

Filière MP - ENS de Cachan, Lyon, Rennes et Paris - Session 2016
Page de garde du rapport de TIPE

| | |
|-----------------------------|----------------------------------|
| NOM : BETHUNE | Prénoms : Louis |
| Classe : MP* | |
| Lycée : Henri-Wallon | Numéro de candidat : 9283 |
| Ville : Valenciennes | |

Concours auxquels vous êtes admissible, dans la banque MP inter-ENS (les indiquer par une croix) :

| | | | | |
|------------|-------------------------|----------|-------------------------|--|
| ENS Cachan | MP - Option MP | | MP - Option MPI | |
| | Informatique | X | | |
| ENS Lyon | MP - Option MP | | MP - Option MPI | |
| | Informatique - Option M | X | Informatique - Option P | |
| ENS Rennes | MP - Option MP | | MP - Option MPI | |
| | Informatique | X | | |
| ENS Paris | MP - Option MP | | MP - Option MPI | |
| | Informatique | X | | |

Matière dominante du TIPE (la sélectionner d'une croix inscrite dans la case correspondante) :

| | | | | | |
|--------------|----------|---------------|--|----------|--|
| Informatique | X | Mathématiques | | Physique | |
|--------------|----------|---------------|--|----------|--|

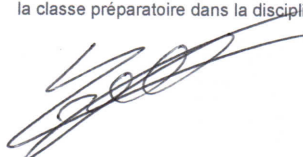

Titre du TIPE : **Diagrammes de Voronoï et algorithme de Fortune**

Nombre de pages (à indiquer dans les cases ci-dessous) :

| | | | | | |
|-------|----------|--------------|----------|---------------|----------|
| Texte | 6 | Illustration | 3 | Bibliographie | 1 |
|-------|----------|--------------|----------|---------------|----------|

Résumé ou descriptif succinct du TIPE (6 lignes, maximum) :

Dans ce TIPE nous nous sommes intéressés aux diagrammes de Voronoï et, plus particulièrement à l'algorithme de Fortune.
 J'ai ensuite utilisé ces diagrammes pour générer procéduralement des cartes.

| | | |
|----------------------------------|---|---|
| À Valenciennes | Signature du professeur responsable de la classe préparatoire dans la discipline | Cachet de l'établissement |
| Le 07/06/2016 |  | <div style="border: 1px solid black; padding: 5px;"> LYCÉE POLYVALENT H. WALLON B.P. 435 16, Place de la République 59322 VALENCIENNES CEDEX Tél. 03.27.19.30.40 - Fax 03.27.19.30.41 </div> |
| Signature du (de la) candidat(e) |  | |

Diagrammes de Voronoï et algorithme de Fortune

BÉTHUNE LOUIS

26 juin 2016

Notre objectif était d'étudier les diagrammes de Voronoï, et plus particulièrement l'algorithme de Fortune permettant de construire ces diagrammes de façon efficace. Cette partie est commune avec celle de mon binôme COIFFIER Guillaume.

Ensuite, je me suis intéressé à la manière dont on peut utiliser les diagrammes de Voronoï pour effectuer de la génération procédurale de cartes polygonales. Pour ce faire j'applique une relaxation de Lloyd sur une distribution aléatoire de points dans le plan, afin de donner un aspect régulier au pavage. Puis j'utilise le bruit de Perlin pour affecter une hauteur à chacun des points du diagramme.

1 Diagrammes de Voronoï de points

Les diagrammes de Voronoï possèdent des applications dans de très nombreux domaines comme le machine learning, l'épidémiologie, la cristallographie, et comme solution intermédiaire à plusieurs problèmes de géométrie algorithmique.

Soit S un ensemble fini de points du plan P , nommés **sites**. La cellule de Voronoï d'un site s est :

$$C(s) = \{p \in P \mid d(s, p) \leq d(s', p) \forall s' \in S\}$$

C'est donc l'ensemble des points du plan qui sont plus proche du site s que de tout autre site. Ainsi chaque cellule contient très exactement un site. Les cellules sont nécessairement convexes, car elles peuvent être définies comme une intersection finie de demi-plans.

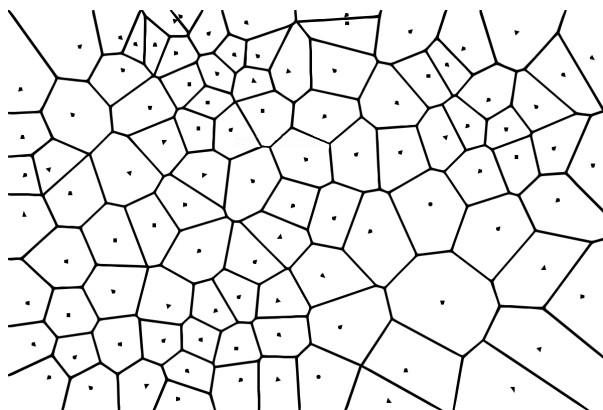


Diagramme de Voronoï d'une centaine de sites.

Deux cellules adjacentes sont séparées par une **arête** : elle est constituée des points équidistants de deux sites. En particulier toute arête est un morceau de la médiatrice du segment joignant les deux sites.

Les arêtes (au minimum trois) s'intersectent en un **sommet** du diagramme, centre du cercle circonscrit à au moins trois sites, et dont l'intérieur du disque ne contient aucun autre site. Un tel disque est nommé **disque de Delaunay**.

Le diagramme est défini comme la réunion de tout les sommets et toutes les arêtes : c'est un graphe, qui est planaire par construction. Les cellules pavent le plan.

Il est possible de construire le diagramme en $\mathcal{O}(n^2)$ à l'aide d'un algorithme incrémental, qui actualise le diagramme courant à chaque ajout d'un nouveau site (algorithme de Green et Sibson, 1978). En

1975 Shamos et Hoey ont proposé un algorithme en $\mathcal{O}(n \log n)$ qui s'appuie sur un diviser pour régner, qui fusionne deux sous-diagrammes à chaque étape de la récursion. Finalement, en 1987 Steve Fortune propose un autre algorithme de complexité $\mathcal{O}(n \log n)$, réputé plus simple à implémenter que le précédent.

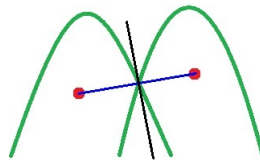
2 Algorithme de Fortune

L'algorithme de Fortune appartient à la famille des algorithmes dit **à ligne de balayage** : il s'appuie sur une droite virtuelle Δ qui balaye le plan dans le sens des ordonnées décroissantes. Le diagramme est construit au fur et à mesure.

On a déjà vu que les points sur les arêtes sont équidistants de deux sites. Comme l'ordonnée y_Δ de la ligne de balayage prend successivement la valeur de l'ordonnée des différents sites, il est naturel de considérer les points équidistants de la droite et des sites déjà rencontrés. Pour chaque site, l'ensemble des points vérifiant cette équation est une parabole ayant le site pour foyer. Chaque parabole définit une région de l'espace non bornée qui s'étend à l'infini *derrière* la ligne de balayage. Ces régions peuvent se recouvrir. Les arcs de parabole qui ne sont inclus dans la région d'aucune autre parabole forment le **front parabolique**.

2.0.1 Point anguleux

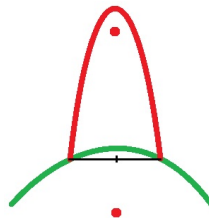
Deux arcs de paraboles du front s'intersectent en un **point anguleux**. La distance du point anguleux à chacun des deux sites est égale à la distance du point anguleux à la droite Δ . Donc le point anguleux est à même distance des deux sites : il appartient donc à une arête du diagramme. Ainsi, au fur et à mesure que la ligne balaye, le front parabolique avance, et les points anguleux tracent les arêtes du diagramme.



L'arête (en noir) est la médiatrice du segment (en bleu) joignant les deux sites, elle est engendrée par les paraboles (en vert). Les dimensions réelles ne sont pas respectées.

2.0.2 Évènement site : apparition d'arcs de parabole

Chaque fois que la ligne de balayage rencontre un site (ce qu'on appellera désormais un **évènement site**), on crée une nouvelle parabole dégénérée (réduite à une demi-droite) qu'on insère dans le front parabolique. L'arc de parabole ainsi intersecté est coupé en deux morceaux, donnant naissance à deux **demi-arêtes**. On les nomme ainsi car chacune d'entre elle s'étend dans un sens différent, et leur "raccordement" donne l'arête finale. En général on ne connaît pas immédiatement leurs autres extrémités. En particulier, on ne connaît pas forcément les deux extrémités de l'arête du diagramme : on sait juste qu'elle existe et passe par ce point.

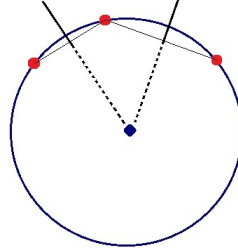


On observe ici les deux demi-arêtes (en noir) en cours de construction. La nouvelle parabole (en rouge) intersecte l'ancienne (en vert). Les dimensions réelles ne sont pas respectées.

2.0.3 Évènement cercle : disparition d'arcs de parabole

Certains arcs de parabole finissent par être réduits à un point, lorsque les deux arcs adjacents finissent par le recouvrir (cela arrive quand le site s_m de l'arc disparu est plus éloigné de la ligne balayage que

les sites s_g et s_d des deux autres arcs). À ce moment les deux points anguleux de part et d'autre sont confondus. Cela signifie donc que les deux arêtes s'intersectent en ce point, qui se trouve être un sommet du diagramme ! Comme il appartient aux deux arêtes, il est à même distance des trois sites, et donc il est bien le centre du cercle circonscrit aux trois sites, d'où la dénomination d'**événement cercle**. On peut alors terminer les deux arêtes a_g et a_d , et en créer une nouvelle, ayant pour direction celle de la médiatrice du segment $[s_g; s_d]$.



On anticipe l'intersection des arêtes (en pointillés) sur un sommet du diagramme, centre du cercle circonscrit (en bleu).

3 Implémentation

L'algorithme décrit ci-dessus semble fonctionner comme si Δ balayait le plan de façon continue. Toutefois ce n'est pas nécessaire : il suffit de "sauter" aux endroits stratégiques pour obtenir les coordonnées des sommets du diagramme. On en déduit alors facilement les arêtes.

