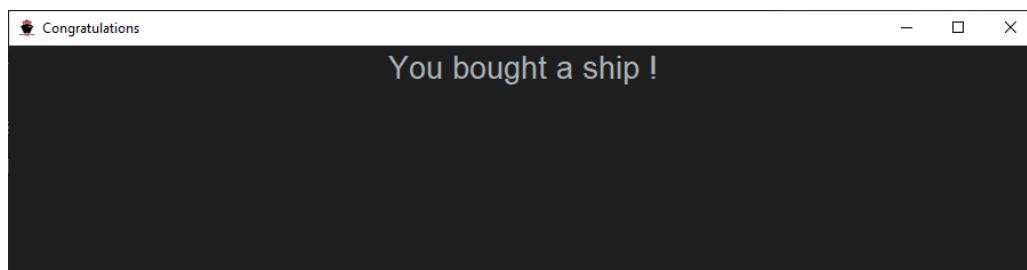


FIGURE 18 – Contrat signé avec succès

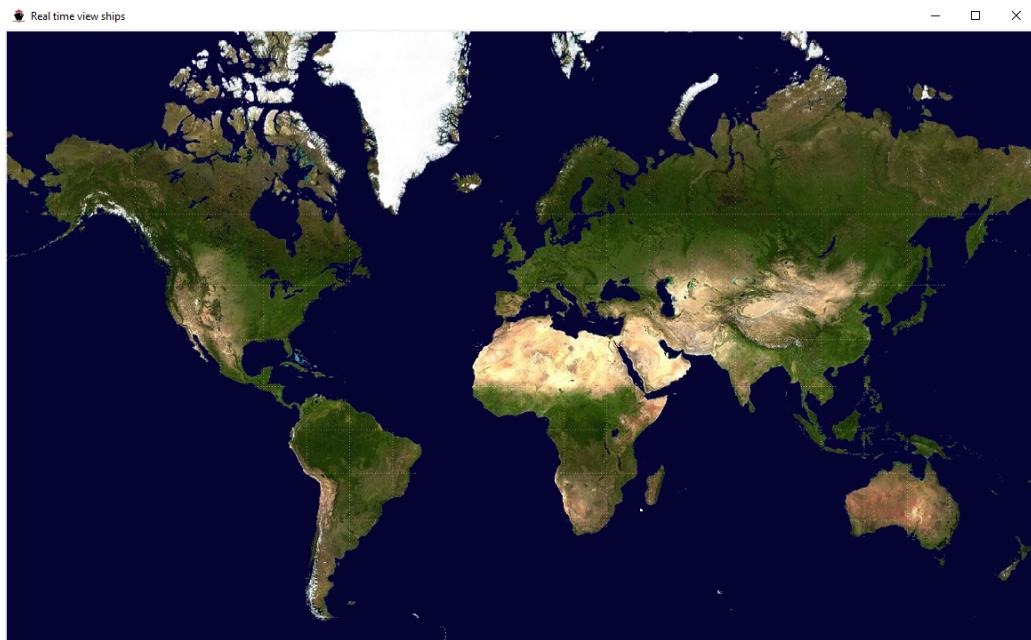


FIGURE 19 – Visualisation du bateau déplacé

Programmation de l'application

Introduction

cProfile, pstats et snakeviz

Lors du lancement de l'application, on peut choisir de lancer l'application en mode analyse de performance en modifiant la valeur de False à True.

```
01 | if __name__ == "__main__":
02 |     startGame(start, False)
```

Une fois l'application fermée, vous pouvez la visualiser avec la commande 'snakeviz profiling.prof' Cela a été important pour l'algorithme de path finding qui avait de très mauvaises performances au début (calcul d'un chemin de 20 à 300 secondes à actuellement au maximum 2 secondes)

