Projet IGSD

Jacobo Ruiz Ocampo / Alexis Delplace

Introduction

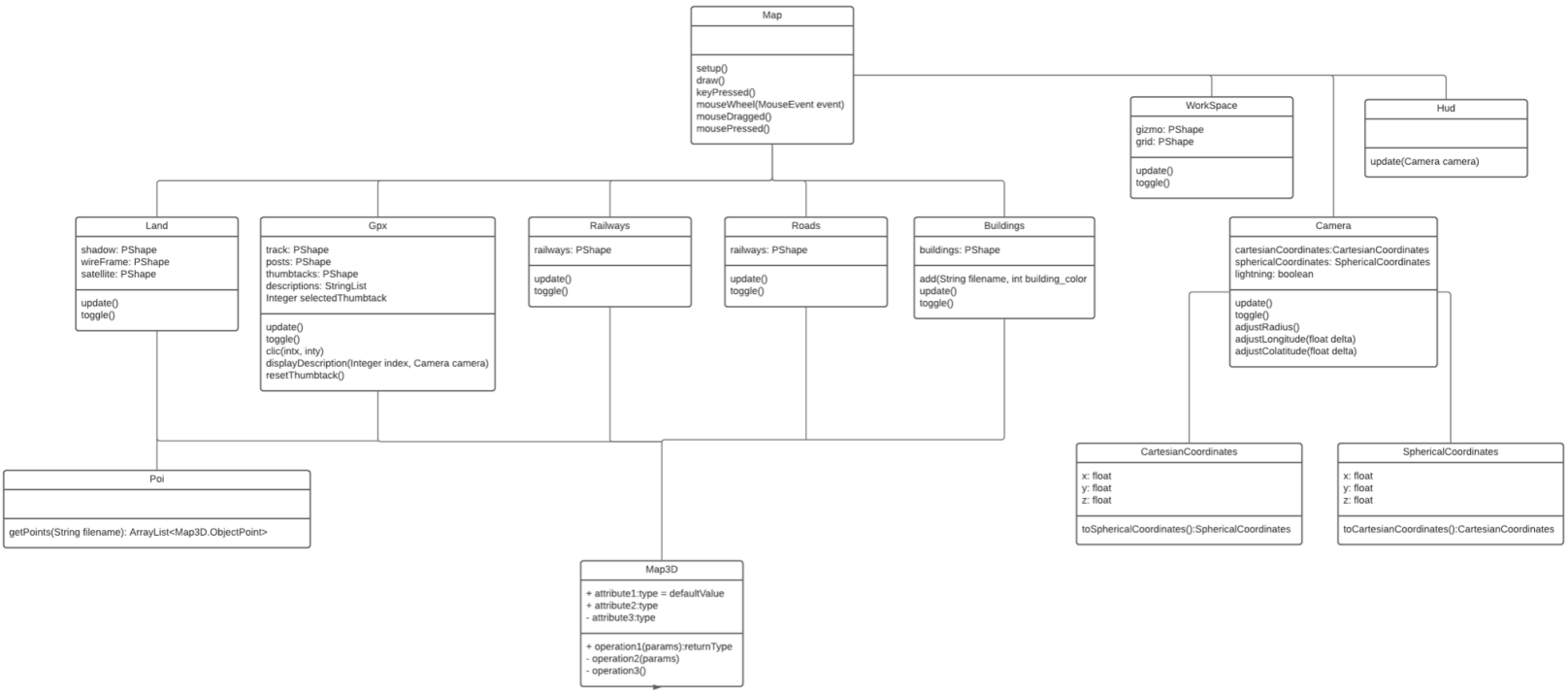
Dans le cadre du projet guidé d'IGSD de fin de semestre, nous avons codé en Processing une représentation 3D de l'université Paris Saclay.

Analyse du problème

Pour débuter le projet, nous avons dû dans un premier temps mettre en place l'environnement de travail ainsi que la caméra. Ensuite, nous avons exploité les données liées au relief, aux autoroutes, aux chemins de fer et aux bâtiments des alentours de l'université et les représenter en 3D via Processing.

Afin de réaslier ce projet à 2, nous avons utilisé github.

Développement du projet



En règle général, toutes les classes qui permettent d'afficher des objets (PShape) à l'écran possèdent les méthodes suivantes :

`update()` permet d'afficher à l'écran notre PShape, cette méthode est appelée dans `drawn()` de manière à actualiser l'affichage du PShape en question.

```
void update() {
   shape(this.myShape);
 }
```

`toggle()` permet d'activer ou désactiver l'affichage de notre PShape à l'écran, cette méthode est appelée lorsque l'on appuie sur une certaine touche via la méthode de processing `keyPressed()`.

```
void toggle() {
   this.myShape.setVisible(!this.myShape.isVisible());
 }
```

Workspace :

Cette classe nous permet de construire les PShape `gizmo` et `grid`.