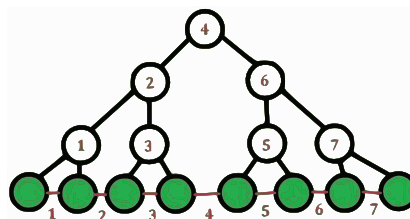
3.0.4 File à priorité et événements

Pour cela on insère dans une file à priorité les événements sites, triés selon l'ordonnée décroissante. Lors de la création des arêtes, on vérifie qu'elles n'intersectent pas une arête adjacente, auquel cas on crée un événement cercle qu'on insère aussi dans la file. On remarque que la découverte d'un événement site peut parfois invalider un événement cercle déjà inséré. Dans ce cas là il faut penser à marquer les événements cercles associés comme "périmés". Ils seront ignorés lorsqu'ils seront sortis de la file (principe du tas paresseux : on ne cherche pas à supprimer l'élément, on se contente de l'ignorer).

3.0.5 Gestion du front parabolique : utilisation d'un arbre binaire

La seconde difficulté consiste à insérer un nouvel arc parabolique dans le front. On pourrait utiliser une liste triée par abscisse (l'abscisse du site correspond à l'arc de parabole), mais les recherches et insertions se feraient en $\mathcal{O}(n)$ ce qui est prohibitif. Au lieu de ça on utilise un **arbre binaire** qui permet les mêmes opérations en $\mathcal{O}(\log n)$ en moyenne, car c'est la profondeur moyenne d'un arbre binaire construit aléatoirement. Il est possible d'équilibrer cet arbre pour garantir une complexité logarithmique en toutes circonstances mais nous n'avons pas implémenté cette fonctionnalité.

Contrairement à un arbre binaire de recherche, dans lequel les éléments sont stockés dans tous les nœuds de l'arbre, on choisit de stocker les arcs de paraboles exclusivement aux feuilles, qui se trouvent être ordonnées par abscisses croissantes. On sait que les arêtes se situent toujours entre deux feuilles. Or on peut mettre en bijection les nœuds (non feuillus) de l'arbre avec une paire de deux feuilles consécutives ! Chaque nœud non feuillu enjambe exactement deux feuilles consécutives, on peut donc y stocker l'arête séparant les deux arcs de parabole.



Les positions des arêtes sont numérotés en rouge. On observe qu'il y a bien bijection entre le nœud et l'intervalle.

Depuis un arc on peut donc accéder à l'arête adjacente, et vice-versa, en $\mathcal{O}(\log n)$ en moyenne.

3.0.6 Pseudo-code

- Ensemble de sites \mathcal{S}
- File à priorité \mathcal{F}
- Tableau d'arêtes \mathcal{E}
- Tableau de sommets \mathcal{V}
- Arbre du front parabolique \mathcal{A}

PROCÉDURE FORTUNE(\mathcal{S}) :

Insérer tous les événements sites de \mathcal{S} dans \mathcal{F}

Tant que \mathcal{F} est non vide

Extraire de \mathcal{F} un événement e

Si e est un événement site

Créer P_m la parabole de site $e.site$

Trouver P la parabole dans \mathcal{A} intersectée par P_m , de site s , et soit v le point d'intersection

Marquer l'événement cercle associé à P comme périmé

Créer deux demi-arêtes A_g et A_d d'extrémité v , entre $e.site$ et s , et en ajouter une dans \mathcal{E}

Soit P_g et P_d deux paraboles de site v

Remplacer P par la séquence $(P_g, A_g, P_m, A_d, P_d)$

TESTERÉVÈNEMENTCERCLE(P_g)

TESTERÉVÈNEMENTCERCLE(P_d)

Si e est un événement cercle non périmé

Soit P la parabole de \mathcal{A} associée à e

Soit P_g et P_d les paraboles de \mathcal{A} voisines de P

Marquer les événements cercles associés à P_g et P_d comme périmés

Soit A_g et A_d les arêtes de \mathcal{E} voisines de P

Soit v le point d'intersection de A_g et A_d , l'ajouter à \mathcal{V}

Terminer les arêtes A_g et A_d avec v

Créer une nouvelle arête A_m partant de v et l'ajouter à \mathcal{E}

Remplacer la séquence (A_g, P, A_d) par A_m

TESTERÉVÈNEMENTCERCLE(P_g)

TESTERÉVÈNEMENTCERCLE(P_d)

Pour chaque arête e dans \mathcal{E}

Si e est une demi-arête

Raccorder e et $e.voisin$

FIN DE LA PROCÉDURE

PROCÉDURE TESTERÉVÈNEMENTCERCLE(P) :

Soit P_g et P_d les paraboles de \mathcal{A} voisines de P

Soit A_g et A_d les arêtes de \mathcal{E} voisines de P

Si P_g , P_d , A_g ou A_d n'existe pas

SORTIR

Soit v le point d'intersection des deux arêtes

Soit d la distance de v au site de P

Si $v.y - d$ est avant la ligne de balayage

SORTIR

Ajouter dans \mathcal{F} un événement cercle au point de coordonnées $(v.x, v.y - d)$

Associer cet événement à P

FIN DE LA PROCÉDURE

3.0.7 Analyse et résultats

Notre implémentation demeure un $\mathcal{O}(n^2)$ en pire des cas, néanmoins elle se comporte comme un $\mathcal{O}(n \log n)$ en moyenne, sur une distribution aléatoire de points (ce que l'on peut vérifier sur les **figures 1 et 2**).

La manière dont les événements cercles sont traités ne permet de gérer que le cas de trois points cocycliques. Lorsque plus de deux arêtes sont concourantes (par exemple lorsque les sites sont aux coins d'un carré) l'algorithme renvoie un résultat incohérent. Heureusement, les chances qu'un tel événement survienne sont faibles sur une distribution aléatoire de points.

Sur une distribution réelle de points, on pourra par exemple limiter cet effet en perturbant les coordonnées de chaque site d'un ϵ . Cela suffira à rendre les points non cocycliques (avec une forte probabilité) tout en conservant l'allure générale du diagramme.

4 Génération procédurale de cartes

La génération procédurale est la création de contenu numérique à grande échelle, de façon automatisée et aléatoire. Elle est très utilisée dans le domaine du jeu vidéo, et plus particulièrement celui de la génération de *maps* (comme *Age of empire*, *Civilisation*, *Minecraft*), ce qui permet de proposer à l'utilisateur du contenu très diversifié en quantité presque illimitée.

Néanmoins l'utilisation d'un aléatoire non contrôlé conduit suivant à des résultats aberrants. On préférera donc un processus **semi-aléatoire**, produisant des résultats localement chaotiques mais globalement cohérents.

4.0.8 Cartes polygonales

Il peut-être intéressant de découper la carte en différentes **régions**. C'est la première étape d'un traitement plus fin qui consistera ensuite à générer le contenu de ces régions (altitude, végétation, village, rivières, forêts, montagnes, ressources...). Le moyen le plus couramment employé est l'utilisation d'une grille bidimensionnelle régulière, où toutes les régions sont de forme carrée et de mêmes dimensions. On peut également trouver des pavages hexagonaux du plan. Cependant la régularité trop importante de ces pavages peut donner un aspect irréaliste à la carte.

En revanche un pavage formé de différents polygones, espacés de façon régulière mais non constante donne un aspect plus naturel et plus chaotique, conformément à l'intuition que l'on se fait d'un terrain.

Ainsi les diagrammes de Voronoï sont tout indiqués : on tire au hasard des points dans le plan, on en calcule le diagramme, et les cellules de notre diagrammes pavent le plan. Plus les points seront nombreux et plus les cellules seront petites, et donc plus le maillage sera fin.

4.0.9 Relaxation de Lloyd

Une distribution purement aléatoire contient parfois des zones avec beaucoup de sites (très proches les uns des autres) et d'autres zones où ils sont plus rares (et les cellules plus grandes). Pour éviter ça on peut appliquer une **relaxation de Lloyd**. Cet algorithme calcule le diagramme de Voronoï d'un ensemble de sites, puis calcule le barycentre de chacune des cellules. Il y a donc autant de barycentres que de sites, et on peut calculer le diagramme de Voronoï de ces barycentres. Cette procédure peut être appliquée autant de fois que souhaité (voir **figures 3, 4 et 5**). Généralement une seule fois suffit.

4.0.10 Height map et bruit de Perlin

Il est nécessaire d'affecter une altitude à chaque sommet du diagramme. Ensuite, on définit la hauteur de chaque site comme la moyenne des hauteurs des sommets de sa cellule. Le terrain est alors composé de plein de petits triangles à l'allure caractéristique (voir **figure 6**).

Pour déterminer la hauteur de chaque sommet on utilise le **bruit de Perlin**, mis au point par Ken Perlin en 1985. C'est un algorithme produisant un bruit de gradient : il est donc parfaitement adapté à la génération de gradients d'altitude.

PROCÉDURE PERLINNOISE(**Tableau de sommets** \mathcal{V} , **Fréquence** f , **Amplitude** a , **Fonction interpoler**) :

Soit \mathcal{G} une grille bidimensionnelle, de largeur f , à cases carrées, et recouvrant une partie du plan contenant \mathcal{V}

Pour chaque sommet c de \mathcal{G}

 Calculer la position $c.p$

 Affecter un vecteur unitaire de direction aléatoire à $\vec{c.g}$

Pour chaque sommet v de \mathcal{V}

 Déterminer la case \mathcal{C} de \mathcal{G} à laquelle appartient v

Tableau de flottants \mathcal{W} de taille 2×2

 Pour chaque coin c de \mathcal{C}

 Calculer $\vec{c.g} \cdot \frac{\vec{c.p} - \vec{v.p}}{\|\vec{c.p} - \vec{v.p}\|}$ et l'ajouter à \mathcal{W}

$y_1 \leftarrow \text{interpoler}(\mathcal{W}[\text{Haut}][\text{Gauche}], \mathcal{W}[\text{Haut}][\text{Droite}])$

$y_2 \leftarrow \text{interpoler}(\mathcal{W}[\text{Bas}][\text{Gauche}], \mathcal{W}[\text{Bas}][\text{Droite}])$

$z_0 \leftarrow \text{interpoler}(y_1, y_2)$

$v.z \leftarrow v.z + z_0 \times a$

FIN DE LA PROCÉDURE

Cet algorithme peut être appliqué autant de fois que nécessaire, avec des fréquences chaque fois plus grandes et des amplitudes chaque fois plus basses. Ainsi les basses fréquences dessinent "les grandes lignes" du relief (montagnes, vallées...) et les hautes fréquences permettent d'en moduler la granularité (voir **figure 8**). On utilisera le terme "d'octave" pour chacune des applications du bruit liée à une fréquence particulière.