Polygons

L'utilisateur a pu remarquer que lorsqu'il cliquait sur un pays, il était détecté sans même qu'il ait été renseigné dans la base de donnée. En réalité un fichier json (geojson) nommé 'cntry08.geojson' est présent dans le projet. Il décrit très précisement la liste des points qui définissent un pays en latitude et longitude (il représente un polygon). Un script python nommé 'jsonRawToRefined.py' a été rédigé pour convertir ce fichier en coordonnées x, y. Ainsi, le fichier perd un très gros poids (environ 94%) et nous gagnons un important temps de chargement. En effet, ce nouveau fichier, appelé 'countries.geojson' est chargé à chaque lancement de l'application. Il a également été ajouté dans ce fichier deux points qui définissent le coin supérieur gauche et le coin inférieur droit. Cela va être très utile pour gagner en performance.

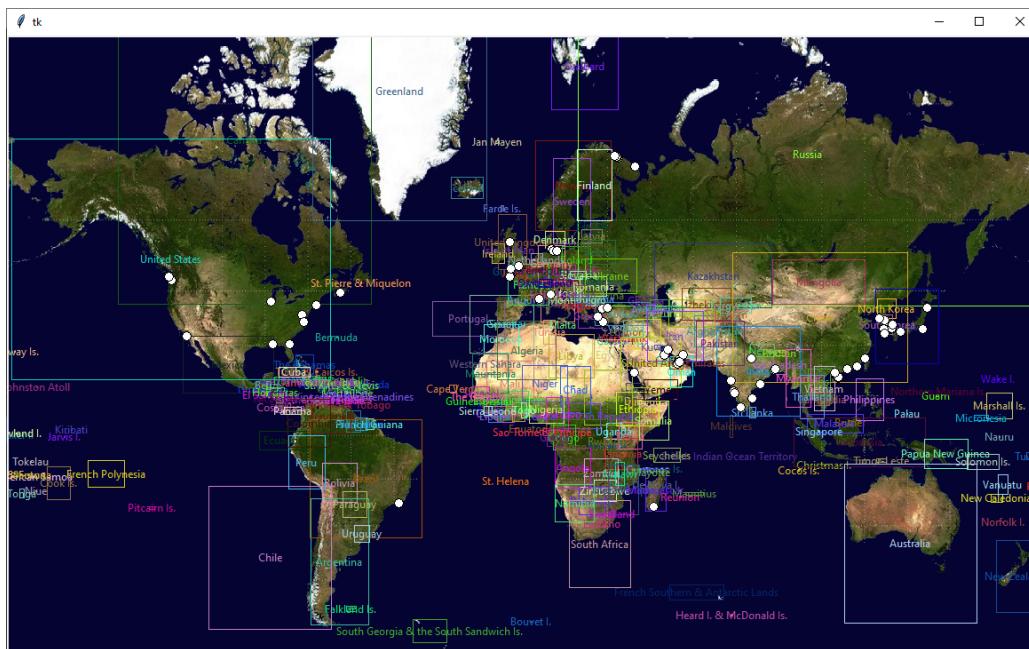


FIGURE 20 – Visualisation des rectangles de pays

En effet, pour savoir sur quel pays l'utilisateur clique, nous devons appliquer un algorithme pour savoir si un clic est dans un polynôme ou pas. Cependant, nous pouvons écarter la piste de tous les pays dont le rectangle n'est pas atteint par le clic. Nous collectons tous les pays qui sont concernés par le clic et nous appliquons ensuite l'algorithme de calcul de détection de clic dans un polygon.

Un algorithme pour créer ces rectangles a été créé. Il devait à la base être simple mais certains pays dépassaient à droite (ou à gauche en fonction du point de vue) et produisaient des rectangles qui faisaient la taille de l'image. Ce à quoi j'ai appliqué l'algorithme suivant :

```

01 | def bernoulliFilter(arrayOfPolygon, imageWidth):
02 |
03 |     countPart = [0] * 10
04 |
05 |     for poly in arrayOfPolygon:
06 |
07 |         for point in poly:
08 |
09 |             countPart[int((point[0] - 1) * 10 / imageWidth)] += 1
10 |
11 |     print(countPart)
12 |
13 |     if countPart[0] == 0 or countPart[9] == 0:
14 |
15 |         return arrayOfPolygon

