Méthode d’optimisation

Hirtz / Naigeon / Mellano / Reveillard

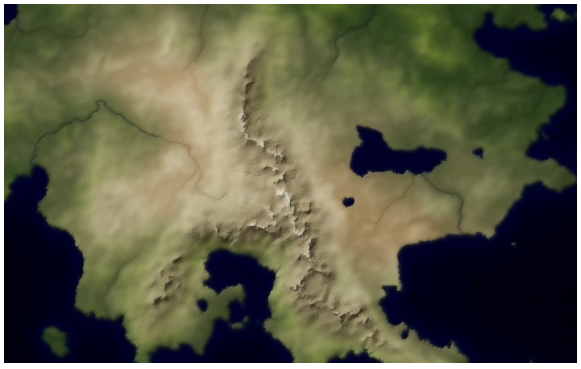
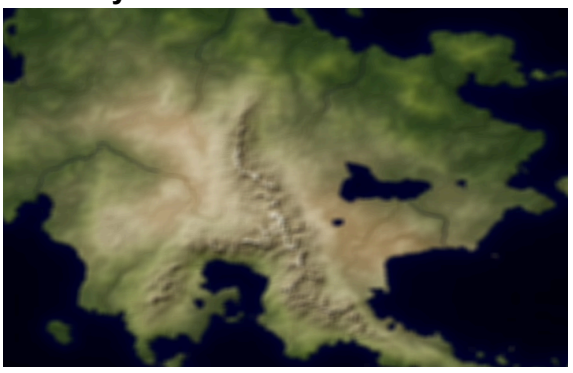
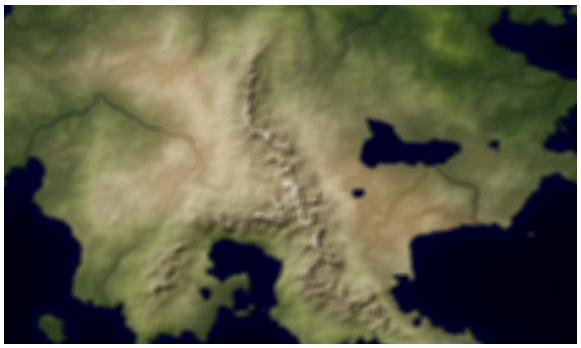
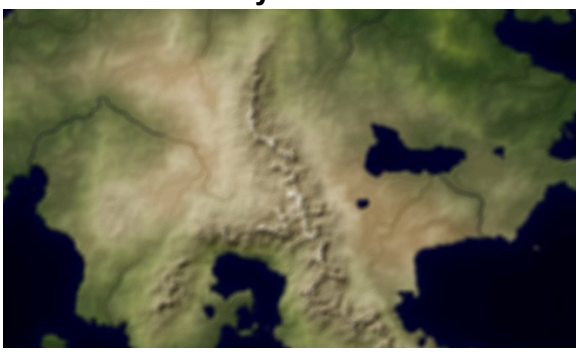
I - Le Flou

Nous avons développé plusieurs algorithmes de floutage :

- **BilateralFilter (Filtre bilatéral)**
- **BlurAverage (Flou moyen)**
- **CircularBlurAverage (Flou circulaire moyen)**
- **GaussianBlur (Flou gaussien)**

Lors des phases de test, nous avons constaté que le filtre bilatéral offrait les meilleurs résultats pour flouter une image. En effet, ce filtre utilise un lissage non linéaire qui permet de préserver les contours tout en réduisant le bruit de l'image. Il réalise une moyenne pondérée des pixels voisins, en ignorant ceux qui n'ont pas la même couleur. Ainsi, il maintient les détails de l'image avec une bonne qualité mais surtout pour notre projet, il conserve les bordures. C’est pour cela que nous avons choisi cet algorithme.