La fonction d'interpolation renvoie $a + f(t)(b - a)$ avec a et b les valeurs que l'on interpole, et $t \in [0, 1]$ un paramètre réel qui ne dépend que de la projection du sommet sur le segment joignant les coordonnées de a et b .

La fonction f peut-être choisie librement, du moment qu'elle est croissante sur $[0, 1]$ à valeurs dans $[0, 1]$, avec $f(0) = 0$ et $f(1) = 1$. On peut par exemple prendre $Id_{\mathbb{R}}$ ce qui donnera une interpolation linéaire, ou quelque chose de plus compliqué (un polynôme de degré supérieur, une fonction sinusoïdale...).

4.0.11 Îles

Le bruit de Perlin donne des résultats assez satisfaisants mais ne permet pas de contrôler finement les formes du territoire (choisir la position des montagnes ou des lacs). On aimerait par exemple pouvoir générer des îles (zone centrale élevée, entourée d'eau).

Pour ce faire on introduit un terme correctif : on calcule la distance de chaque sommet au centre de l'image (ce centre pouvant être défini de façon arbitraire par translation de l'origine). Cette distance est ensuite normalisée (divisée par sa valeur maximum) pour être ramenée entre 0 et 1, appelons la d_0 . On multiplie alors les hauteurs $v.z$ par $e^{-ad_0} + b$. Les paramètres a et b peuvent être réglés par essais-erreurs selon le résultat souhaité.

Cette formule tend à élever le relief proche du centre, et à abaisser celui qui s'en éloigne. Il en résulte l'apparition d'une "île" près du centre.

4.0.12 Couleurs

Une correspondance a été établie manuellement entre la hauteur et la couleur d'une région. Ainsi en dessous de certains seuils on trouvera la mer, la plage, la forêt, la montagne, la neige... certains intervalles de hauteur sont associés à un dégradé de couleur (eau et forêt) pour que le résultat soit plus agréable au regard. Ici comme dans la partie précédente, les critères de choix sont moins d'ordre technique que esthétique.

5 Bibliographie

- D. Beauquier, J. Berstel, Ph. Chrétienne, *Éléments d'algorithmique*, 2005
- Vincent Pilaud, *Sur le diagramme de Voronoï et la triangulation de Delaunay d'un ensemble de points dans un revêtement du plan euclidien*, 2006
- Franck Hétroy, *Un petit peu de géométrie algorithmique*, cours de Grenoble INP Ensimag
- Ivan Kuckir, *Fortune's algorithm and implementation*,
<http://blog.ivank.net/fortunes-algorithm-and-implementation.html>
- Wikipédia, *Fortune's algorithm*,
https://en.wikipedia.org/wiki/Fortune's_algorithm
- Matt Brubeck, *Fortune's Algorithm in C++*,
<https://www.cs.hmc.edu/~mbrubeck/voronoi.html>
- Red Blob Games, *Polygonal Map Generation for Games*,
<http://www-cs-students.stanford.edu/~amitp/game-programming/polygon-map-generation/>
- Développez, *Génération de terrain par l'algorithme de Perlin*,
<http://khayyam.developpez.com/articles/algo/perlin/>
- Adrian's soap box, *Understanding Perlin Noise*,
<http://flafla2.github.io/2014/08/09/perlinnoise.html>

6 Figures

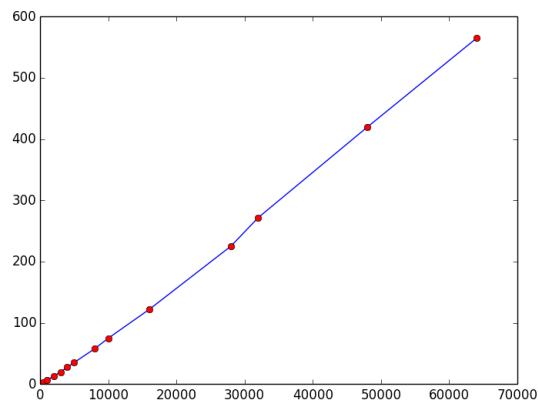


Figure 1 : nombre de sites en abscisse, et temps de calcul en ordonnée (en millisecondes) sur un Intel(R) Core(TM) i3-4010U CPU cadencé à 1.7GHz. Les temps pour les petits ensembles de points (inférieurs à 10 000) ont été moyennés sur plusieurs dizaines d'exécutions.

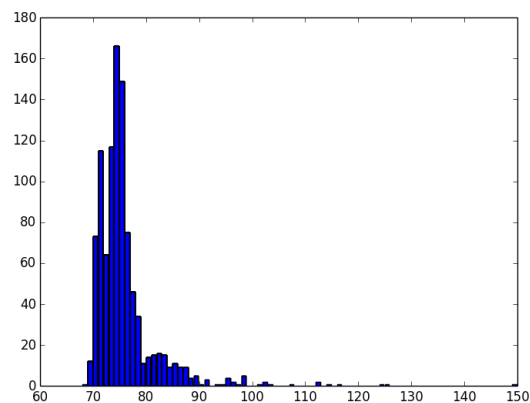


Figure 2 : répartition des temps de calcul pour 1000 exécutions d'un ensemble de 10 000 points générés aléatoirement. En abscisse le temps de calcul (en millisecondes) et en ordonnée les effectifs. La courbe a l'allure d'une distribution à "queue épaisse". La plupart des temps se concentrent autour de la moyenne mais certains sont jusqu'à deux fois plus long.

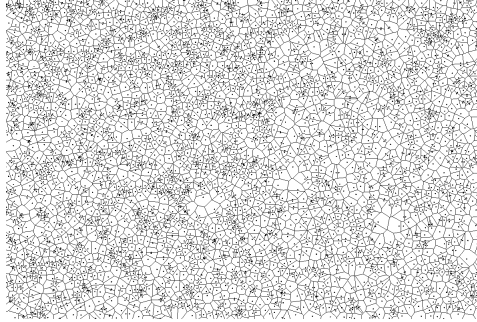


Figure 3 : diagramme de Voronoï d'un ensemble de 4000 points tirés au hasard.

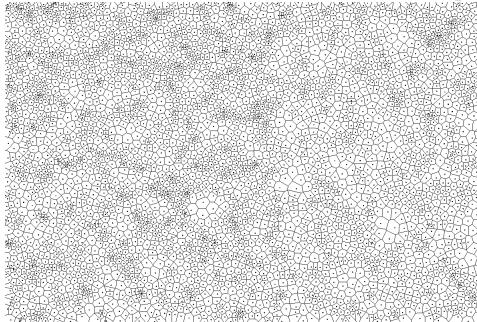


Figure 4 : même ensemble de points que **Figure 3**, après une itération de la relaxation de Lloyd.

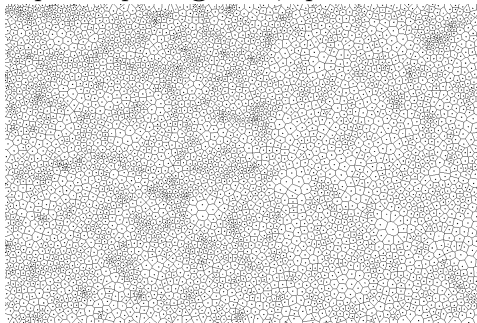


Figure 5 : même ensemble de points que **Figure 3**, après deux itérations de la relaxation de Lloyd.

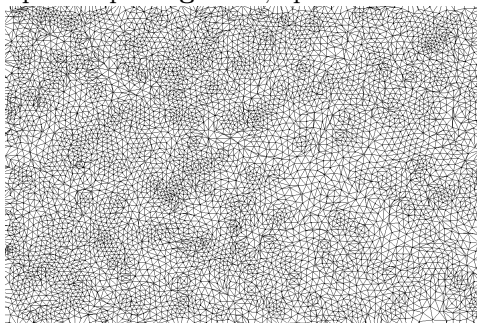


Figure 6 : diagramme de Voronoï de 2000 points, avec en plus les rayons joignant chaque site aux sommets de sa cellule. Ne pas confondre avec la triangulation de Delaunay (figure 7).

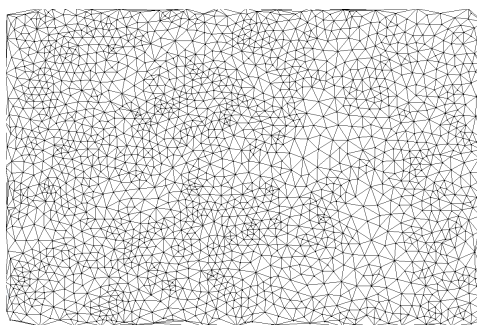


Figure 7 : triangulation de Delaunay de 2000 points, graphe dual du diagramme de Voronoï (relaxé deux fois).

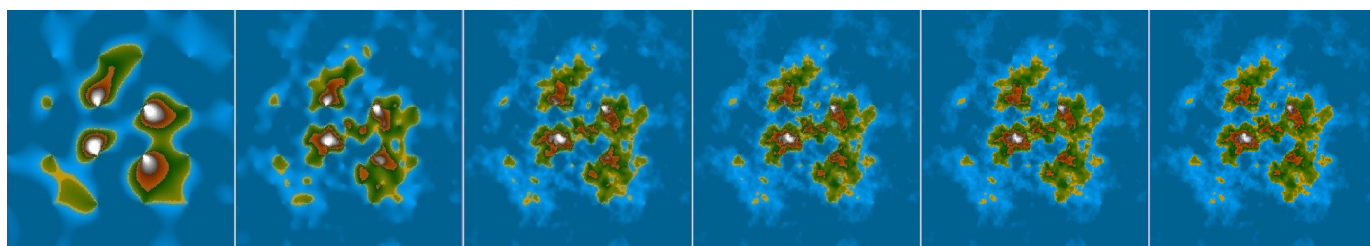


Figure 8 : six octaves du bruit de Perlin sur la même carte (64 000 polygones).

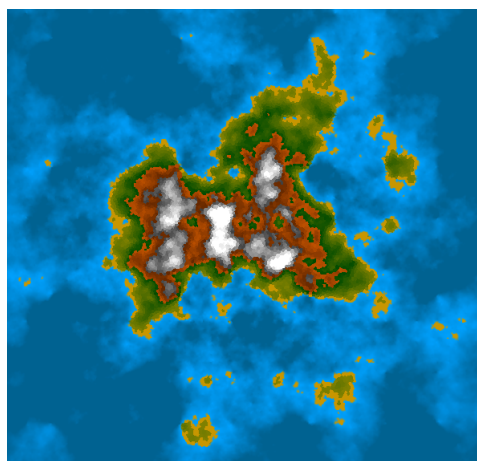


Figure 9 : autre île (64 000 polygones).

