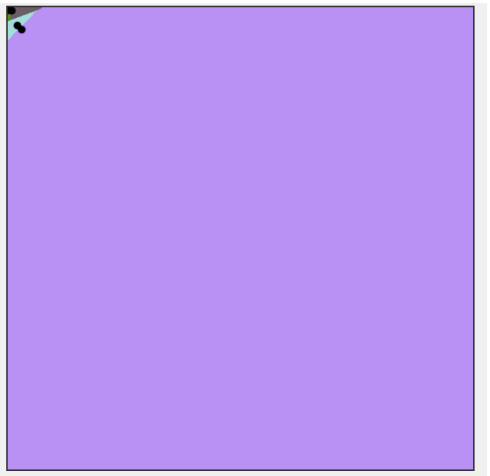
Pour réaliser le diagramme de voronoï nous avons utilisé 3 langages HTML,CSS et Javascript. Le HTML m'a servi à créer des éléments pour structurer l'interface comme le canevas, les boutons ou encore les textes qui me permettent de faire un retour à l'utilisateur. Pour le CSS il m'a servi pour la mise en page. C'est ensuite avec Javascript que l'on effectue toute la logique métier. Plusieurs logique de code on était enfermés dans des méthodes car on ne voulait pas de méthode trop longue, le but était de découper ces logiques car c'est un bon pattern en architecture, facile à reprendre en main pour un autre développeur si besoin de changements et pour faciliter la lecture du code. On a aussi décidé de coder en français car c'est plus compréhensible et plus facile pour nous. Plusieurs des fonctions dont je vais parler ont été retouchées au fur et à mesure de l'avancement de l'application.

Pour commencer on a d'abord récupéré les éléments comme en instanciant dans le JavaScript qui permet de le faire avec le ciblage du DOM et on les a créés en les enregistrant dans notre fichier Javascript pour pouvoir les utiliser plus tard dans notre code, c'est aussi là où on crée un tableau vide qui contiendra les points que l'on nous envoie dans le fichier .txt. On a d'abord écrit un code qui va lire le fichier que l'on nous envoie. Ensuite la plus grosse méthode qui fait réaliser le calcul c'est "analyserLeTexte(*texteBrut*)", c'est la méthode qui rassemble toutes les autres logiques pour nous donner notre diagramme de voronoï.

Ensuite on déclare un tableau qui contient nos points "`let listeDesPoints = [];`" il vide au départ et donc ce tableau a une position x et une position y et la couleurs associer à cette position. En mémoire ça fonctionne un peu comme ça "`{ posX: 120, posY: 80, couleur: "rgb(100,200,300)" }`".

Dans un premier temps on récupère les coordonnées que l'on nous envoie et on les nettoie pour les utiliser c'est-à- dire on supprime les espaces, on découpe le texte a chaque retour de ligne. On transforme chaque ligne en objet pour faciliter l'accès lors des calculs. Une fois que l'on a réussi à récupérer ces données, on les agrandit, on effectue une mise à l'échelle on appelle ça du "scaling" en multipliant par 20 car comme choix de base nous avons voulu un créé canevas de 600 pixels par 600 pixels donc quand on a essayé avec des valeurs sans agrandir le diagramme prenait à peine un quart de notre canevas. Suite à ça on retrouve les coordonnées des points à placer on les place et on colorie la zone.



exemple de rendu sans scaling

Quand l'ordinateur effectue le calcule il se passe 4 à 6 secondes ou rien ne s'affiche, pour pas que l'utilisateur pense que c'est un bug ou une erreur on a voulu donner un retour en temps réel mais le navigateur ne peut pas à la fois calculer et à la fois afficher le message donc on utilise setTimeout pour lancer le calcul un tout petit peu juste après et avoir une gestion Asynchrone.

Pour les logique de calcul utilisée, nous avons la méthode "calculer_distance_entre_deux_points(*x1*, *y1*, *x2*, *y2*)" qui calcul la distance entre 2 points, pour ça elle regarde la différence qu'il faut pour aller au point A plutôt qu' au point B. La distance entre un pixel et un point source donné par une formule qui revient à dire que la distance c'est la racine carré de $A^2 + B^2$ Cette formule c'est le théorème de Pythagore appliqué à la distance euclidienne.

