

# 3D - Java Labs 2022

## Catch me if you can !

Nicolas Papazoglou, Antoine Tauvel, ENSEA

Décembre 2021

1 Introduction

Bienvenue dans ces séances de TP. Voici quelques règles pour le bon déroulement de ces séances :

- A Bac+5, nous ne corigeons plus les erreurs de “;” manquant et autre trivialités.
  - Le travail se fait sur vos propres PC. Merci de venir en séance avec les logiciels installés et testés.
  - La sauvegarde régulière est VOTRE problème.
  - Nous suggérons fortement l’usage d’un logiciel de gestion de version comme git.

## 2 Présentation générale du projet

Il s'agit d'un petit utilitaire permettant d'afficher quelques données relatives à l'aviation de tourisme. Au cours de ces séances, on va utiliser une source de données statiques (la liste des aéroports mondiaux au format .csv), et une source de données dynamiques (une API sur le site [aviationstack.com](http://aviationstack.com)).

La structure UML de notre application est la suivante :

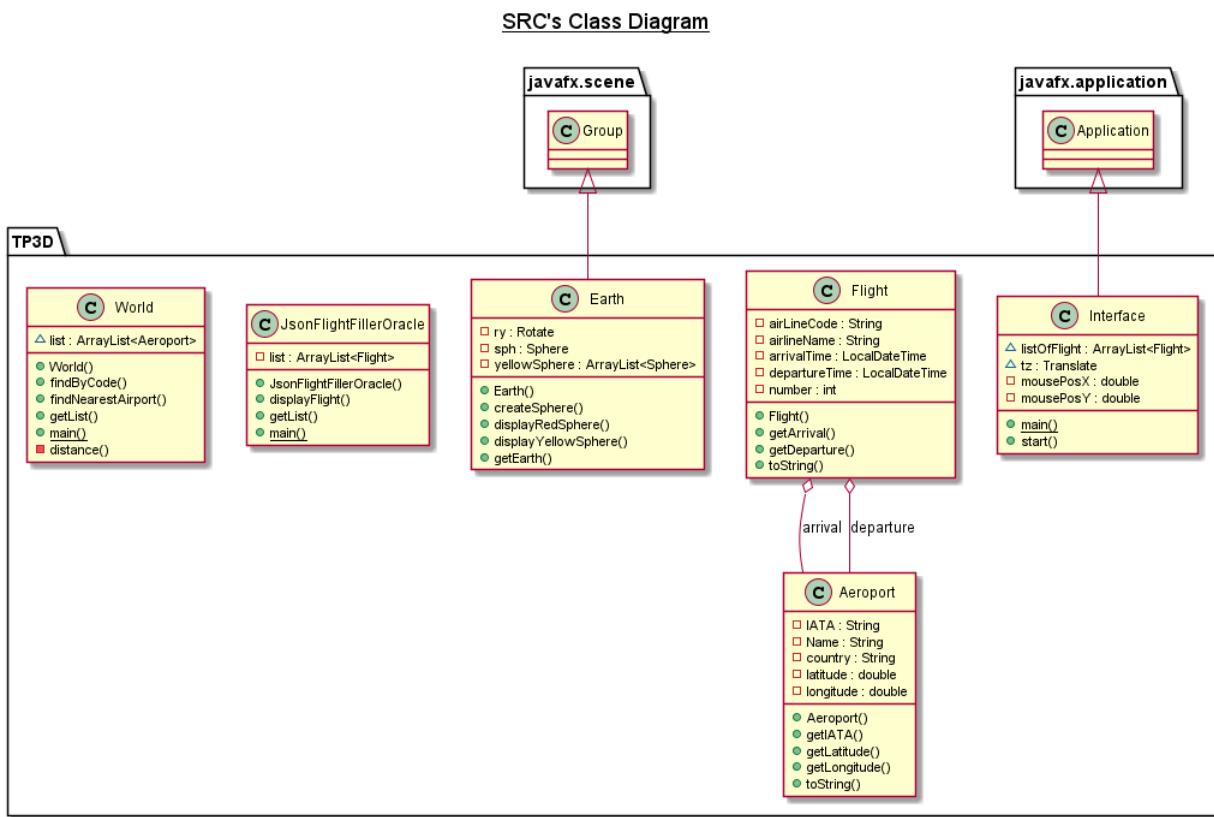


Figure 1: Le diagramme de classe du projet complet

La liste ci-dessous est dans l'ordre dans laquelle nous construirons les classes :