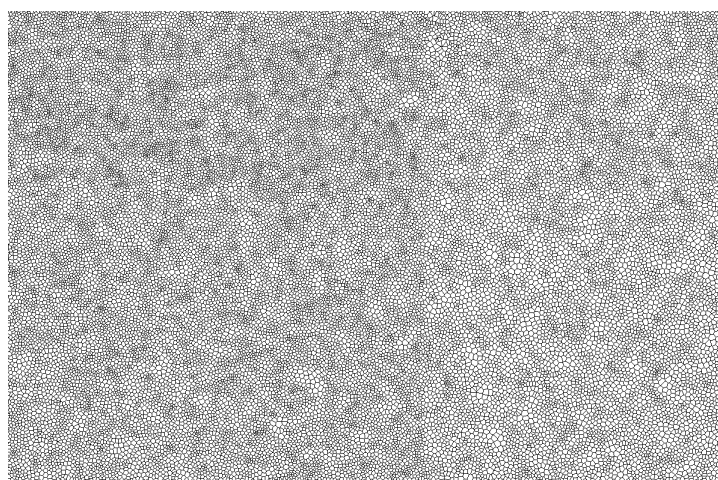


Figure 9 : diagramme de Voronoï de 32 000 sites (non affichés) après deux relaxations.

7 Code source

On trouvera l'algorithme de Fortune dans **FortuneProcedure.h** et **FortuneProcedure.cpp**, le bruit de Perlin et la génération d'îles dans **PerlinNoise.h** et **PerlinNoise.cpp**. **VoronoiDiagram.h** et **VoronoiDiagram.cpp** définissent la structure de graphe et la relaxation de Lloyd, ainsi que d'autres utilitaires. **GeometricUtility.h** et **GeometricUtility.cpp** contiennent quelques fonctions pratiques. **VoronoiDiagram.cuh** et **VoronoiDiagram.cu** implémente la génération d'un diagramme de Voronoï massivement parallélisé sur carte graphique avec la technologie CUDA.

7.0.13 main.cpp

```
1 #include <algorithm>
2 #include <iostream>
3 #include <functional>
4 #include <fstream>
5 #include <vector>
6
7 #include <SFML/Graphics.hpp>
8
9 #include "FortuneProcedure.h"
10 #include "PerlinNoise.h"
11 #include "PointGenerator.h"
12 #include "VoronoiDiagram.h"
13
14 using namespace std;
15 using namespace sf;
16
17 #define ISLAND
18 #define GPU_NOISE
19
20 void computeTests(unsigned int width, unsigned int height, unsigned int
    nbPoints, unsigned int nbTests) {
21     ofstream out("tests.txt");
22     for (unsigned int test = 0; test < nbTests; ++test) {
23         vector<Vector2d> points = VoronoiDiagram::randomPoints(nbPoints, width,
            height);
24         Fortune fortune(width, height, points);
25         Clock clock;
26         fortune.computeDiagram();
27         out << clock.getElapsedTime().asMilliseconds() << "\n";
28     }
29 }
30
31 int main() {
32     const unsigned int width = 800;
33     const unsigned int height = 800;
34
35     const unsigned int nbPoints = 64*1000;
36
37     vector<Vector2d> points = VoronoiDiagram::randomPoints(nbPoints, width,
        height);
38
39     cout << "Calcul du diagramme...\n";
40     Clock clock;
41     points = VoronoiDiagram::LloydRelaxation(width, height, points, 2);
```

```

42     VoronoiDiagram::VoronoiGraph graph = VoronoiDiagram::
        optimisedProceduralGeneration(width, height, points);
43     vector<CircleShape> sites = graph.getSites(1.5);
44     cout << clock.getElapsedTime().asMilliseconds() << "_ms\n";
45
46 #ifdef ISLAND
47     double fundamental = 5.;
48     unsigned int nbOctaves = 6;
49     double persistence = 0.5;
50     const auto colorGradient = islandGradient();
51     const auto vertices = &VoronoiDiagram::VoronoiGraph::getMixedMap;
52
53     cout << "Calcul_de_la_carte..._";
54     clock.restart();
55 #ifndef GPU_NOISE
56     computeMap(graph, quinticInterpolate, fundamental, nbOctaves, persistence
        );
57 #else
58     CudaPerlinNoise::computeParallelMap(graph, fundamental, nbOctaves,
        persistence);
59 #endif
60     cout << clock.getElapsedTime().asMilliseconds() << "_ms\n";
61
62     VertexArray pixels = (graph.*vertices)(colorGradient);
63 #else
64     VertexArray pixels = graph.getDelaunayTriangulation();
65 #endif
66
67     RenderWindow window(VideoMode(width, height), "Map_generator");
68
69     while (window.isOpen()) {
70         Event event;
71         while (window.pollEvent(event)) {
72             if (event.type == Event::Closed)
73                 window.close();
74         }
75
76         window.clear(sf::Color::White);
77         window.draw(pixels);
78 #ifdef ISLAND
79         for (const auto& site : sites)
80             window.draw(site);
81 #endif
82         window.display();
83     }
84
85     return 0;
86 }

```

7.0.14 VoronoiDiagram.h

```
1  #ifndef VORONOI_DIAGRAM_HPP
2  #define VORONOI_DIAGRAM_HPP
3
4  #include <functional>
5  #include <memory>
6  #include <set>
7
8  #include "SFML\Graphics.hpp"
9
10 typedef sf::Vector2<double> Vector2d;
11
12 namespace VoronoiDiagram {
13
14     std::vector<Vector2d> randomPoints(unsigned int, unsigned int, unsigned
        int);
15     std::vector<sf::Color> randomColors(unsigned int);
16
17     sf::VertexArray naiveProceduralGeneration
18         (unsigned int, unsigned int, const std::vector<Vector2d>&, const std::
            vector<sf::Color>&);
19
20     sf::VertexArray naiveParallelGeneration
21         (unsigned int, unsigned int, const std::vector<Vector2d>&, const std::
            vector<sf::Color>&);
22
23     class VoronoiGraph;
24     VoronoiDiagram::VoronoiGraph optimisedProceduralGeneration (unsigned int,
        unsigned int, const std::vector<Vector2d>&);
25
26     std::vector<Vector2d> LloydRelaxation(unsigned int, unsigned int, const
        std::vector<Vector2d>&, unsigned int);
27
28     class VoronoiGraph {
29     public:
30
31         struct DualEdge {
32             DualEdge(unsigned int, unsigned int, unsigned int, unsigned);
33             unsigned int nodeA;
34             unsigned int nodeB;
35             unsigned int siteA;
36             unsigned int siteB;
37         };
38
39         struct Edge;
40         struct Node;
41         struct Site;
42
43         struct Site {
44             Site(const Vector2d&);
45
46             Vector2d pos;
47             std::vector<Site*> borders;
48             std::vector<Edge*> edges;
49             std::set<Node*> nodes;
```

```

50
51     double elevation;
52 };
53
54 struct Node {
55     Node(const Vector2d&);
56
57     Vector2d pos;
58     std::vector<Node*> nodes;
59     std::vector<Edge*> edges;
60
61     double elevation;
62 };
63
64 struct Edge {
65     Edge(Node*,Node*,Site*,Site*);
66     Node* nodeA;
67     Node* nodeB;
68     Site* siteA;
69     Site* siteB;
70 };
71
72 VoronoiGraph(
73     double,
74     double,
75     const std::vector<Vector2d>&,
76     const std::vector<Vector2d>&,
77     const std::vector<DualEdge>&);
78
79 sf::VertexArray getEdges() const;
80 sf::VertexArray getBeams() const;
81 sf::VertexArray getDelaunayTriangulation() const;
82
83 std::vector<sf::CircleShape> getSites(float) const;
84
85 sf::VertexArray getCellMap(std::function<sf::Color (double)>) const;
86 sf::VertexArray getTriangulationMap(std::function<sf::Color (double)>)
87     const;
88 sf::VertexArray getMixedMap(std::function<sf::Color (double)>) const;
89
90 double width;
91 double height;
92 std::vector<std::unique_ptr<Site>> sites;
93 std::vector<std::unique_ptr<Node>> nodes;
94 std::vector<std::unique_ptr<Edge>> edges;
95 };
96 }
97 #endif

```

7.0.15 VoronoiDiagram.cpp

```
1 #include "VoronoiDiagram.h"
2
3 #include <chrono>
4 #include <iostream>
5 #include <memory>
6 #include <random>
7 #include <set>
8
9 #include "FortuneProcedure.h"
10 #include "GeometricUtility.h"
11 #include "VoronoiDiagram.h"
12 #include "VoronoiDiagram.cuh"
13
14 using namespace std;
15 using namespace sf;
16
17 namespace {
18     const unsigned seed = static_cast<unsigned int>(chrono::system_clock::now
19         ().time_since_epoch().count());
20     default_random_engine generator(643227911);
21 }
22
23 vector<Vector2d> VoronoiDiagram::randomPoints(unsigned int nbPoints,
24     unsigned int width, unsigned int height) {
25     const unsigned factor = 1000*1000;
26     vector<Vector2d> points(nbPoints);
27     generate(begin(points), end(points),
28         [=]() {return Vector2d(double(generator()%(width*factor))/factor, double
29             (generator()%(height*factor))/factor);});
30     return points;
31 }
32
33 vector<Color> VoronoiDiagram::randomColors(unsigned int nbPoints) {
34     vector<sf::Color> colors(nbPoints);
35     generate(begin(colors), end(colors),
36         [=]() {return Color(generator()%255, generator()%255, generator()%255)
37             ;});
38     return colors;
39 }
40
41 VertexArray VoronoiDiagram::naiveProceduralGeneration
42     (unsigned int width, unsigned int height,
43     const vector<Vector2d>& points,
44     const vector<Color>& colors)
45 {
46     VertexArray pixels(PrimitiveType::Points, width*height);
47     for (unsigned int y = 0; y < height; ++y) {
48         for (unsigned int x = 0; x < width; ++x) {
49             Vertex& currentPixel = pixels[y*width + x];
50             currentPixel.position = Vector2f(float(x), float(y));
51             double bestDistance = distance2(points[0], Vector2d(currentPixel.
52                 position));
53             unsigned int bestNeighbour = 0;
```



```

49     for (unsigned int neighbour = 1; neighbour < points.size(); ++
50         neighbour) {
51         double distance = distance2(points[neighbour], Vector2d(
52             currentPixel.position));
53         if (distance < bestDistance)
54         {
55             bestDistance = distance;
56             bestNeighbour = neighbour;
57         }
58     }
59     currentPixel.color = colors[bestNeighbour];
60 }
61 return pixels;
62 }
63
64 VertexArray VoronoiDiagram::naiveParallelGeneration
65     (unsigned int width, unsigned int height,
66     const vector<Vector2d>& points,
67     const vector<Color>& colors)
68 {
69     VertexArray pixels(sf::PrimitiveType::Points, width*height);
70     for (unsigned int y = 0; y < height; ++y)
71         for (unsigned int x = 0; x < width; ++x)
72             pixels[y*width + x].position = Vector2f(float(x), float(y));
73
74     unique_ptr<unsigned int[]> h_index(new unsigned int[width*height]);
75     unique_ptr<unsigned int[]> h_x(new unsigned int [points.size()]);
76     unique_ptr<unsigned int[]> h_y(new unsigned int [points.size()]);
77     for (unsigned int point = 0; point < points.size(); ++point) {
78         h_x[point] = static_cast<unsigned int>(points[point].x);
79         h_y[point] = static_cast<unsigned int>(points[point].y);
80     }
81
82     CUDA_VoronoiDiagram::naiveParallelGeneration(width, height, h_index.get(),
83         , points.size(), h_x.get(), h_y.get());
84
85     for (unsigned int pixel = 0; pixel < pixels.getVertexCount(); ++pixel)
86         pixels[pixel].color = colors[h_index[pixel]];
87     return pixels;
88 }
89
90 VoronoiDiagram::VoronoiGraph VoronoiDiagram::optimisedProceduralGeneration
91     (
92     unsigned int width,
93     unsigned int height,
94     const std::vector<Vector2d>& points) {
95     Fortune fortune(width, height, points);
96     fortune.computeDiagram();
97     return fortune.getVoronoiGraph();
98 }
99
100 vector<Vector2d> VoronoiDiagram::LloydRelaxation
    (unsigned int width,