```

```
12 |     print("→ Create poly replacement")  
13 |  
14 |     countLeft = countPart[0]  
15 |     countRight = 0  
16 |  
17 |     i = 0  
18 |  
19 |     while countPart[i] != 0:  
20 |  
21 |         countLeft += countPart[i]  
22 |  
23 |         i += 1  
24 |  
25 |  
26 |         middleZero = int((i + 1) * (imageWidth / 10))  
27 |  
28 |         print(middleZero)  
29 |  
30 |         while countPart[i] == 0:  
31 |  
32 |             i += 1  
33 |  
34 |  
35 |             countRight += countPart[i]  
36 |  
37 |             i += 1  
38 |  
39 |             bernoulliFilterPolygons = []  
40 |  
41 |             if countLeft > countRight:  
42 |  
43 |                 for poly in arrayOfPolygon:  
44 |  
45 |                     bernoulliFilterPoly = []  
46 |  
47 |                     for point in poly:  
48 |  
49 |                         if point[0] < middleZero:  
50 |  
51 |                             bernoulliFilterPoly.append(point)  
52 |  
53 |                     bernoulliFilterPolygons.append(bernoulliFilterPoly)  
54 |  
55 |             else:  
56 |  
57 |                 for poly in arrayOfPolygon:  
58 |  
59 |                     bernoulliFilterPoly = []  
60 |  
61 |                     for point in poly:  
62 |  
63 |                         if point[0] > middleZero:  
64 |  
65 |                             bernoulliFilterPoly.append(point)
```

```
46 |         bernoulliFilterPolygons.append(bernoulliFilterPoly)
47 |
48 |     return bernoulliFilterPolygons
```

Connection à la DDB

Un fichier a spécialement été conçu pour la connection à la base de donnée, nommé 'dbConnectionLoader.py'. Il permet d'initialiser une connection avec la base de donnée ainsi que fermer la connexion à la base de donnée à la fermeture de l'application et de mettre à jour tous les changements opérés sur la base de donnée. Il est également important de savoir que les informations de connexion tel que le mot de passe n'est pas écrit en clair dans le code. Il est renseigné dans un fichier nommé 'database.ini'. Cela permet de produire un système de hashage et une configuration plus maléable.

Intermédiaire entre la base de donnée et l'application

Un fichier Python nommé 'DDBUtils.py' a été créé dans l'optique de gagner du temps lors de requête.

Bouton tkinter custom

Cette classe n'a pas été codée par moi-même mais récupérée sur [github](#), le propriétaire est Tom Schimansky. Elle permet de créer des boutons plus esthétiques. En réalité je ne m'en suis que très peu servi. Il faut également savoir que cette librairie repose évidemment sur l'objet Button de Tkinter.

Scrollbar

Cette classe n'a pas été codée par moi même mais récupérée sur [stackoverflow](#). Elle permet de faire des scrollbar efficacement. Tkinter en possède de base mais sont instables et inutilisables.

Utils maths

Pour cette partie, nous verrons comment convertir des coordonnées latitude/longitude en projection mercatore à X/Y puis inversement :

$$\begin{aligned}
 x &= (\text{longitude} + 180) \times \left(\frac{\text{mapWidth}}{360}\right) \\
 \text{latitudeRad} &= \text{latitude} \times \left(\frac{\pi}{180}\right) \\
 \text{mercN} &= \log\left(\tan\left(\frac{\pi}{4} + \frac{\text{latitudeRad}}{2}\right)\right) \\
 y &= \frac{\text{mapHeight}}{2} - \left(\text{mapWidth} \times \frac{\text{mercN}}{2 \times \pi}\right)
 \end{aligned}$$