- La classe Aeroport définit ce qu'est un aéroport du point de vue de l'application : un nom, des coordonnées GPS nommées latitude et longitude, et un code IATA.
- La classe World contient la liste de tous les aéroports du monde entier. Elle est créée par la lecture d'un fichier CSV. Cette classe offre deux moyens de retrouver un aéroport : par son code, qui doit être exacte, ou par les coordonnées GPS (le système renvoie l'aéroport le plus proche quelque soit la distance réelle).
- La classe Interface contient l'interface du projet. Elle sera complétée au fur et à mesure. Elle contient entre autre un objet de groupe "Earth" et des méthodes pour gérer l'interactivité. En cas de clic sur le globe terrestre, elle calcul l'aéroport le plus proche, l'affiche en rouge, recherche sur Internet tous les vols en direction de cet aéroport, et affiche en jaune les aéroports d'origines. Attention, malgré son nom, la classe Interface n'est pas une interface, mais une ihm...
- La classe Flight modélise un vol. Un vol est notamment défini par deux Aéroports, de départ et d'arrivée.
- La classe JsonFlightFillerOracle utilise un paquet d'analyse JSON fournis par Oracle pour interroger une base en ligne et récupérer tous les vols.
- La classe Earth hérite de la classe groupe (en effet, elle est constituée de la terre et des aéroports concernées).

### 3 Première session : classes de bases et lecture .csv

Nous travaillerons dans un projet Java standard, de type Java 16 (11 minimum pour l'usage de JavaFX et des paquets de lecture JSON).



- Créez la classe "Aéroport" comportant les éléments du graphe UML.
- Surchargez la méthode `toString` pour nos tests.
- Testez votre classe.

Pour la suite, vous allez avoir besoin du fichier .csv contenant la liste des aéroports. Ce fichier est disponible sous moodle. Il provient du site "aviationstack", mais je l'ai édité pour une lecture plus simple. Je suggère d'enregistrer ce fichier dans un répertoire "data" du projet.



- Créez la classe "World".
- Créez d'abord son constructeur qui prend comme paramètre une chaîne de caractère pointant vers le fichier contenant la liste des aéroports au format ".csv". Vous pouvez vous aider des éléments ci-dessous. Limitez les aéroports à ceux étant des "large-airport".

Vous trouverez ci-dessous un extrait du fichier `airport-codes_no_comma.csv`. Vous remarquerez que tous les champs ne sont pas systématiquement remplis. Nous aurons besoin de plusieurs champ. Le champ "0" contient le code IATA, qui nous servira à retrouver notre aéroport. De même, le champ "1" contient la nature de l'aéroport.

Le listing ci-dessous est fait pour vous guider dans la construction de la liste d'aéroport par lecture du fichier csv. Vous aurez aussi besoin de la méthode `parseDouble` pour transformer une chaîne de caractère en `double`.

Pour trouver l'aéroport le plus proche, nous allons avoir besoin d'une norme de proximité. Pour nous, si l'on nomme  $\Theta$  la latitude et  $\Phi$  la longitude, on utilisera la norme suivante entre deux aéroports 1 et 2.

$$norme = (\Theta_2 - \Theta_1)^2 + \left( (\Phi_2 - \Phi_1) \cos\left(\frac{\Theta_2 + \Theta_1}{2}\right) \right)^2$$

---

```

1 AYPW,small_airport,Panakawa Airport,42,0C,PG,PG-WPD,,AYPW,,,"143.124722222, -7.67205555556",,
2 AYPY,large_airport,Port Moresby Jacksons International Airport,146,0C,PG,PG-NCD,Port Moresby,AYPY,POM,,,"147.22000122070312, -9.
3 AYQB,small_airport,Simbari Airstrip,3560,0C,PG,PG-EHG,Simbari,AYQB,,SBR,"145.6448, -6.9623",,

```

---

Listing 1: Extrait du fichier airport-codes\_no\_comma.