```

```

101     unsigned int height,
102     const vector<Vector2d>& sites,
103     unsigned int nbIterations)
104 {
105     vector<Vector2d> diagram = sites;
106     for (unsigned int iteration = 0; iteration < nbIterations; ++iteration) {
107         Fortune fortune(width, height, diagram);
108         fortune.computeDiagram();
109         VoronoiGraph cells = fortune.getVoronoiGraph();
110         vector<Vector2d> newDiagram; newDiagram.reserve(cells.sites.size());
111         for (const auto& site : cells.sites) {
112             double x = 0.;
113             double y = 0.;
114             for (const auto& vertice : site->nodes) {
115                 x += max(min(vertice->pos.x, double(width)), 0.);
116                 y += max(min(vertice->pos.y, double(height)), 0.);
117             }
118             newDiagram.emplace_back(x / site->nodes.size(), y / site->nodes.size
119                                     ());
120         }
121         diagram = newDiagram;
122     }
123     return diagram;
124 }
125 VoronoiDiagram::VoronoiGraph::DualEdge::DualEdge(
126     unsigned int _v1,
127     unsigned int _v2,
128     unsigned int _s1,
129     unsigned int _s2):
130     nodeA(_v1), nodeB(_v2), siteA(_s1), siteB(_s2)
131 {}
132
133 VoronoiDiagram::VoronoiGraph::Site::Site(const Vector2d& _pos): pos(_pos),
134     elevation(0.0)
135 {}
136
137 VoronoiDiagram::VoronoiGraph::Node::Node(const Vector2d& _pos): pos(_pos),
138     elevation(0.0)
139 {}
140
141 VoronoiDiagram::VoronoiGraph::Edge::Edge(
142     Node* _nodeA,
143     Node* _nodeB,
144     Site* _siteA,
145     Site* _siteB):
146     nodeA(_nodeA), nodeB(_nodeB), siteA(_siteA), siteB(_siteB)
147 {}
148
149 VoronoiDiagram::VoronoiGraph::VoronoiGraph(
150     double _width,
151     double _height,
152     const std::vector<Vector2d>& sitesCoordinates,
153     const std::vector<Vector2d>& nodesCoordinates,
154     const std::vector<DualEdge>& edgesData):
155     width(_width), height(_height)

```

```

154 {
155     sites.reserve(sitesCoordinates.size());
156     for (const auto& site : sitesCoordinates)
157         sites.emplace_back(new Site(site));
158
159     nodes.reserve(nodesCoordinates.size());
160     for (const auto& vertex : nodesCoordinates)
161         nodes.emplace_back(new Node(vertex));
162
163     edges.reserve(edgesData.size());
164     for (const auto& edge : edgesData) {
165         Node* nodeA = nodes[edge.nodeA].get();
166         Node* nodeB = nodes[edge.nodeB].get();
167         Site* siteA = sites[edge.siteA].get();
168         Site* siteB = sites[edge.siteB].get();
169         Edge* graphEdge = new Edge(nodeA, nodeB, siteA, siteB);
170         edges.emplace_back(graphEdge);
171
172         nodeA->nodes.push_back(nodeB);
173         nodeB->nodes.push_back(nodeA);
174
175         nodeA->edges.push_back(graphEdge);
176         nodeB->edges.push_back(graphEdge);
177
178         siteA->borders.push_back(siteB);
179         siteB->borders.push_back(siteA);
180
181         siteA->edges.push_back(graphEdge);
182         siteA->edges.push_back(graphEdge);
183
184         siteA->nodes.insert(nodeA);
185         siteA->nodes.insert(nodeB);
186         siteB->nodes.insert(nodeA);
187         siteB->nodes.insert(nodeB);
188     }
189 }
190
191 sf::VertexArray VoronoiDiagram::VoronoiGraph::getEdges() const {
192     sf::Color color = sf::Color::Black;
193     sf::VertexArray lines(sf::PrimitiveType::Lines);
194     for (const auto& edge : edges) {
195         lines.append(sf::Vertex(sf::Vector2f(edge->nodeA->pos), color));
196         lines.append(sf::Vertex(sf::Vector2f(edge->nodeB->pos), color));
197     }
198     return lines;
199 }
200
201 sf::VertexArray VoronoiDiagram::VoronoiGraph::getBeams() const {
202     sf::Color color = sf::Color::Black;
203     sf::VertexArray triangles = getEdges();
204     auto addVertex = [&](const Vector2d& pos){ triangles.append(sf::Vertex(
205         static_cast<sf::Vector2f>(pos), color));};
206     for (const auto& edge : edges) {
207         addVertex(edge->siteA->pos);
208         addVertex(edge->nodeA->pos);

```

```

209     addVertex(edge->siteA->pos);
210     addVertex(edge->nodeB->pos);
211
212     addVertex(edge->siteB->pos);
213     addVertex(edge->nodeA->pos);
214
215     addVertex(edge->siteB->pos);
216     addVertex(edge->nodeA->pos);
217 }
218 return triangles;
219 }
220
221 sf::VertexArray VoronoiDiagram::VoronoiGraph::getDelaunayTriangulation()
222     const {
223     sf::Color color = sf::Color::Black;
224     sf::VertexArray delaunay(sf::PrimitiveType::Lines);
225     for (const auto& edge: edges) {
226         delaunay.append(sf::Vertex(static_cast<sf::Vector2f>(edge->siteA->pos),
227             color));
228         delaunay.append(sf::Vertex(static_cast<sf::Vector2f>(edge->siteB->pos),
229             color));
230     }
231     return delaunay;
232 }
233
234 vector<sf::CircleShape> VoronoiDiagram::VoronoiGraph::getSites(float radius)
235     const {
236     vector<sf::CircleShape> circles; circles.reserve(sites.size());
237     for (const auto& site : sites) {
238         sf::CircleShape circle(radius);
239         circle.setOrigin(sf::Vector2f(radius/2, radius/2));
240         circle.setPosition(static_cast<sf::Vector2f>(site->pos));
241         circle.setFill_color(sf::Color::Black);
242         circles.push_back(circle);
243     }
244     return circles;
245 }
246
247 sf::VertexArray VoronoiDiagram::VoronoiGraph::getCellMap(std::function<sf::
248     Color (double)> colorGradient) const {
249     sf::VertexArray ground(sf::PrimitiveType::Triangles);
250     auto addVertex = [&](Vector2d pos, double elevation) {
251         ground.append(sf::Vertex(static_cast<sf::Vector2f>(pos), colorGradient(
252             elevation)));
253     };
254     for (const auto& edge : edges) {
255         addVertex(edge->siteA->pos, edge->siteA->elevation);
256         addVertex(edge->nodeA->pos, edge->siteA->elevation);
257         addVertex(edge->nodeB->pos, edge->siteA->elevation);
258
259         addVertex(edge->siteB->pos, edge->siteB->elevation);
260         addVertex(edge->nodeA->pos, edge->siteB->elevation);
261         addVertex(edge->nodeB->pos, edge->siteB->elevation);
262     }
263     return ground;
264 }

```

```

259
260 sf::VertexArray VoronoiDiagram::VoronoiGraph::getTriangulationMap(std::
    function<sf::Color (double)> colorGradient) const {
261     sf::VertexArray ground(sf::PrimitiveType::Triangles);
262     auto addVertex = [&](Vector2d pos, double elevation) {
263         ground.append(sf::Vertex(static_cast<sf::Vector2f>(pos), colorGradient(
            elevation)));
264     };
265     for (const auto& edge : edges) {
266         addVertex(edge->siteA->pos, edge->siteA->elevation);
267         addVertex(edge->nodeA->pos, edge->nodeA->elevation);
268         addVertex(edge->nodeB->pos, edge->nodeB->elevation);
269
270         addVertex(edge->siteB->pos, edge->siteB->elevation);
271         addVertex(edge->nodeA->pos, edge->nodeA->elevation);
272         addVertex(edge->nodeB->pos, edge->nodeB->elevation);
273     }
274     return ground;
275 }
276
277 sf::VertexArray VoronoiDiagram::VoronoiGraph::getMixedMap(std::function<sf
    ::Color (double)> colorGradient) const {
278     sf::VertexArray ground(sf::PrimitiveType::Triangles);
279     auto addVertex = [&](Vector2d pos, double elevation) {
280         ground.append(sf::Vertex(static_cast<sf::Vector2f>(pos), colorGradient(
            elevation)));
281     };
282     for (const auto& edge : edges) {
283         addVertex(edge->siteA->pos, edge->siteA->elevation);
284         if (edge->siteA->elevation <= 0.5) {
285             addVertex(edge->nodeA->pos, edge->nodeA->elevation);
286             addVertex(edge->nodeB->pos, edge->nodeB->elevation);
287         } else {
288             addVertex(edge->nodeA->pos, edge->siteA->elevation);
289             addVertex(edge->nodeB->pos, edge->siteA->elevation);
290         }
291
292         addVertex(edge->siteB->pos, edge->siteB->elevation);
293         if (edge->siteB->elevation <= 0.5) {
294             addVertex(edge->nodeA->pos, edge->nodeA->elevation);
295             addVertex(edge->nodeB->pos, edge->nodeB->elevation);
296         } else {
297             addVertex(edge->nodeA->pos, edge->siteB->elevation);
298             addVertex(edge->nodeB->pos, edge->siteB->elevation);
299         }
300     }
301     return ground;
302 }