On peut donc déduire le code Python suivant :

```

01 | def GPSToCartesianCoordinates(latitude, longitude, mapWidth, mapHeight):
02 |     x = (longitude + 180) * (mapWidth / 360)
03 |     latitudeRad = latitude * pi / 180
04 |     mercN = 0
05 |     try:
06 |         mercN = log(tan((pi / 4) + (latitudeRad / 2)))
07 |     except ValueError:
08 |         return GPSToCartesianCoordinates(-89.9, longitude, mapWidth, mapHeight)
09 |     y = (mapHeight / 2) - (mapWidth * mercN / (2 * pi))
10 |     return x, y

```

Maintenant nous allons voir comment convertir des coordonnées X/Y à latitude/-longitude en projection mercatore :

$$\begin{aligned}
 \text{longi} &= \frac{x \times 360}{\text{mapWidth}} - 180 \\
 \text{mercN} &= \frac{2 \times \pi \times (-y + \frac{\text{mapHeight}}{2})}{\text{mapWidth}} \\
 \text{latitudeRad} &= 2 \times \arctan\left(e^{\text{mercN}} - \frac{\pi}{2}\right) \\
 \text{lat} &= \text{latitudeRad} \times \frac{180}{\pi}
 \end{aligned}$$

On peut donc déduire le code Python suivant :

```

01 | def cartesianCoordinatesToGPS(x, y, mapWidth, mapHeight):
02 |     longi = ((x * 360) / mapWidth) - 180
03 |     mercN = ((-y + (mapHeight / 2)) * 2 * pi) / mapWidth
04 |     latitudeRad = 2 * atan(exp(mercN)) - pi/2
05 |     lat = latitudeRad * 180 / pi
06 |     return lat, longi

```

PathFinding

L'algorithme de path finding est utilisé pour le déplacement d'un bateau, d'un point A à un point B. Il est basé sur le pseudo-code suivant :

```

OPEN //the set of nodes to be evaluated
CLOSED //the set of nodes already evaluated
add the start node to OPEN

loop
    current = node in OPEN with the lowest f_cost
    remove current from OPEN
    add current to CLOSED

    if current is the target node //path has been found
        return

    foreach neighbour of the current node
        if neighbour is not traversable or neighbour is in CLOSED
            skip to the next neighbour

        if new path to neighbour is shorter OR neighbour is not in OPEN
            set f_cost of neighbour
            set parent of neighbour to current
            if neighbour is not in OPEN
                add neighbour to OPEN

```

FIGURE 21 – Pseudo code du fonctionnement de l'algorithme de path finding

Vous pouvez également lancer le script du path finding seul, il exécutera un labyrinthe généré. Le fichier possède également des tests unitaires. De nombreuses optimisations ont été réalisées car l'algorithme naïf était trop lent, cela a rendu le code particulièrement difficile à comprendre.

Fonctions prix en fonction de l'affection

Comme vu précédemment, il existe trois courbes qui sont définies par les trois fonctions suivantes qui sont définies pour $x \in [0; 100]$:

Soit $(minCost, maxCost) \in \mathbb{R}_+$ avec $minCost \leq maxCost$

Soit $tau \in]0; 50]$

Soit $r \in [-30; 30]$

$$up(x) = minCost + (maxCost - minCost) \times e^{-\frac{tau}{1.1^x}}$$

$$down(x) = maxCost - up(-x + 100) + minCost$$

$$price(x) = \frac{1.1^r \times up(x) + 1.1^{-r} \times down(x)}{1.1^r + 1.1^{-r}}$$

Il existe un fichier geogebra nommé 'contract_buying.ggb' qui permet de visualiser le rendu des trois fonctions