---

```

public World (String fileName){
    try{
        BufferedReader buf = new BufferedReader(new FileReader(fileName));
        String s = buf.readLine();
        while(s!=null){
            s=s.replaceAll("\\"", "");
            //Enlève les guillemets qui séparent les champs de données GPS.
            String fields[] = s.split(",");
            // Une bonne idée : placer un point d'arrêt ici pour du debuggage.
            if (fields[1].equals("large_airport")){
                // A continuer
            }
            s = buf.readLine();
        }
    }
    catch (Exception e){
        System.out.println("Maybe the file isn't there ?");
        System.out.println(list.get(list.size()-1));
        e.printStackTrace();
    }
}

```

---

Listing 2: Aide pour la création du constructeur de la classe World.

**Remarque :** il ne s'agit pas de la vraie distance, mais ce calcul devant se dérouler pour chaque aéroport, on l'a simplifié. Il faudrait notamment prendre la racine carrée de cette norme et multiplier ensuite par le rayon de la terre.



- Complétez la méthode distance au sein de la classe World avec la formule ci-dessus.
- Complétez la méthode findNearest au sein de la classe World. Cette fonction calcul toutes les distances possibles et détermine l'aéroport le plus proche.
- Complétez la méthode findByCode qui cherche un code précis dans la liste.
- Testez avec le code à l'aide du main ci-dessous. L'aéroport le plus proche de Paris (latitude : 48.866, longitude 2.316) est Orly (Code IATA ORY).

---

```

public static void main(String[] args){
    World w = new World ("./data/airport-codes_no_comma.csv");
    System.out.println("Found "+w.getList().size()+" airports.");
    Aeroport paris = w.findNearestAirport(2.316,48.866);
    Aeroport cdg = w.findByCode("CDG");
    double distance = w.distance(2.316,48.866,paris.getLongitude(),paris.getLatitude());
    System.out.println(paris);
    System.out.println(distance);
    double distanceCDG = w.distance(2.316,48.866,cdg.getLongitude(),cdg.getLatitude());
    System.out.println(cdg);
    System.out.println(distanceCDG);
}

```

---

Listing 3: Test de la classe World.

## 4 Travail à la maison

La prochaine section va consister à proposer une IHM (interface homme-machine) utilisant JavaFx.<sup>1</sup>. Nous allons donc ici modifier notre setup pour utiliser ce framework.

Dans le framework JavaFx, une application doit obligatoirement hériter d'une super classe de type Application. Une application JavaFx aura donc la structure de base suivante :

---

```

import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class Interface extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception{

        primaryStage.setTitle("Hello world");
        Group root = new Group();
        Pane pane = new Pane(root);
        Scene theScene = new Scene(pane, 600, 400,true);
        primaryStage.setScene(theScene);

        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}

```

---

Listing 4: Un "Hello world" en JavaFx

Nous avons ici quelques concepts de bases de JavaFx : une scène ("Stage" en Anglais, ne pas confondre avec la Scene) correspond à la fenêtre. Pour une application multifenêtre nous aurions besoin de plusieurs Stages.

---

<sup>1</sup> Attention, à l'heure actuelle le moteur de rendu 3D de JavaFx n'est pas disponible sous Linux

Une Scene (Hum... No comment) correspond à un affichage à un moment donnée. Il est facile de passer d'une scène à l'autre avec la méthode SetScene appliquée à un "Stage". Vous devez donc construire votre Scene.