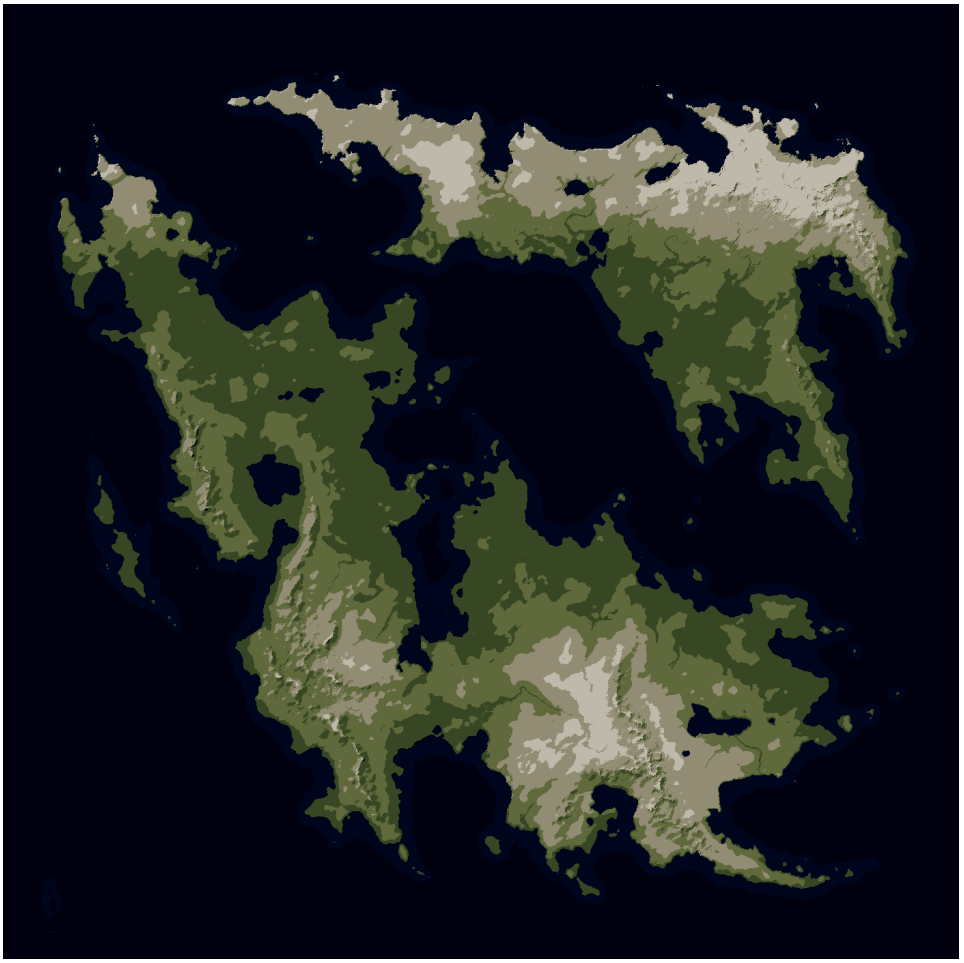
Pour démontrer la supériorité du filtre bilatéral, voici des exemples d'images floues obtenues avec les différents algorithmes.

Filtre Bilatéral : 	**Flou Moyenne :** 
Flou Gaussien : 	**Flou Circulaire Moyenne :** 

II - Biomes

Pour déterminer les différents biomes, on utilise l'algorithme K-means. C'est un algorithme réputé pour trouver des zones de couleurs, car elles sont généralement sous forme de sphères, et l'algorithme détermine les différentes zones à partir de leurs centres appelés centroïdes.
L'algorithme associe chaque donnée à son centroïde le plus proche pour créer des clusters.

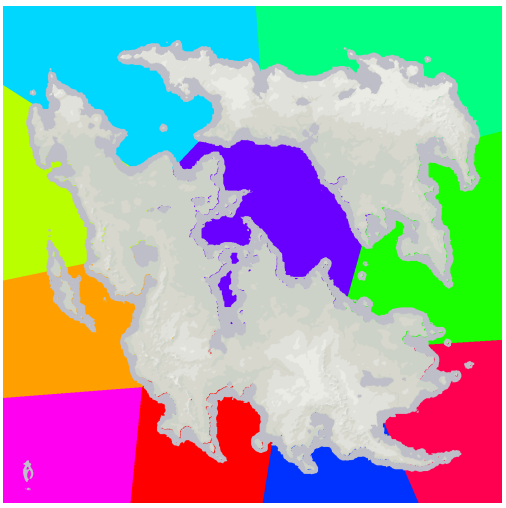
K-means permet de détecter automatiquement la palette. Pour cela on l'associe à une fonction de coût. Cela va nous permettre d'évaluer la qualité des palettes et ainsi déterminer la meilleure palette.
On obtient alors une image avec un certain nombre de couleurs différentes permettant de distinguer facilement les différents biomes