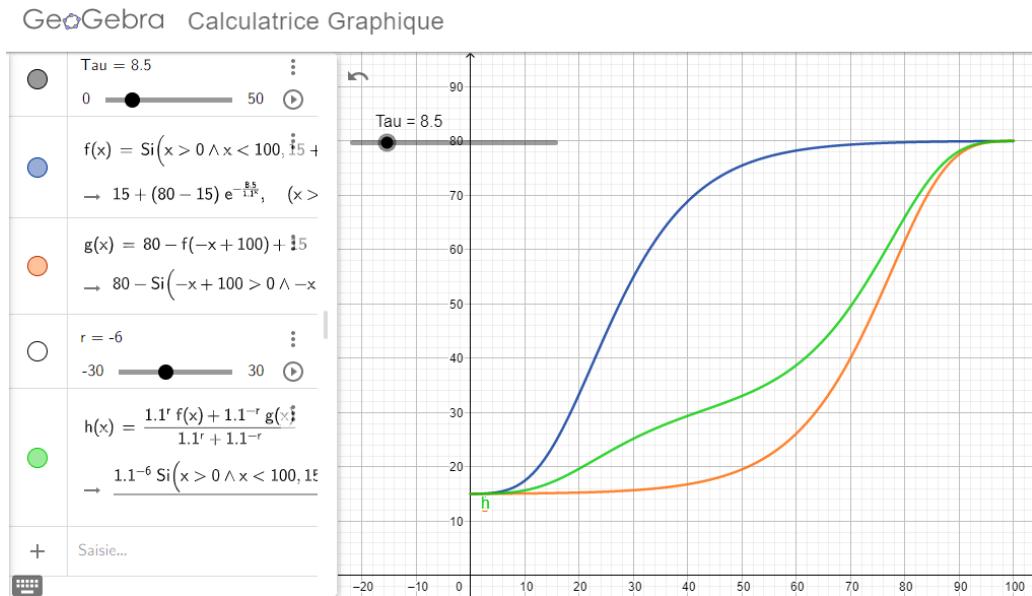


FIGURE 22

Intéractions entre les frames et la DDB

CountrySelectionFrame

```

01 | amount = DDBUtils.prepareRequest(f"""
02 | 
03 |     SELECT COUNT(*)
04 | 
05 |     FROM Country
06 | 
07 |     WHERE Country.country_name = '{countryClicked.getName()}';
08 | 
09 |     """, 0)[0]

```

```

01 | countryExist = DDBUtils.prepareRequest(f"""
02 | 
03 |     SELECT COUNT(*)
04 | 
05 |     FROM Country
06 | 
07 |     WHERE Country.country_name = '{countryClicked.getName()}';
08 | 
09 |     """, 0)[0]

```

PresentCountryWindows

```

01 | #Combobox des regions
02 | regions = DDBUtils.prepareRequest(f"""
03 | 
04 |     SELECT Region.*
05 | 
06 |     FROM Region, Country_Region
07 | 
08 |     WHERE Country_Region.country_id={countryID}

```

```

06 |             AND Country_Region.region_id=Region.region_id;
07 |         """
08 |
09 |
10 | shipsOfRegion = DDBUtils.prepareRequest(f"""
11 |     SELECT Batiment.batiment_id, Batiment.batiment_name
12 |     FROM Batiment, Region
13 |     WHERE Batiment.region_id = Region.region_id
14 |     AND Region.region_name = '{regionInComboBox}'
15 | """
16 |
17 |
18 | shipsInfo = DDBUtils.prepareRequest(f"""
19 |     SELECT Batiment.batiment_id, Batiment.batiment_name, Batiment.
20 |     latitude, Batiment.longitude, Batiment.health, Batiment_type.max_health,
21 |     Batiment_type.batiment_power, Batiment.comment
22 |     FROM Batiment, Batiment_type
23 |     WHERE Batiment.batiment_type_id = Batiment_type.batiment_type_id
24 |     AND Batiment.batiment_id = {boatID};
25 | """
26 |     """
27 |     , 1)[0]

```

CountryAddingFrame

```

01 | countryID = DDBUtils.prepareRequest(f"""
02 |     SELECT MAX(Country.country_id)
03 |     FROM Country;
04 | """
05 |     , 0)[0] + 1
06 |
07 |
08 | DDBUtils.prepareInsert(f"""
09 |     INSERT INTO Country(country_id, country_name, country_money,
10 |     country_power, surface)
11 |     VALUES ({countryID},
12 |             '{self.country.getName()}',
13 |             {int(self.textBoxMoney.get("1.0", "end-1c"))},
14 |             0,
15 |             {int(self.textBoxDimensions.get("1.0", "end-1c"))});
16 |
17 | """