La Scene contient des groupes. Un groupe est un ensemble d'éléments (Nodes en Anglais) auxquels on peut appliquer des transformations géométriques. Dans notre TP, la classe Earth correspondra à un groupe au sein duquel il y aura la Terre et la représentation des aéroports.

La figure ci-dessous illustre le lien entre stage, scene et nodes.

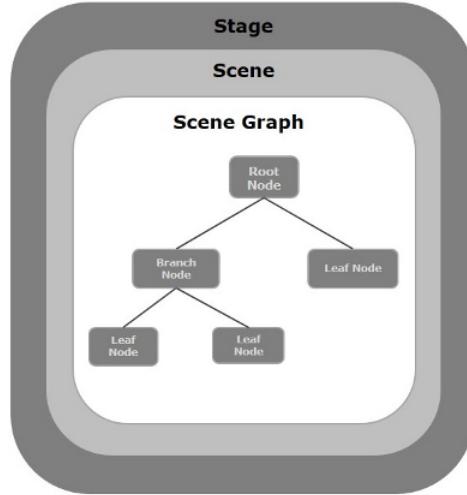


Figure 2: Java Fx Application structure

La figure ci-dessous indique l'ensemble des nodes disponibles.

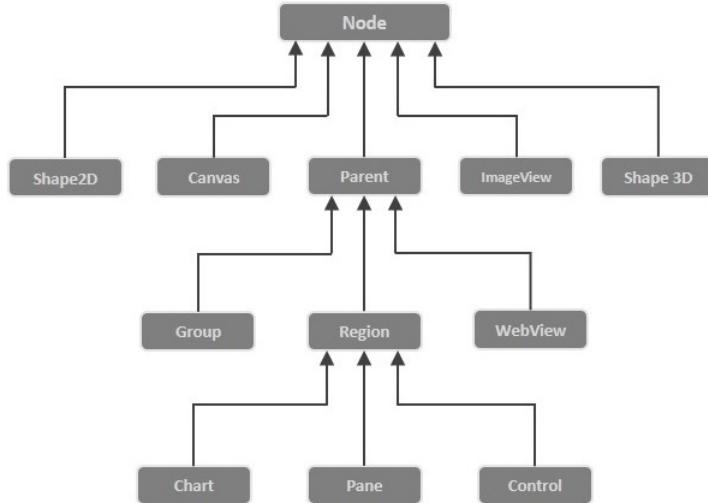


Figure 3: Nodes relation in JavaFx



## Tutoriel sur le déploiement d'une application JavaFx avec IntelliJ

- Téléchargez en ligne la bibliothèque JavaFx. Attention à ne pas vous trompez de version d'OS et de processeur. Dézippez le fichier dans un répertoire "facile à retrouver". Copier / Coller dans un fichier texte le chemin du répertoire "/lib" où se trouve les fichiers d'extension .jar.
- Conservez votre projet avec les classes World et Aéroport.
- Créez dans votre projet la classe Interface dont le code est ci-dessus. Toutes les références à JavaFx vont apparaître en rouge, car IntelliJ ne les trouve pas.
- Ouvrez le menu "Project Structure". Ajouter le répertoire "/lib" noté précédemment dans les chemins de bibliothèque. Vous devriez alors avoir une compilation sans erreur (mais pas une exécution). Corrigez les erreurs que vous pourriez trouver.
- Lancer le main de la classe Interface pour créer une configuration d'exécution (qui apparaîtra en haut et à droite de l'écran).
- Allez dans le menu "Edit configuration". Cliquez sur Modify Option / Add VM Option.
- Ajoutez dans "VM Option" les deux lignes suivantes : `--module-path "Your /lib library" --add-modules javafx.controls`.
- Vous devriez maintenant obtenir une fenêtre vide lors de l'exécution.

## 5 Deuxième partie : IHM

Nous allons maintenant afficher un modèle de la Terre tournant sur elle-même. Lorsque l'utilisateur cliquera sur un point de la Terre, on affichera dans la console l'aéroport le plus proche.

La classe Earth étend donc la classe Group. Elle contient un objet sph de nature Sphere de rayon **300 pixels**.