```

7.0.16 Fortune.h

```
1  #ifndef FORTUNE_PROCEDURE_H
2  #define FORTUNE_PROCEDURE_H
3
4  #include <exception>
5  #include <memory>
6  #include <queue>
7  #include <set>
8
9  #include "SFML\Graphics.hpp"
10
11 #include "GeometricUtility.h"
12 #include "VoronoiDiagram.h"
13
14 class Fortune {
15 public:
16
17     struct Edge;
18     struct Event;
19     struct Tree;
20
21     Fortune(unsigned int, unsigned int, const std::vector<Vector2d>&);
22
23     void computeDiagram();
24     VoronoiDiagram::VoronoiGraph getVoronoiGraph() const;
25
26     struct Edge
27     {
28         Edge();
29         Edge(const Vector2d*, const Vector2d*, const Vector2d*);
30
31         void print() const;
32
33         const Vector2d* start;
34         const Vector2d* end;
35         const Vector2d* cellLeft;
36         const Vector2d* cellRight;
37         Edge* neighbour;
38         Vector2d direction;
39         double f;
40         double g;
41     };
42
43     enum class EventType {
44         None,
45         Site,
46         Circle
47     };
48
49     struct Event
50     {
51         Event(Vector2d*);
52         Event(Vector2d*, Tree*);
53
54         const Vector2d* const site;
```

```

55     const EventType type;
56     Tree* parabola;
57     bool deprecated;
58 };
59
60 struct Tree
61 {
62     Tree();
63     Tree(const Vector2d*);
64     Tree(Tree*, Tree*, Edge*);
65
66     bool isLeaf;
67     const Vector2d* site;
68     Tree* parent;
69     Tree* left;
70     Tree* right;
71     Edge* edge;
72     Event* event;
73 };
74
75 private:
76
77     Fortune(const Fortune&);
78
79     void insertParabola(const Vector2d*);
80     void removeParabola(Tree*, const Vector2d*);
81     void checkCircleEvent(Tree*, double);
82     void finishEdges(Tree*);
83     void pickEdges();
84     static Vector2d* const edgeIntersection(const Fortune::Edge&, const
        Fortune::Edge&);
85
86     Tree* findParabolaIntersection(Tree*, const Vector2d*);
87     Tree* leastCommonAncestry(Tree*, Tree*, Tree*);
88     void replaceBy(Tree*, Tree*);
89     Tree* leftChild(Tree*);
90     Tree* rightChild(Tree*);
91     Tree* leftParent(Tree*);
92     Tree* rightParent(Tree*);
93
94     void deprecateEvent(Tree*);
95     struct maxHeapAxisY { bool operator()(const Event* const, const Event*
        const) const; };
96
97     std::vector<std::unique_ptr<Vector2d>> vertices;
98     std::vector<std::unique_ptr<Edge>> edges;
99
100    double width;
101    double height;
102    std::set<std::unique_ptr<Vector2d>, compVector2dPtr<std::unique_ptr<
        Vector2d>>> sites;
103    Vector2d* const sentinel;
104
105    std::priority_queue<Event* const, std::vector<Event* const>,
        maxHeapAxisY> events;
106    Tree* root;

```

```

107     bool computed;
108 };
109
110 #endif

7.0.17 Fortune.cpp

1 #include <cassert>
2 #include <iostream>
3 #include <map>
4
5 #include "FortuneProcedure.h"
6 #include "GeometricUtility.h"
7
8 Fortune::Fortune(unsigned int _width,
9                 unsigned int _height,
10                 const std::vector<Vector2d>& points):
11 width(static_cast<double>(_width)), height(static_cast<double>(_height)),
    sentinel(new Vector2d(width/2.0, height*100000)), root(nullptr),
    computed(false) {
12     for (auto point: points)
13         sites.insert(std::unique_ptr<Vector2d>(new Vector2d(point)));
14     // sites.insert(std::unique_ptr<Vector2d>(sentinel));
15     for (const auto& site : sites)
16         events.push(new Event(site.get()));
17 }
18
19 void Fortune::computeDiagram() {
20     if (computed)
21         return ;
22     computed = true;
23
24     while (!events.empty()) {
25         Event* event = events.top();
26         events.pop();
27         if (!event->deprecated) {
28             switch (event->type) {
29                 case EventType::Site:
30                     insertParabola(event->site);
31                     break;
32                 case EventType::Circle:
33                     removeParabola(event->parabola, event->site);
34                     break;
35             }
36         }
37         delete event;
38     }
39     finishEdges(root);
40     pickEdges();
41 }
42
43 VoronoiDiagram::VoronoiGraph Fortune::getVoronoiGraph() const {
44     std::vector<Vector2d> sitesCoordinates;
45     sitesCoordinates.reserve(sites.size());
46     std::map<const Vector2d*, unsigned> sitesIndices;
47     unsigned int index = 0;

```



```

48     for (const auto& site: sites) {
49         sitesCoordinates.push_back(*site);
50         sitesIndices[site.get()] = index;
51         index += 1;
52     }
53
54     std::vector<Vector2d> nodesCoordinates;
55     nodesCoordinates.reserve(vertices.size());
56     std::map<const Vector2d*, unsigned> nodesIndices;
57     for (unsigned int node = 0; node < vertices.size(); ++node) {
58         nodesCoordinates.push_back(*vertices[node]);
59         nodesIndices[vertices[node].get()] = node;
60     }
61
62     std::vector<VoronoiDiagram::VoronoiGraph::DualEdge> edgesData;
63     edgesData.reserve(edgesData.size());
64     for (const auto& edge : edges) {
65         unsigned int v1 = nodesIndices[edge->start];
66         unsigned int v2 = nodesIndices[edge->end];
67         unsigned int s1 = sitesIndices[edge->cellLeft];
68         unsigned int s2 = sitesIndices[edge->cellRight];
69         edgesData.emplace_back(v1, v2, s1, s2);
70     }
71
72     return VoronoiDiagram::VoronoiGraph(width, height, sitesCoordinates,
73         nodesCoordinates, edgesData);
74 }
75
76 void Fortune::insertParabola(const Vector2d* site) {
77     if (root == nullptr) {
78         root = new Tree(site);
79         return;
80     }
81     Tree* const parabola = findParabolaIntersection(root, site);
82     deprecateEvent(parabola);
83
84     Vector2d* vertex = degenerateIntersection(*parabola->site, *site);
85     vertices.emplace_back(vertex);
86
87     Edge* edgeLeft = new Edge(vertex, parabola->site, site);
88     Edge* edgeRight = new Edge(vertex, site, parabola->site);
89     edgeLeft->neighbour = edgeRight;
90     edges.emplace_back(edgeLeft);
91
92     Tree* pLeft = new Tree(parabola->site);
93     Tree* pMid = new Tree(site);
94     Tree* pRight = new Tree(parabola->site);
95
96     replaceBy(parabola, new Tree(new Tree(pLeft, pMid, edgeLeft), pRight,
97         edgeRight));
98     checkCircleEvent(pLeft, site->y);
99     checkCircleEvent(pRight, site->y);
100 }
101

```

```

102 void Fortune::removeParabola(Tree* parabola, const Vector2d* site) {
103     Tree* const lp = leftParent(parabola);
104     Tree* const rp = rightParent(parabola);
105     Tree* const previousLeaf = leftChild(lp);
106     Tree* const nextLeaf = rightChild(rp);
107
108     deprecateEvent(previousLeaf);
109     deprecateEvent(nextLeaf);
110
111     Vector2d* vertex = edgeIntersection(*lp->edge, *rp->edge);
112     vertices.emplace_back(vertex);
113
114     lp->edge->end = vertex;
115     rp->edge->end = vertex;
116
117     Edge* edge = new Edge(vertex, previousLeaf->site, nextLeaf->site);
118     edges.emplace_back(edge);
119
120     Tree* const higher = leastCommonAncestry(parabola, lp, rp);
121     higher->edge = edge;
122
123     Tree* const father = parabola->parent;
124     Tree* const brother = (father->left == parabola)? father->right : father
        ->left;
125     replaceBy(father, brother);
126     delete parabola;
127
128     checkCircleEvent(previousLeaf, site->y);
129     checkCircleEvent(nextLeaf, site->y);
130 }
131
132 void Fortune::checkCircleEvent(Tree* parabola, double sweepLine) {
133     Tree* const lp = leftParent(parabola);
134     Tree* const rp = rightParent(parabola);
135
136     Tree* const previousLeaf = leftChild(lp);
137     Tree* const nextLeaf = rightChild(rp);
138
139     if (previousLeaf == nullptr
140         || nextLeaf == nullptr
141         || previousLeaf->site == nextLeaf->site)
142         return ;
143
144     Vector2d* const intersection = edgeIntersection(*lp->edge, *rp->edge);
145     if (intersection == nullptr)
146         return ;
147
148     double d = distance2(*parabola->site, *intersection);
149     if (intersection->y - d >= sweepLine) {
150         delete intersection;
151         return ;
152     }
153     vertices.emplace_back(intersection);
154
155     Vector2d* const circle = new Vector2d(intersection->x, intersection->y -
        d);