```

```

01 | regionID = DDBUtils.prepareRequest(f"""
02 |     SELECT MAX(Region.region_id)
03 |     FROM Region;
04 |     """ , 0)[0] + 1
05 |
06 |
07 |
08 |
09 |
10 |
11 |
12 |
13 |
14 |

```

```

01 | for regionText in regionsTextArray:
02 |     regionInformation = regionText.split(", ")
03 |     regionName = regionInformation[0]
04 |     regionLat = regionInformation[1]
05 |     regionLong = regionInformation[2]
06 |     DDBUtils.prepareInsert(f"""
07 |         INSERT INTO Region(region_id, region_name, latitude, longitude)
08 |             VALUES ({regionID}, '{regionName}', {regionLat}, {regionLong});
09 |         """
10 |         DDBUtils.prepareInsert(f"""
11 |             INSERT INTO Country_Region(country_id, region_id)
12 |                 VALUES ({countryID}, {regionID});
13 |             """
14 |             regionID += 1

```

CountryMakeContractFrame

```

01 | regions = DDBUtils.prepareRequest(f"""
02 |     SELECT Region.*
03 |     FROM Region, Country_Region
04 |     WHERE Country_Region.country_id={countryID}
05 |     AND Country_Region.region_id=Region.region_id;
06 |     """
07 |
08 |
09 |
10 |
11 |
12 |
13 |
14 |

```

```

01 | shipsOfRegion = DDBUtils.prepareRequest(f"""
02 |     SELECT Batiment.batiment_id, Batiment.batiment_name
03 |     FROM Batiment, Region
04 |     WHERE Batiment.region_id = Region.region_id
05 |     AND Region.region_id = '{self.regionID}'

```

```

06 |             AND Batiment.state = 0;
07 |             """", 1)
08 |
09 |
10 |     costOfShipProduction = DDBUtils.prepareRequest(f"""
11 |         SELECT Batiment_type.costOfProductionInMillions
12 |             FROM Batiment, Batiment_type
13 |                 WHERE Batiment.batiment_id = {self.shipID}
14 |                     AND Batiment.batiment_type_id = Batiment_type.batiment_type_id;
15 |             """", 0)[0]
16 |
17 |
18 |     DDBUtils.updateRequest(f"""
19 |         UPDATE Batiment
20 |             SET state = 1
21 |                 WHERE Batiment.batiment_id = {self.shipID};
22 |             """")
23 |
24 |
25 |     DDBUtils.prepareInsert(f"""
26 |         INSERT INTO Contract(contract_id, region_id_seller, batiment_id, minCost
27 |             , maxCost, tau, r, date_creating, active)
28 |                 VALUES
29 |                     (DEFAULT, {self.regionID}, {self.shipID}, {self.textBoxMoneyMin.get
30 |                         ("1.0", "end")}, {self.textBoxMoneyMax.get("1.0", "end")}, {self.scaleTau.get
31 |                             ()}, {self.scaleR.get()}, current_timestamp, 1);
32 |             """")
33 |
34 |
35 |     self.contracts = DDBUtils.prepareRequest(f"""
36 |         SELECT Contract.contract_id, Contract.minCost, Contract.maxCost,
37 |             Contract.tau, Contract.r, Contract.date_creating, Country.country_id, Country
38 |                 .country_name, Batiment.batiment_id, Batiment.batiment_name,
39 |                     Country_Affection.score
40 |                         FROM Contract, Batiment, Country, Country_Region, Country_Affection
41 |                             WHERE Contract.batiment_id = Batiment.batiment_id
42 |                                 AND Batiment.state = 1
43 |                                     AND Contract.region_id_seller = Country_Region.region_id
44 |                                         AND Country.country_id = Country_Region.country_id
45 |                                             """)