- Créez la classe Earth avec son constructeur.
- N'oubliez pas d'ajouter l'objet sph au groupe... par exemple avec la ligne `this.getChildren().add(sph);`
- Changez dans la classe Interface le Group root par un objet Earth.
- On ne voit presque rien... Que se passe-t-il ?

L'usage de la 3D impose d'avoir un objet de type "Caméra". Il faut reculer notre Caméra d'au moins 300 pixels puisque notre sphere fait 300 pixels de rayon. SetNearClip et SetFarClip indique les distances auxquelles les objets ne sont plus affichés. setFieldOfView permet de fixer l'angle de vision, en degré.

Nous allons ajouter cette caméra avec le code suivant (ihm est le nom de la Scene) :

```
PerspectiveCamera camera = new PerspectiveCamera(true);
camera.setTranslateZ(-1000);
camera.setNearClip(0.1);
camera.setFarClip(2000.0);
camera.setFieldOfView(35);
ihm.setCamera(camera);
```

Listing 5: Ajout d'une caméra de type PerspectiveCaméra

Nous allons maintenant ajouter une fonctionnalité de zoom, sur un seul axe, tout simplement en déplaçant notre caméra. Dans la classe Interface, nous allons donc commencer à coder ce qui concerne les interactions avec l'utilisateur. Cela se fait au moyen d'une lambda-expression ressemblant à cela :

---

```
ihm.addEventHandler(MouseEvent.ANY, event -> {
    if (event.getEventType() == MouseEvent.MOUSE_PRESSED) {
        System.out.println("Clicked on : (" + event.getSceneX() + ", " + event.getSceneY() + ")");
    }
    if (event.getEventType() == MouseEvent.MOUSE_DRAGGED) {
        camera.getTransforms().add(...);           // A vous de compléter
    }
});
```

---

Listing 6: Ajout d'un EventHandler sur le clique gauche



- Grâce aux informations ci-dessus, ajoutez une caméra à votre Scène.
- Testez la visualisation.
- Ajoutez un EventHandler qui capture la valeur de la position du clique en X et en Y.
- Ajoutez une transformation qui modifie la translation de la caméra sur l'axe Z lorsque la position de la souris est modifiée avec le bouton enfoncé.
- Testez le mouvement de caméra.

Notre prochaine étape consiste à rajouter une texture sur la sphère sph qui symbolise la Terre. Cela se fait en 3 lignes à l'aide d'une classe nommée PhongMaterial. L'image à charger (représentée ci-dessous), a une haute résolution pour que le zoom ne soit pas trop problématique. Elle est disponible sous Moodle.



- Dans la classe Earth, implémentez un nouvel objet PhongMaterial.
- Avec la méthode setDiffuseMap mappez l'image sur votre matériaux.
- Avec la méthode setMaterial, mappez le matériaux sur votre sphère.
- (A faire si vous avez le temps). Remplacez la sphère par un cube, pour voir...