Nous avons crée cette méthode car pour réaliser notre diagramme nous allons demander à chaque pixel quel est le point le plus proche de ta position et pour chaque pixel on va leur attribuer une couleur ce qui au final donnera des zones de couleur différentes avec les points que on nous a données, c'est comme sa que apparait notre diagramme de veronoï.

Nous avons une méthode qui va faire cela nommée "dessinerLesZones()" elle va parcourir tous les pixels grâce à une boucle imbriquée qui va parcourir chaque ligne de chaque colonne qui ensuite utilisera notre méthode pour calculer les distances et les appliquer à tous les pixels. De cette manière chaque pixel sait déjà de quel point il est le plus proche.

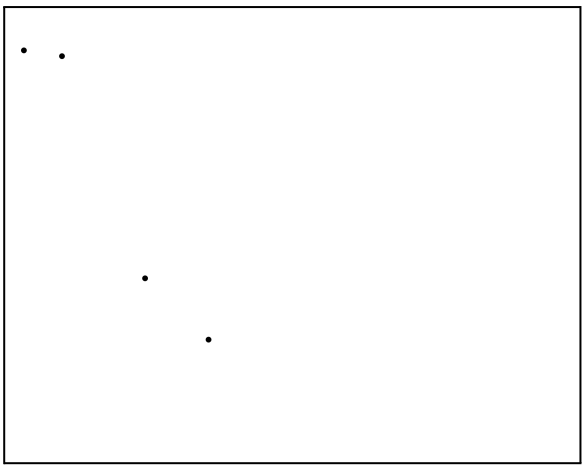
Pour la partie représentative on à séparé plusieurs logiques. Premièrement la méthode"dessinerLesPointsSources()" qui prend la liste des points et elle utilise la position du point comme centre et va donner un rayon de 5 pixels, ensuite c'est comme si elle trempait son pinceau dans la couleur que l'on a générée aléatoirement et elle colorie le cercle avec.

Pour générer ces couleurs aléatoirement nous avons créé “genererCouleurAleatoire()” qui va choisir un nombre entre 0 et 1 aléatoirement et va multiplier par 206 et si on a nombre qui se rapproche de 1 on le multiplie et ça donne à peu près 205 ce qui correspond à peu près à la largeur de la palette. Quand on a tester notre code on s’est rendu compte que parfois les couleurs etait trop sombre donc ce rapprocher du noir et que on voyait pas le point alors sur les 255 couleurs disponible on en à enlever 50 en faisant du coup plus + 50 dans notre code car plus on se rapproche de 0 plus c’etait noir, et avec Math.floor on arrondi sans la virgule et chaque valeur est mis dans 3 constante r g b et on les retourne. On a penser à une autre solution qui etait de redéfinir une liste de couleur mais on c’est que le fichier .txt pourrait contenir plus de points donc plus de zones que de couleur sur notre liste.

Autre soucis que on a eu c’est que d’abord on dessiner les points noir et ensiute on colorier les zones ce qui cacher les points noir donc a tout simplement créé une méthoide “redessinerPointsNoirs()” qui va retracer exactement les mêmes point noirs elle utilise la même logique que “dessinerLesPointsSources()” .

Dernier problème qu’il a fallu corriger c’est à propos des choix de couleur en testant on a eu une situation ou il y avait 2X la même couleur ce qui nous permettait pas de délimiter les zones visuellement donc on a créé une méthode appelé “couleurDejaUtilisee()” qui vérifie grâce a une boucle si cette couleur a déjà été utilisée.