```

```

08 |             AND Country.country_id = Country_Affection.country_id_in
09 |             AND Country_Affection.country_id_out = {self.countryID}
10 |             AND Contract.active = 1;
11 |         """", 1)


---


01 |     self.contracts = DDBUtils.prepareRequest(f"""
02 |         SELECT Contract.contract_id, Contract.minCost, Contract.maxCost,
03 |             Contract.tau, Contract.r, Contract.date_creating, Country.country_id, Country
04 |                 .country_name, Batiment.batiment_id, Batiment.batiment_name,
05 |             Country_Affection.score
06 |             FROM Contract, Batiment, Country, Country_Region, Country_Affection
07 |             WHERE Contract.batiment_id = Batiment.batiment_id
08 |                 AND Batiment.state = 1
09 |                 AND Contract.region_id_seller = Country_Region.region_id
10 |                 AND Country.country_id = Country_Region.country_id
11 |                 AND Country.country_id = Country_Affection.country_id_in
12 |                 AND Country_Affection.country_id_out = {self.countryID}
13 |                 AND Contract.active = 1;
14 |         """", 1)

```

ViewContractsFrame

```

01 |     self.contracts = DDBUtils.prepareRequest(f"""
02 |         SELECT Contract.contract_id, Contract.minCost, Contract.maxCost,
03 |             Contract.tau, Contract.r, Contract.date_creating, Country.country_id, Country
04 |                 .country_name, Batiment.batiment_id, Batiment.batiment_name,
05 |             Country_Affection.score
06 |             FROM Contract, Batiment, Country, Country_Region, Country_Affection
07 |             WHERE Contract.batiment_id = Batiment.batiment_id
08 |                 AND Batiment.state = 1
09 |                 AND Contract.region_id_seller = Country_Region.region_id
10 |                 AND Country.country_id = Country_Region.country_id
11 |                 AND Country.country_id = Country_Affection.country_id_in
12 |                 AND Country_Affection.country_id_out = {self.countryID}
13 |                 AND Contract.active = 1;
14 |

```

```
11 |         """ , 1)
```

```
01 | regions = DDBUtils.prepareRequest(f"""
02 |         SELECT Region.*
03 |         FROM Region, Country_Region
04 |         WHERE Country_Region.country_id={self.countryID}
05 |         AND Country_Region.region_id=Region.region_id;
06 |         """ , 1)
```

```
01 |     # Update affections
02 |     scoreAffection = DDBUtils.prepareRequest(f"""
03 |         SELECT Country_Affection.score
04 |         FROM Country_Affection
05 |         WHERE Country_Affection.country_id_in = {self.countryID}
06 |         AND Country_Affection.country_id_out = {contract[6]};
07 |         """ , 0)[0]
08 |
09 |     DDBUtils.updateRequest(f"""
10 |         UPDATE Country_Affection
11 |         SET score = {min(int(scoreAffection) * 1.1, 100)}
12 |         WHERE Country_Affection.country_id_in = {self.countryID}
13 |         AND Country_Affection.country_id_out = {contract[6]};
14 |         """)
15 |
16 |     scoreAffection = DDBUtils.prepareRequest(f"""
17 |         SELECT Country_Affection.score
18 |         FROM Country_Affection
19 |         WHERE Country_Affection.country_id_in = {contract[6]}
20 |         AND Country_Affection.country_id_out = {self.countryID};
21 |         """ , 0)[0]
22 |
23 |     DDBUtils.updateRequest(f"""
24 |         UPDATE Country_Affection
25 |         SET score = {min(int(scoreAffection) * 1.1, 100)}
```

```

26 |             WHERE Country_Affection.country_id_in = {contract[6]}
27 |             AND Country_Affection.country_id_out = {self.countryID};
28 |         """
01 | DDBUtils.prepareInsert(f"""
02 |             INSERT INTO Command(command_id, contract_id, region_id_buyer,
03 |             date_buy, price)
04 |             VALUES
05 |             (DEFAULT, {contract[0]}, {self.countryID}, current_timestamp, {
06 |             price});
07 |         """
08 |
09 | # Add action to batiment
10 |     regionID = self.comboBoxRegion.get().split(":")[0]
11 |     regionLatLong = DDBUtils.prepareRequest(f"""
12 |         SELECT Region.latitude, Region.longitude
13 |         FROM Region
14 |         WHERE Region.region_id = {regionID};
15 |     """, 0)
16 |
17 |     DDBUtils.prepareInsert(f"""
18 |         INSERT INTO Action_Batiment(action_id, batiment_id, goalLat,
19 |         goalLong)
20 |             VALUES
21 |             (DEFAULT, {contract[8]}, {regionLatLong[0]}, {regionLatLong[1]});
22 |     ;
23 |     """
24 |
25 | DDBUtils.prepareInsert(f"""
26 | # Update contract ID
27 |     DDBUtils.prepareDelete(f"""
28 |         UPDATE Contract
29 |             SET active = 0
30 |             WHERE Contract.contract_id = {contract[0]};
31 |     """