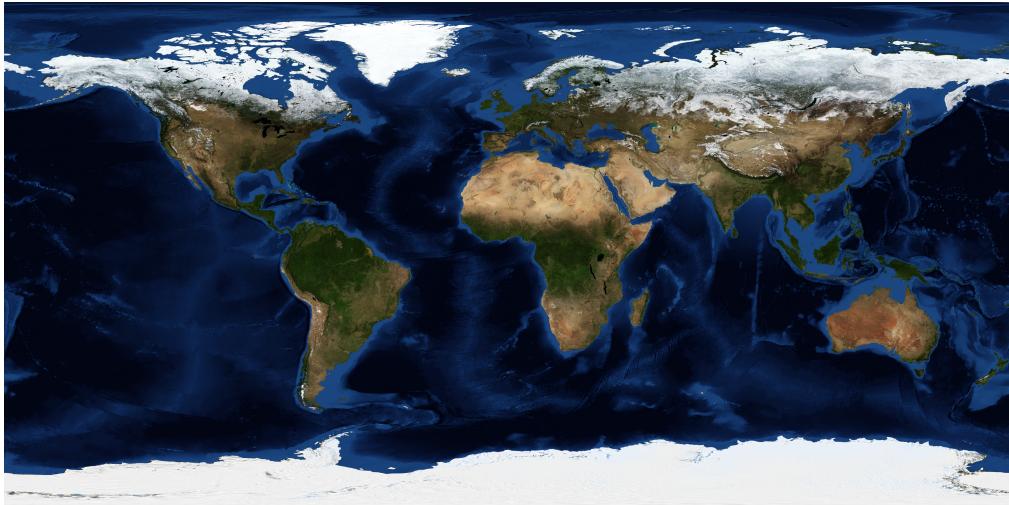


Figure 4: La texture de notre Sphere : la Terre.

Nous allons maintenant mettre en rotation la Terre. Pour cela, nous allons faire appel à deux classes différentes. La classe AnimationTimer permet de surcharger une méthode handle qui s'exécute automatiquement toutes les 16 millisecondes. Puisque l'on est dans une autre classe, on ne peut pas employer this... Du coup, on va définir un objet Rotate qui correspond à une rotation. Par défaut, le centre de rotation est le point de coordonnées (0,0,0), ce qui nous va très bien. On va chercher à tourner autour de l'axe y.

---

```
AnimationTimer animationTimer = new AnimationTimer() {
    @Override
    public void handle(long time) {
        System.out.println("Valeur de time : " + time);
        ry.setAngle(...);           // A compléter
    }
};
animationTimer.start();
```

---

Listing 7: Ajout d'un EventHandler sur le clique gauche



- Dans le constructeur de la classe Earth, modifiez l'objet ry (classe Rotate) pour obtenir le bon axe de rotation.
- Dans le constructeur de la classe Earth, ajouter un AnimationTimer comme ci-dessus.
- Pour que l'angle varie avec le temps, utiliser le paramètre time (en nanoseconde) de manière à faire un tour en 15 secondes.

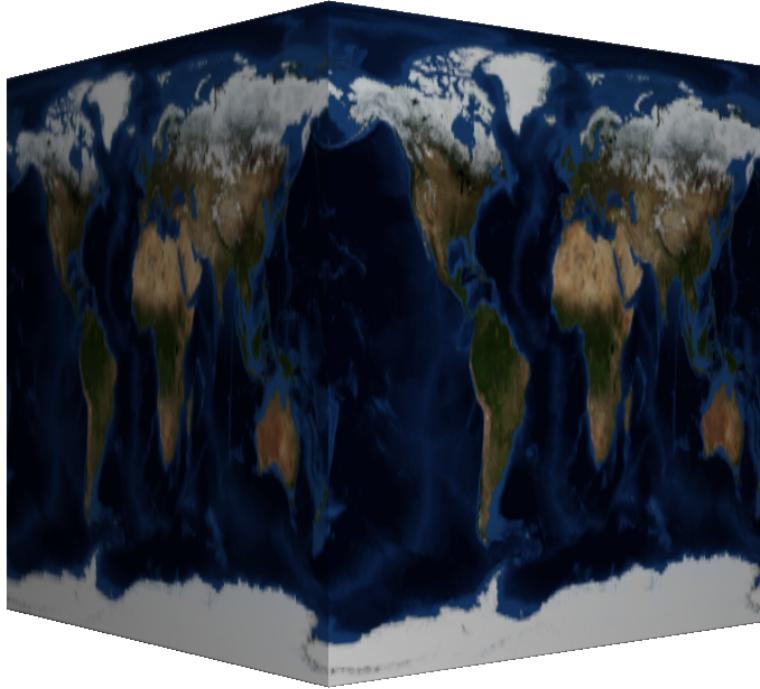


Figure 5: Une image pour faire plaisir aux plattistes.

On approche de la fin, courage !

Il faut maintenant récupérer les coordonnées en latitude et longitude d'un clique droit de l'utilisateur. Il va y avoir un peu de magie dans l'air...