```

```

156     vertices.emplace_back(circle);
157     Event* circleEvent = new Event(circle, parabola);
158     parabola->event = circleEvent;
159     events.push(circleEvent);
160 }
161
162 void Fortune::finishEdges(Tree* node) {
163     if (!node->isLeaf) {
164         double mx = (node->edge->direction.x >= 0.)?
165             std::max(width, node->edge->start->x + 10.):
166             std::min(0., node->edge->start->x - 10.);
167
168         Vector2d* end = new Vector2d(mx, mx*node->edge->f + node->edge->g);
169         vertices.emplace_back(end);
170         node->edge->end = end;
171
172         finishEdges(node->left);
173         finishEdges(node->right);
174     }
175     delete node;
176 }
177
178 void Fortune::pickEdges() {
179     for (const auto& edge : edges) {
180         if (edge->neighbour != nullptr) {
181             edge->start = edge->neighbour->end;
182             delete edge->neighbour;
183             edge->neighbour = nullptr;
184         }
185     }
186 }
187
188 void Fortune::deprecateEvent(Tree* leaf) {
189     if (leaf->event == nullptr)
190         return;
191     leaf->event->deprecated = true;
192     leaf->event = nullptr;
193 }
194
195 Vector2d* const Fortune::edgeIntersection(const Fortune::Edge& left, const
    Fortune::Edge& right) {
196     const double epsilon = 1e-10;
197     if (fabs(left.f-right.f) <= epsilon)
198         return nullptr;
199
200     double x = (right.g - left.g)/(left.f - right.f);
201     double y = right.f*x + right.g;
202
203     if (sgn(x - left.start->x)*sgn(left.direction.x) < 0
204         || sgn(y - left.start->y)*sgn(left.direction.y) < 0
205         || sgn(x - right.start->x)*sgn(right.direction.x) < 0
206         || sgn(y - right.start->y)*sgn(right.direction.y) < 0)
207         return nullptr;
208
209     return new Vector2d(x, y);
210 }

```

```

211
212 bool Fortune::maxHeapAxisY::operator()(const Event* const left, const Event
    * const right) const {
213     if (left->site->y != right->site->y)
214         return left->site->y < right->site->y;
215     return left->site->x < right->site->x;
216 }
217
218 void Fortune::replaceBy(Tree* old, Tree* subTree) {
219     if (old == root) {
220         delete root;
221         root = subTree;
222     } else {
223         Tree* const father = old->parent;
224         subTree->parent = father;
225         Tree** const son = (father->left == old)? &father->left : &father->
            right;
226         *son = subTree;
227         delete old;
228     }
229 }
230
231 Fortune::Tree* Fortune::findParabolaIntersection(Tree* node, const Vector2d
    * degenerate) {
232     while (!node->isLeaf) {
233         Tree* const left = leftChild(node);
234         Tree* const right = rightChild(node);
235         double xIntersection = xParabolaIntersection(degenerate->y, *left->site
            , *right->site);
236         if (degenerate->x < xIntersection)
237             node = node->left;
238         else
239             node = node->right;
240     }
241
242     return node;
243 }
244
245 Fortune::Tree* Fortune::leastCommonAncestry(Tree* start, Tree* candidate1,
    Tree* candidate2) {
246     if (start == root)
247         return root;
248
249     if (start->parent == candidate1)
250         return candidate2;
251     else
252         return candidate1;
253 }
254
255 Fortune::Tree* Fortune::leftChild(Tree* node) {
256     if (node == nullptr)
257         return nullptr;
258     node = node->left;
259     while (!node->isLeaf)
260         node = node->right;
261     return node;

```

```

262 }
263
264 Fortune::Tree* Fortune::rightChild(Tree* node) {
265     if (node == nullptr)
266         return nullptr;
267     node = node->right;
268     while (!node->isLeaf)
269         node = node->left;
270     return node;
271 }
272
273 Fortune::Tree* Fortune::leftParent(Tree* node) {
274     Tree* parent = node->parent;
275     while (parent != nullptr && parent->left == node) {
276         node = parent;
277         parent = parent->parent;
278     }
279     return parent;
280 }
281
282 Fortune::Tree* Fortune::rightParent(Tree* node) {
283     Tree* parent = node->parent;
284     while (parent != nullptr && parent->right == node) {
285         node = parent;
286         parent = parent->parent;
287     }
288     return parent;
289 }
290
291 Fortune::Edge::Edge():
292     start(nullptr),
293     end(nullptr),
294     cellLeft(nullptr),
295     cellRight(nullptr),
296     neighbour(nullptr){}
297 Fortune::Edge::Edge(const Vector2d* _start,
298                     const Vector2d* _cellLeft,
299                     const Vector2d* _cellRight):
300     start(_start),
301     end(nullptr),
302     cellLeft(_cellLeft),
303     cellRight(_cellRight),
304     neighbour(nullptr),
305     direction(cellRight->y - cellLeft->y, cellLeft->x - cellRight->x) {
306     const double epsilon = 1e-10;
307     if (fabs(cellLeft->y - cellRight->y) <= epsilon) {
308         double verticality = sgn(cellRight->x - cellLeft->x);
309         f = 1/epsilon;
310         direction.x = -epsilon*verticality;
311     } else {
312         f = (cellRight->x - cellLeft->x)/(cellLeft->y - cellRight->y);
313     }
314
315     g = start->y - f*start->x;
316 }
317

```

```

318 void Fortune::Edge::print() const {
319     std::cout << "(" << start->x << ", " << start->y << ")_to_";
320     if (end != nullptr)
321         std::cout << "(" << end->x << ", " << end->y << ")\n";
322     else
323         std::cout << "NULL\n";
324 }
325
326 Fortune::Event::Event(Vector2d* _site):
327     site(_site),
328     type(EventType::Site),
329     parabola(nullptr),
330     deprecated(false){}
331 Fortune::Event::Event(Vector2d* _site, Tree* _parabola):
332     site(_site),
333     type(EventType::Circle),
334     parabola(_parabola),
335     deprecated(false){}
336
337 Fortune::Tree::Tree():
338     isLeaf(false),
339     site(nullptr),
340     parent(nullptr),
341     left(nullptr),
342     right(nullptr),
343     edge(nullptr),
344     event(nullptr){}
345 Fortune::Tree::Tree(const Vector2d* _site):
346     isLeaf(true),
347     site(_site),
348     parent(nullptr),
349     left(nullptr),
350     right(nullptr),
351     edge(nullptr),
352     event(nullptr){}
353 Fortune::Tree::Tree(Tree* _left, Tree* _right, Edge* _edge):
354     isLeaf(false),
355     site(nullptr),
356     parent(nullptr),
357     left(_left),
358     right(_right),
359     edge(_edge),
360     event(nullptr) {
361         if (left != nullptr)
362             left->parent = this;
363         if (right != nullptr)
364             right->parent = this;
365     }

```

7.0.18 GeometricUtility.h

```
1 #ifndef GEOMETRIC_UTILITY_HPP
2 #define GEOMETRIC_UTILITY_HPP
3
4 #include "SFML\Graphics.hpp"
5 // #include "FortuneProcedure.h"
6
7 template <typename T> int sgn(T val) {
8     return (val >= T(0)) ? 1 : -1;
9 }
10
11 typedef sf::Vector2<double> Vector2d;
12
13 struct compVector2d {
14     bool operator()(const Vector2d&, const Vector2d&) const;
15 };
16
17 template<typename T>
18 struct compVector2dPtr {
19     bool operator()(const T& left, const T& right) const {
20         if (left->y != right->y)
21             return left->y < right->y;
22         return left->x < right->x;
23     }
24 };
25
26 double norm2(Vector2d);
27 double distance2(Vector2d, Vector2d);
28 double dot(Vector2d, Vector2d);
29 Vector2d normalize(Vector2d);
30
31 Vector2d* degenerateIntersection(const Vector2d&, const Vector2d&);
32 double xParabolaIntersection(double, const Vector2d&, const Vector2d&);
33 double yParabolaIntersection(const Vector2d&, const Vector2d&);
34
35 #endif
```

7.0.19 GeometricUtility.cpp

```
1 #include <algorithm>
2 #include <iostream>
3 #include <cmath>
4
5 #include "GeometricUtility.h"
6 #include "FortuneProcedure.h"
7
8 using namespace sf;
9 using namespace std;
10
11 bool compVector2d::operator()(const Vector2d& a, const Vector2d& b) const {
12     return (a.y!=b.y)? a.y<b.y : a.x<b.x;
13 }
14
15 double distance2(Vector2d left, Vector2d right) {
16     return norm2(right - left);
17 }
```

```

17 }
18
19 double norm2(Vector2d vect) {
20     return sqrt(dot(vect, vect));
21 }
22
23 double dot(Vector2d vect1, Vector2d vect2) {
24     return vect1.x*vect2.x + vect1.y*vect2.y;
25 }
26
27 Vector2d normalize(Vector2d vect) {
28     double n = norm2(vect);
29     if (n <= 1e-11)
30         return vect;
31     return vect/n;
32 }
33
34 static double distanceParabola(const Vector2d& site, double sweepLine)
35 { return 2*(site.y - sweepLine); }
36
37 static double computeB(const Vector2d& site, double dp)
38 { return -2*site.x/dp; }
39
40 static double computeC(const Vector2d& site, double dp, double y)
41 { return y+dp/4+site.x*site.x/dp; }
42
43 double xParabolaIntersection(double sweepLine, const Vector2d& left, const
    Vector2d& right) {
44     const double epsilon = 1e-10;
45     if (fabs(left.y-right.y) <= epsilon)
46         return (left.x+right.x)/2.;
47     if (fabs(left.x-right.x) <= epsilon)
48         return left.x;
49
50     if (fabs(left.y-sweepLine) <= epsilon)
51         return left.x;
52     if (fabs(right.y-sweepLine) <= epsilon)
53         return right.x;
54
55     double dpLeft = distanceParabola(left, sweepLine);
56     double dpRight = distanceParabola(right, sweepLine);
57
58     double a = 1/dpLeft - 1/dpRight;
59     double b = computeB(left, dpLeft) - computeB(right, dpRight);
60     double c = computeC(left, dpLeft, sweepLine) - computeC(right, dpRight,
        sweepLine);
61
62     double delta = b*b - 4*a*c;
63     double x1 = (-b + sqrt(delta))/(2*a);
64     double x2 = (-b - sqrt(delta))/(2*a);
65
66     if (left.y < right.y)
67         return max(x1, x2);
68     else
69         return min(x1, x2);
70 }

```



```

71
72 double yParabolaIntersection(const Vector2d& parabola, const Vector2d&
    degenerate) {
73     const double epsilon = 1e-10;
74     if (fabs(parabola.y - degenerate.y) <= epsilon)
75         return degenerate.y;
76
77     const double x = degenerate.x;
78     double dp = distanceParabola(parabola, degenerate.y);
79     double a = 1/dp;
80     double b = computeB(parabola, dp);
81     double c = computeC(parabola, dp, degenerate.y);
82     return a*x*x+b*x+c;
83 }
84
85 Vector2d* degenerateIntersection(const Vector2d& parabola, const Vector2d&
    degenerate) {
86     const double epsilon = 1e-10;
87     double x = (fabs(parabola.y - degenerate.y) <= epsilon)?
88         (degenerate.x+parabola.x)/2 : degenerate.x;
89     double y = yParabolaIntersection(parabola, Vector2d(x, degenerate.y));
90
91     return new Vector2d(x, y);
92 }