```

```

01 | # Update batiement values
02 |         DDBUtils.updateRequest(f"""
03 |             UPDATE Batiment
04 |                 SET region_id = {regionID},
05 |                     state = 0
06 |                 WHERE Batiment.batiment_id = {contract[8]};
07 |         """
08 |

```

```

01 | # Reduce money country
02 |         DDBUtils.updateRequest(f"""
03 |             UPDATE Country
04 |                 SET country_money = {currentMoney - price}
05 |                 WHERE Country.country_id = {self.countryID}
06 |         """
07 |

```

```

01 | # Add money country
02 |         countrySellerMoney = DDBUtils.prepareRequest(f"""
03 |             SELECT Country.country_money
04 |                 FROM Country
05 |                 WHERE Country.country_id = {contract[6]};
06 |         """
07 |
08 |         DDBUtils.updateRequest(f"""
09 |             UPDATE Country
10 |                 SET country_money = {countrySellerMoney + price}
11 |                 WHERE Country.country_id = {contract[6]};
12 |         """
13 |

```

BoatViewRealTimeFrame

```

01 | shipsPosLongLat = DDBUtils.prepareRequest(f"""
02 |             SELECT Batiment.latitude, Batiment.longitude
03 |                 FROM Batiment
04 |                 WHERE Batiment.region_id = {self.regionID};
05 |

```

```
05 |     """ , 1)
```

WalkerShip

```
01 | actionsBatiment = DDBUtils.prepareRequest(f"""
02 |         SELECT Action_Batiment.*
03 |         FROM Action_Batiment;
04 |     """ , DDBUtils.ALL)
```

```
01 | latLongActual = DDBUtils.prepareRequest(f"""
02 |         SELECT Batiment.latitude, Batiment.longitude
03 |         FROM Batiment
04 |         WHERE Batiment.batiment_id = {batimentID};
05 |     """ , DDBUtils.ONE)
```

```
01 |     DDBUtils.updateRequest(f"""
02 |         UPDATE Batiment
03 |             SET latitude = {currentLatLong[0]},
04 |                 longitude = {currentLatLong[1]}
05 |             WHERE Batiment.batiment_id = {batimentID};
06 |     """ )
```

```
01 | DDBUtils.prepareDelete(f"""
02 |         DELETE FROM Command
03 |         WHERE Command.command_id = {commandID};
04 |     """ )
```

Conclusion

Ce projet était un véritable défi. J'ai eu beaucoup de mal à choisir un sujet adéquat. Il y a d'abord eu le projet camping puis un autre projet autour d'un jeu avec des profils d'utilisateurs. Ce dernier était actif encore trois semaines avant la date de présentation du projet. J'ai changé de sujet pour finalement une application portant sur la vente de bateaux de guerre.

Avoir été seul dans ce projet m'aura malgré cela permis de voir toutes les facettes du développement de l'application. J'ai appris à me servir de l'API Tkinter et la façon qu'il a de gérer les widgets. J'ai également appris à programmer un algorithme de path finding et à mettre en pratique les connaissances acquises lors de ce semestre en I32.