En effet, s'il est aisément de récupérer les coordonnées d'un clique droit sur un objet 3D grâce à la classe PickResult, il n'est pas évident de le convertir en latitude et longitude. Pourquoi ? Parce que la projection de Mercator employé sur notre image tend à diverger vers l'infini pour les hautes latitudes.

Voyons d'abord comment récupérer ces coordonnées en (x,y) au sein de la classe Interface :

---

```

ihm.addEventHandler(MouseEvent.ANY, event -> {
    if (event.getButton() == MouseButton.SECONDARY && event.getEventType() == MouseEvent.MOUSE_CLICKED) {
        PickResult pickResult = event.getPickResult();
        if (pickResult.getIntersectedNode() != null) {
            // Code à compléter : on récupère le point d'intersection
            // Conversion en longitude et latitude
            // Recherche dans l'objet w de la classe World de l'aéroport le plus proche.
            // Affichage dans la console
        }
    }
})

```

---

Listing 8: Ajout d'un EventHandler sur le clique droit

Le passage aux coordonnées classiques se fait par les formules suivantes :

$$\left[ \begin{array}{l} \Theta = 2 \arctan \left( e^{\frac{(0.5-Y)}{0.2678}} \right) - 90 \\ \Phi = 360 * (X - 0.5) \end{array} \right]$$

Autant la formule de la longitude est totalement logique si l'on sait que la méthode getPickResult renvoie une fraction entre 0 et 1 de la texture et que l'image commence à la longitude -180, autant la formule de la longitude fait apparaître un nombre magique, 0.2678, que j'ai obtenu par essai-erreur sur une série de capitale. Cette formule donne des résultats très acceptables pour des latitudes inférieures à 60 degrés.

Attention lors de l'implémentation, la méthode Math.atan renvoie un angle en radians en Java.



- Dans la classe Interface, ajouter un EventHandler qui en cas de clique droit affiche dans la console l'aéroport du point le plus proche cliqué.
- Vous aurez peut être besoin du debugger...

La cerise sur le gâteau...

On veut maintenant être capable d'ajouter une sphère colorée à l'emplacement de nos aéroports. Cette sphère aura les coordonnées suivantes (Ou R est le rayon de la Terre) :

$$\begin{bmatrix} X = R.\cos(\Theta - 13).\sin(\Phi) \\ Y = -R\sin(\Theta - 13) \\ Z = -R.\cos(\Theta - 13).\sin(\Phi) \end{bmatrix}$$

Le facteur 13 est encore une correction empirique de la latitude.



- Dans la classe Earth, ajoutez une méthode `public Sphere createSphere(Aeroport a, Color color)` qui renvoie une sphère de rayon 2 et de couleur color.
- Dans la classe Earth, ajoutez une méthode `public void displayRedSphere(Aeroport a)` qui fait appel à la méthode précédente avec le paramètre `Color.RED` et ajoute la sphère en question à l'ensemble du groupe Earth (usage de la méthode `getChildren()`...)
- Affichez une boule rouge sur l'aéroport le plus proche du clique droit de l'utilisateur.

Ouf, cette longue section est enfin finis !

## 6 Troisième partie : accès à des données "live"

Le site [aviationstack.com](http://aviationstack.com) propose une API permettant de suivre en live les vol à destination ou au départ d'un aéroport.

**Manipulation à faire chez vous.**



- Lisez la documentation de cet API
- Obtenez une access\_key gratuite. Attention à ne pas surabuser de la mienne.
- Effectuez via un navigateur la requête suivante (remplacer ma clé par la vôtre) : [http://api.aviationstack.com/v1/flights?access\\_key=cfaf27d3b7c76c08baf0e49ddb0df72c&arr\\_iata=CDG](http://api.aviationstack.com/v1/flights?access_key=cfaf27d3b7c76c08baf0e49ddb0df72c&arr_iata=CDG)
- Faites un schéma du format de la réponse.