Pour finir il a fallu que l'on puisse exporter notre diagramme pour ce faire nous avons d’abord essayé le format SVG, avec une méthode appelée “genererContenuSVG()” et on y connaissait rien donc on c’est principalement aidé de la documentation officielle du W3C et ce qui a le plus retenue mon attention c’est que on a besoin de ”`svg xmlns="http://www.w3.org/2000/svg`” pour utiliser SVG, c’est une adresse qui dit à mon navigateur que c’est un format spécial et pas du texte et comme sa l'ordinateur comprend les balises.
On a donc appris que le SVG est un fichier texte vectoriel contenant des balises et qu'un fichier SVG peut être ouvert dans n'importe quel logiciel de dessin ou navigateur sans perdre de qualité. Mais quand on a exporté on c’est rendu compte que ça exportait seulement les points et pas les zones colorées, on a penser à un bug et en regardant la console on a remarqué que le code SVG ne générait pas de balise de couleur pour le fond. En fait, le SVG permet d’avoir une image mais il fonctionne avec des instructions mathématiques et comme j’avais mis des instructions juste pour les points je n’ai eu que les points et mettre des instructions pour chaque pixel pour avoir la zone colorée aurait été trop long.



résultat obtenue en SVG

On a donc choisi d'exporter en PNG pour cela, on faisant des recherches j'ai trouvé la balise "`toDataURL("image/png")`" qui elle copie directement ce que l'on a sur le canevas, elle copie une chaîne qui contient chaque pixel. Comme le navigateur ne propose pas de bouton "Enregistrer sous" par défaut en JavaScript, nous avons dû ruser, on crée un élément `<a>` (un lien hypertexte) en mémoire, puis on place notre capture PNG dans l'attribut `href` de ce lien et du coup ça simule un clic informatique sur ce lien via le code (`lien.click()`) qui déclenche instantanément le téléchargement du fichier `mon_voronoi.png` chez l'utilisateur.

Pour réaliser les test j'ai voulu faire en Node.js vue que on fait du JavaScript mais j'ai vu que c'était pas propre que sa respectait pas un pattern car il faut recopier les fonctions dans le fichier test donc au final j'ai juste fait un fichier JavaScript et c'est sur la console du navigateur que on peut voir les résultat des tests. Il y a bien des méthodes que je ne peux pas tester car elle utilise le canevas et ne retourne rien et le résultat et visuel donc pas de code.

Pour les limites de ce code je pense que premièrement c'est la taille de notre canvas plus grand le temps de calcul deviendrait trop long car c'est la complexité algorithmique.
Et il y aussi le fait que si l'utilisateur envoie des données coordonnées trop grosses, elles sortiront de notre canevas, il faudrait trouver un moyen automatique pour ajuster automatiquement toutes les coordonnées que l'on peut recevoir.
Je pourrais aussi relever une erreur si le fichier reçu ne contenait pas que des chiffres.

Pour les couleurs il se peut que j'ai quand même des couleurs assez similaires à l'œil nu d'après ce qu' on a vu HSL est possiblement une solution car il garantit un écart de teinte.

Sources qui nous ont été utiles :

- *I Cracked The Code Behind Nature! (Voronoi Diagram Explanation)*
  - https://www.youtube.com/watch?v=m4W6ey2l6lk
- *Où faut-il construire ces palais ? (Le problème de la distance)*
  - https://www.youtube.com/watch?v=zL7eHC6WfyA
- *Calculer la distance entre deux points (Théorème de Pythagore)*
  - https://www.youtube.com/watch?v=x6kT22a7sOA
- *Les tutoriels sur l'API Canvas (MDN)*
  - https://developer.mozilla.org/fr/docs/Web/API/Canvas_API/Tutorial
- *MDN (Math.random)*
- *MDN Web Docs - Tutoriel SVG*
  - https://developer.mozilla.org/fr/docs/Web/SVG/Tutorial/Basic_Shapes
- *W3Schools - SVG Circle*
  - https://www.w3schools.com/graphics/svg_circle.asp
- https://developer.mozilla.org/fr/docs/Web/API/HTMLCanvasElement/toDataURL
- *Stack Overflow*
  - https://stackoverflow.com/questions/10673122/how-to-save-canvas-as-an-image-with-canvas-todataurl