Camera :

Cette classe gère la position de la caméra ainsi que les paramètres de lumière. Le principal challenge ici, était que nous voulions que notre caméra soit positionnée à la surface d'une sphère virtuelle d'origine (0,0,0) et de rayon radius variable. D'un autre côté, Processing utilise un système de coordonnées cartésiennes. Il fallait donc passer d'un système à l'autre, pour cela, on a implémenté 2 classes :

- `CartesianCoordinates`

  Cette classe possède les attributs x, y, z représentant les coordonnées cartésiennes ainsi qu'une méthode `toSphericalCoordinates()` qui permet de passer du système de coordonnées cartésiennes au système de coordonnées polaire.

- `SphericalCoordinates`

  Cette classe possède les attributs radius, longitude, colatitude représentant les coordonnées sphériques ainsi qu'une méthode `toCartesianCoordinates()` qui permet de passer du système de coordonnées polaire au système de coordonnées cartésiennes.

La classe caméra a donc les attributs `cartesianCoordinates` et `sphericalCoordinates` qui sont des instances des deux classes vu précédement.

Pour déplacer la caméra, on utilise les méthodes :

- `adjustRadius(float offset)`
- `adjustLongitude(float delta)`
- `adjustColatitude(float delta)`

Elles permettent de modifier `sphericalCoordinates`, ensuite pour que ces modifications soient prises en compte, dans la méthode `update()` de Camera, on met à jour `cartesianCoordinates` via `sphericalCoordinates` en utilisant `toCartesianCoordinates()`. On passe ensuite les x, y, z de `cartesianCoordinates` à la méthode de Processing `camera()`.

Hud

La classe Hud permet d'afficher à l'écran les informations sur notre position et le nombre de fps. Pour afficher tout ça, on fait des rectangles via `rect()` pour l'arrière-plan auquel on superpose du texte via la méthode `text()`.

Ce qui était important ici, c'était de mémoriser la matrice de transformation initiale via la méthode `begin()` fournie dans le projet et puis de rétablir la matrive via `end()`. Cette manipulation est importante puisqu'elle permet d'inhiber les éclairages et d'inhiber le test de profondeur du pipeline OpenGL de sorte que les informations soient toujours visibles en avant-plan.

Land

Cette classe construit les PShape suivant:

- `shadow` : Ombre rectangulaire sous la carte
- `wireFrame` : Maillage fillaire représentant le relief
- `satellite` : Image satellite avec le relief

Pour construire `shadow`, c'est assez simple, il suffit de construire un PShape avec 4 sommets, un pour chaque extrémité de notre carte.

Pour `wireFrame` et `satellite`, c'est plus compliqué, il faut pour chaque point de notre quadrillage créer un carré, vu que notre PShape est de type `QUAD`, il nous suffit d'ajouter 4 sommets à notre PShape via `vertex()` afin de former un carré.

Pour avoir du relief, il faut pour chaque point du quadrillage préciser le z approprié, pour récupérer la valeur de ce z, on utilise la classe `ObjectPoint` fournie dans la bibliothèque `Map3D` qui permet d'obtenir la hauteur d'un point en fonction de ses coordonnées x, y.

À ce stade, on a un joli quadrillage avec du relief donc pour `wireFrame`, c'est bon. En revanche, on a toujours pas de texture pour `satellite`. Pour ajouter cette texture, il faut lorsque ce que l'on fait appel à `vertex()` en plus de passer en paramètre x, y, z donner des valeurs pour u et v qui permettent d'appliquer une texture2D (une image) à une forme 3D (notre PShape) et ce en concervant les proportions.

Gpx

Cette classe construit les PShape suivants :

- `track` : Le tracé violet sur la route représentant l'itinéraire
- `posts` : Les "poteaux" des waypoints
- `thumbtacks` : Les "punaises" (sphères) des waypoints.

Ce qui était compliqué ici, c'était de garder afficher la description du waypoint sur lequel on avait cliqué. Pour la couleur des punaises, il suffisait de changer la couleur du Pshape thumbtacks associé au waypoint en question via la méthode `setStroke()` de Processing.

Pour ce qui est de la description, il n'y a pas de manière évidente de le faire comme vu ci-dessus pour la couleur. Nous avons donc implémenté cette fonctionnalité comme ceci :

- Les descriptions de chaque point sont récupérées lors de la lecture du gson puis sauvegardées dans la StringList `descriptions`. Ainsi `descriptions.get(1)` correspond à la description du waypoint n°1.
- Lors de l'appel à la fonction `click()` si on clique sur un waypoint alors son index est sauvegardé dans la varriable `selectedThumbtack`.
- Enfin, lors de l'appel à `update()`, si `selectedThumbtack` n'est pas `null` (c'est à dire que l'on a cliqué sur un waypoint) alors on affiche à l'écran la description du waypoint ayant pour index `selectedThumbtack` via notre méthode `displayDescription(Integer index, Camera camera)`

Railways, Roads

Pour ces 2 classes, le procédé est le même :

- On crée un PShape de type GROUP

- On extrait les données du GeoJSON pour construire les PShapes. Dans les 2 cas, les fichiers GeosJSON contiennent des "`LineString`" qui sont des listes de coordonnées représentant des portions de route ou de voie ferrée.