Pour le début de ce développement, un fichier `test.txt` est disponible contenant une requête sur l'aéroport d'Orly. Cela limite les temps de chargement en ligne.

Nous allons construire une nouvelle classe `JsonFlightFiller` reposant sur une classe `Json` fournis par Oracle (il y en a beaucoup d'autre, notamment `gson` fournis par google, mais celle-ci est la plus simple à mon sens).



- Télécharger sur Moodle la bibliothèque Json.
- Ajouter dans "Project Structure" cette bibliothèque.
- Créez une classe "Flight" correspondant au diagramme UML.
- Créez le squelette de la classe JsonFlightFiller contenant un seul objet de type `private ArrayList<Flight> list = new ArrayList<Flight>();`

Voici un extrait de code pour vous guider dans la mise en place de cette analyse :

```
public JsonFlightFiller(String jsonString,World w){  
    try {  
  
        InputStream is = new ByteArrayInputStream(jsonString.getBytes(StandardCharsets.UTF_8));  
        JsonReader rdr = Json.createReader(is);  
        JsonObject obj = rdr.readObject();  
        JSONArray results = obj.getJSONArray("data");  
        for (JsonObject result : results.getValuesAs(JsonObject.class)) {  
            try {  
                /*****  
                * A vous de jouer !  
                *****/  
            catch(Exception e){  
                e.printStackTrace();  
            }  
        }  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

Listing 9: Un morceau du constructeur de la classe JsonFlightFiller.



- Modifiez le constructeur selon l'exemple ci-dessus.
- Générez autant d'objet "Flight" que d'enregistrement.
- Testez et validez avec le main ci-dessous.

---

```

public static void main (String[] args){
    try {
        World w = new World ("./data/airport-codes_no_comma.csv");
        BufferedReader br = new BufferedReader(new FileReader("data/test.txt"));
        String test = br.readLine();
        JsonFlightFiller jSonFlightFiller = new JsonFlightFiller(test,w);
    }
    catch (Exception e){
        e.printStackTrace();
    }
}
}

```

---

Listing 10: Test de la classe JsonFlightFiller avec un fichier à plat.

Il ne nous reste plus qu'à interroger la base de donnée en ligne. Pour cela il va falloir lire de la doc... du côté des classes HttpClient, HttpRequest et HttpResponseMessage. On va pas tout vous donner quand même !



- Après l'affichage de la boule rouge dans le code de la classe Interface, ajouter l'interrogation via HttpRequest de l'API du site web. Sa réponse est analysée à l'aide de la classe JsonFlightFiller et la liste des vols fait apparaître les aéroports de départ sous la forme d'une liste de boules jaunes, affiché par la méthode displayYellowBall.

## 7 Un petit dernier pour la route

Comment se passer du freeze de l'animation lors de la requête API http ? Il faut mettre cette partie dans un processus à part, donc une classe "Runnable". A vous d'imaginer la suite...

## 8 Conclusion

Nous espérons que ce rapide voyage dans le domaine de la programmation Objet vous a convaincu. Les principaux concepts à retenir sont les suivants :

- 

## List of Figures

1	Le diagramme de classe du projet complet.	1
2	Java Fx Application structure	5
3	Nodes relation in JavaFx	5
4	La texture de notre Sphere : la Terre.	8
5	Une image pour faire plaisir aux plattistes.	9

## List of Listings

1	Extrait du fichier airport-codes_no_comma.	3
2	Aide pour la création du constructeur de la classe World.	3
3	Test de la classe World.	4
4	Un "Hello world" en JavaFx	4
5	Ajout d'une caméra de type PerspectiveCaméra	6
6	Ajout d'un EventHandler sur le clique gauche	7
7	Ajout d'un EventHandler sur le clique gauche	8

8	Ajout d'un EventHandler sur le clique droit . . . . .	9
9	Un morceau du constructeur de la classe JsonFlightFiller. . . . .	11
10	Test de la classe JsonFlightFiller avec un fichier à plat. . . . .	12