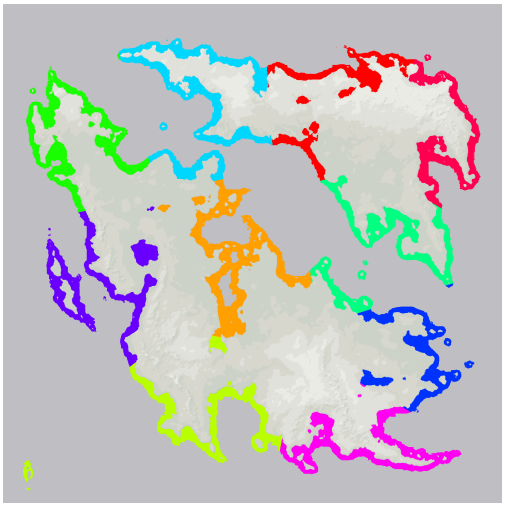


Un autre algorithme, dbscan aurait également pu être utilisé, mais K-means s'avère être plus rapide.
K-means peut avoir des difficultés pour trouver des clusters de forme complexe mais ce n'est pas trop un problème dans notre situation car les biomes sont des zones sous forme de sphères.

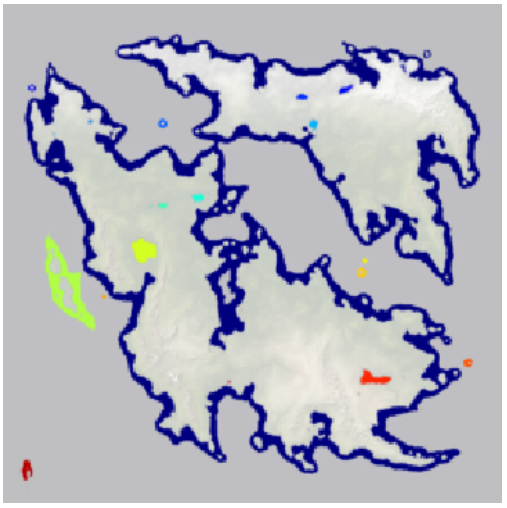
III - Le Clustering

Pour détecter les écosystèmes au sein d'un même biome, nous avons besoin d'un algorithme de clustering. La position des écosystèmes sur l'image étant importante pour cela, cette recherche va devoir prendre en compte la position des pixels. Cela rend l'utilisation de K-Means non adaptée dans cette situation. Pour répondre au problème, nous allons utiliser DBSCAN qui va nous permettre de regrouper les pixels de même couleur, donc même biome en cluster selon leurs positions sur l'image.
Comme montré ci-dessous, on se retrouve avec un découpage arbitraire selon le nombre de centroïdes choisie avec K-Means.





Et on ne retrouve pas de cluster cohérent avec la position des pixels entre eux, on cherche plus à avoir comme dans le sujet.



Malheureusement, notre implémentation ne nous permet pas d'effectuer dans un temps raisonnable le traitement sur l'image de test (800px X 800px).

IV - Lancement des programmes

```
public static void main(String[] args) {

        String entree = "ressource/img/Planete1.jpg";
        String sortie = "ressource/out/Planete1.jpg";
        String format = "png";

        try{
            Color[] colors = {new Color(34, 34, 31), new Color(214, 210, 206), ...
            Palette palette = new Palette(colors);

            String[] nameExtension = sortie.split("\\.(?=[^\\.]+$)");
            StepExporter exporter = new StepExporter(nameExtension[0],"." + nameExte
...

            Processor[] processes = {
                    exporter,
                    new BilateralFilter(7, 100, 100),
                    exporter,
                    new ColorReduction(new CostBasedKMeans(new ExpoColorCountC ...
                    exporter,
                    new Clusterer(new DBSCAN(1, 5), Clusterer.CLUSTER_BY_COLOR_AN ...
                    exporter,
            };

            ImageProcessor imageProcessor = new ImageProcessor(processes);
            BufferedImage image = ImageIO.read(new File(entree));
            imageProcessor.process(image);
            // No need to export the image, as the exporter is already
            // doing so for every step in the process
        }catch (IOException e){
            System.out.println(e);
        }
    }
```

Le lancement des programmes est d'une extrême simplicité. Les différents algorithmes implémentant l'interface Processor avec une méthode process qui lance l'algorithme. Il suffit donc de mettre les différentes étapes dans un tableau de Processor (ici appelé processes) et de le donner à une classe ImageProcessor qui va exécuter la liste de Processor. Il est également possible d'afficher les différentes étapes grâce à un StepExporter qui implémente également l'interface Processor. Ce système permet d'utiliser les différents algorithmes et d'exporter les résultats très facilement.