Buildings

Pour la classe buildings, c'est toujours le même procédé que les 2 classes ci-dessus, en revanche dans ce cas, nous avions pour chaque bâtiment une liste de coordonnées représentant le sol du bâtiment ainsi que le nombre d'étages du bâtiment.

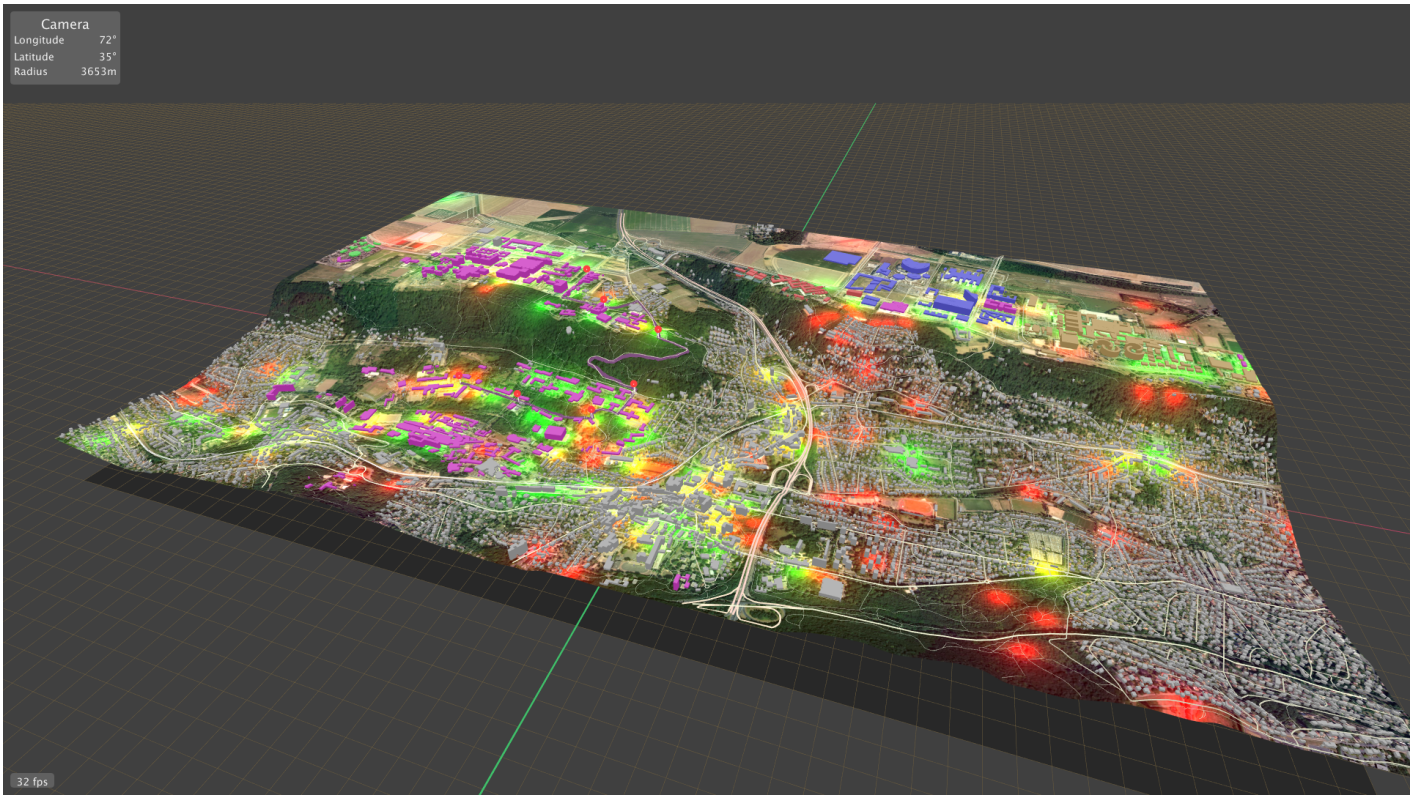
Poi et Shader

La classe Poi permet de récupérer sous forme d'`ArrayList<Map3D.ObjectPoint>` les coordoonées des points d'interêts du fichier que l'on passe en paramètre comme par exemple "`bicycle_parking.geojson`".

Ensuite, pour chaque point de la carte, on attribue la distance entre ce point et le point et le point d'interêts le plus proche. Les distances sont stockées dans le fichier "`poi_distances.json`". Si ce fichier est supprimé alors les distances sont re-calculées et un nouveau fichier `json` est créé, cela permet d'optimiser le programme afin qu'il ne calcul pas toutes les distances à chaque éxécution.

Enfin grâce au shader on colorie chaque point de la carte en fonction des distances attribuées précedement.

Résultat



Conclusion

Grâce à ce projet, nous avons désormais les connaissances techniques qui nous permettront dans nos futurs projet de réaliser des représentations 3D via Processing.