```

7.0.20 PerlinNoise.h

```
1 #ifndef PERLIN_NOISE
2 #define PERLIN_NOISE
3
4 #include <functional>
5
6 #include "VoronoiDiagram.h"
7
8 std::vector<std::vector<Vector2d>> getGradientField(unsigned int, unsigned
    int);
9 std::vector<std::vector<Vector2d>> getCoord(unsigned int, unsigned int,
    double);
10
11 void noise(VoronoiDiagram::VoronoiGraph&, std::function<double (double,
    double, double)>, double, double);
12 void perlinNoise(VoronoiDiagram::VoronoiGraph&, std::function<double (
    double, double, double)>, double, unsigned int, double);
13 void computeAverageSiteElevation(VoronoiDiagram::VoronoiGraph&);
14 void computeMap(VoronoiDiagram::VoronoiGraph&, std::function<double (double
    , double, double)>, double, unsigned int, double);
15
16 double linearInterpolate(double, double, double);
17 double quinticInterpolate(double, double, double);
18
19 struct GradientParameters {
20     GradientParameters(sf::Color, sf::Color, double);
21     sf::Color low;
22     sf::Color high;
23     double ratio;
24 };
25
26 std::function<sf::Color (double)> linearGradient(const std::vector<
    GradientParameters>&);
27 std::function<sf::Color (double)> groundGradient();
28 std::function<sf::Color (double)> islandGradient();
29
30 void makeIsland(VoronoiDiagram::VoronoiGraph&);
31
32 namespace CudaPerlinNoise {
33     void PerlinNoise(VoronoiDiagram::VoronoiGraph&, double, unsigned int,
        double);
34     void computeParallelMap(VoronoiDiagram::VoronoiGraph&, double, unsigned
        int, double);
35 }
36
37 #endif
```

7.0.21 PerlinNoise.cpp

```
1 #include <chrono>
2 #include <iostream>
3 #include <random>
4
5 #include "PerlinNoise.h"
6 #include "PerlinNoise.cuh"
```

```

7 #include "VoronoiDiagram.h"
8 #include "GeometricUtility.h"
9
10 using namespace std;
11 using namespace VoronoiDiagram;
12
13 namespace {
14     const unsigned int seed = static_cast<unsigned int>(std::chrono::
        system_clock::now().time_since_epoch().count());
15     std::default_random_engine generator (seed);
16     std::uniform_real_distribution<double> uniform(-1., 1.);
17 }
18
19 vector<vector<Vector2d>> getGradientField(unsigned int width, unsigned int
    height) {
20     vector<vector<Vector2d>> gradientField(height, vector<Vector2d>(width));
21     for (auto& line : gradientField) {
22         for (auto& vect : line) {
23             vect = normalize(Vector2d(uniform(generator), uniform(generator)));
24         }
25     }
26     return gradientField;
27 }
28
29 vector<vector<Vector2d>> getCoord(unsigned int width, unsigned int height,
    double squareSize) {
30     vector<vector<Vector2d>> coord(height, vector<Vector2d>(width));
31     for (unsigned int y = 0; y < height; ++y) {
32         for (unsigned int x = 0; x < width; ++x) {
33             coord[y][x] = Vector2d(x*squareSize, y*squareSize);
34         }
35     }
36     return coord;
37 }
38
39 void noise(
40     VoronoiDiagram::VoronoiGraph& graph,
41     function<double (double, double, double)> interpolate,
42     double frequency, double amplitude)
43 {
44     using sf::Vector2i;
45     enum Cardinal {
46         SouthEast = 0,
47         SouthWest = 1,
48         NorthWest = 2,
49         NorthEast = 3
50     };
51     vector<Vector2i> corners(4);
52     corners[SouthEast] = Vector2i(1, 0);
53     corners[SouthWest] = Vector2i(0, 0);
54     corners[NorthWest] = Vector2i(0, 1);
55     corners[NorthEast] = Vector2i(1, 1);
56
57     double squareSize = graph.height / frequency;
58     unsigned int height = static_cast<unsigned int>(ceil(frequency) + 1);

```

```

59 unsigned int width = static_cast<unsigned int>(ceil(graph.width/
    squareSize) + 1);
60 vector<vector<Vector2d>> gradientField = getGradientField(width, height);
61 vector<vector<Vector2d>> coord = getCoord(width, height, squareSize);
62 for (auto& node : graph.nodes) {
63     double x = node->pos.x;
64     if (x < 0.0) x = squareSize/2;
65     if (x >= graph.width) x = graph.width - squareSize/2;
66
67     double y = node->pos.y;
68     if (y < 0.0) y = squareSize/2;
69     if (y >= graph.height) y = graph.height - squareSize/2;
70
71     unsigned int xCell = static_cast<unsigned int>(x / squareSize);
72     unsigned int yCell = static_cast<unsigned int>(y / squareSize);
73     vector<double> values;
74     for (const auto& corner : corners) {
75         Vector2d cornerPos = coord[yCell+corner.y][xCell+corner.x];
76         Vector2d grad = gradientField[yCell+corner.y][xCell+corner.x];
77         double value = dot(normalize(cornerPos - node->pos), grad);
78         values.push_back(value);
79     }
80
81     Vector2d cell = coord[yCell][xCell];
82
83     double xReal = (x - cell.x)/squareSize;
84     double y1 = interpolate(xReal, values[SouthWest], values[SouthEast]);
85     double y2 = interpolate(xReal, values[NorthWest], values[NorthEast]);
86
87     double yReal = (y - cell.y)/squareSize;
88     double z = interpolate(yReal, y1, y2);
89
90     node->elevation += z*amplitude;
91 }
92 }
93
94 void perlinNoise(
95     VoronoiDiagram::VoronoiGraph& graph,
96     std::function<double (double, double, double)> interpolate,
97     double fundamental,
98     unsigned int nbOctaves,
99     double persistence)
100 {
101     double frequency = fundamental;
102     double amplitude = 1.;
103     double amplitudeMax = 0.;
104     for (unsigned int octave = 0; octave < nbOctaves; ++octave) {
105         amplitudeMax += amplitude;
106         noise(graph, interpolate, frequency, amplitude);
107         frequency *= 2.5;
108         amplitude *= persistence;
109     }
110
111     for (auto& node : graph.nodes) {
112         node->elevation /= amplitudeMax;
113         node->elevation = (node->elevation+1)/2;

```

```

114     }
115 }
116
117 void computeAverageSiteElevation(VoronoiDiagram::VoronoiGraph& graph) {
118     for (auto& site : graph.sites) {
119         double sum = 0.;
120         for (const auto& node : site->nodes) {
121             sum += node->elevation;
122         }
123         if (!site->nodes.empty())
124             site->elevation = sum/site->nodes.size();
125     }
126 }
127
128 void computeMap(
129     VoronoiDiagram::VoronoiGraph& graph,
130     std::function<double (double, double, double)> interpolate,
131     double fundamental,
132     unsigned int nbOctaves,
133     double persistence) {
134     perlinNoise(graph, interpolate, fundamental, nbOctaves, persistence);
135     makeIsland(graph);
136     computeAverageSiteElevation(graph);
137 }
138
139 double linearInterpolate(double t, double a, double b) {
140     return a + t*(b-a);
141 }
142
143 double quinticInterpolate(double t, double a, double b) {
144     return linearInterpolate(t*t*t*(6*t*t-15*t+10), a, b);
145 }
146
147 function<sf::Color (double)> linearGradient(const vector<GradientParameters
148     >& parameters) {
149     return [=](double elevation) -> sf::Color {
150         double sum = 0.;
151         for (const auto& param : parameters) {
152             double coeff = (elevation-sum)/param.ratio;
153             sum += param.ratio;
154             if (elevation > sum)
155                 continue;
156             int deltaRed = int(param.high.r) - int(param.low.r);
157             int deltaGreen = int(param.high.g) - int(param.low.g);
158             int deltaBlue = int(param.high.b) - int(param.low.b);
159             sf::Uint8 red = static_cast<sf::Uint8>(param.low.r + coeff*deltaRed);
160             sf::Uint8 green = static_cast<sf::Uint8>(param.low.g + coeff*
161                 deltaGreen);
162             sf::Uint8 blue = static_cast<sf::Uint8>(param.low.b + coeff*deltaBlue
163                 );
164             return sf::Color(red, green, blue);
165         }
166         return parameters.back().high;
167     };
168 }

```

```

167 GradientParameters::GradientParameters(sf::Color _low, sf::Color _high,
      double _ratio):
168 low(_low), high(_high), ratio(_ratio){}
169
170 function<sf::Color (double)> groundGradient() {
171     vector<GradientParameters> param;
172     param.emplace_back(sf::Color(0, 98, 145), sf::Color(0, 98, 145), 0.2);
173     param.emplace_back(sf::Color(0, 98, 145), sf::Color(0, 162, 255), 0.2);
174     param.emplace_back(sf::Color(0, 130, 0), sf::Color(0, 75, 0), 0.2);
175     param.emplace_back(sf::Color(192, 82, 0), sf::Color(100, 40, 0), 0.15);
176     param.emplace_back(sf::Color(64, 64, 64), sf::Color(255, 255, 255), 0.15);
      ;
177     param.emplace_back(sf::Color::White, sf::Color::White, 1);
178     return linearGradient(param);
179 }
180
181 function<sf::Color (double)> islandGradient() {
182     vector<GradientParameters> param;
183     param.emplace_back(sf::Color(0, 98, 145), sf::Color(0, 98, 145), 0.25);
184     param.emplace_back(sf::Color(0, 98, 145), sf::Color(0, 162, 255), 0.25);
185     param.emplace_back(sf::Color(200, 150, 0), sf::Color(200, 150, 0), 0.03);
186     param.emplace_back(sf::Color(130, 130, 0), sf::Color(0, 75, 0), 0.2);
187     param.emplace_back(sf::Color(192, 82, 0), sf::Color(100, 40, 0), 0.12);
188     param.emplace_back(sf::Color(64, 64, 64), sf::Color(255, 255, 255), 0.18);
      ;
189     param.emplace_back(sf::Color::White, sf::Color::White, 1);
190     return linearGradient(param);
191 }
192
193 void makeIsland(VoronoiDiagram::VoronoiGraph& graph) {
194     const double a = 2.;
195     const double b = 0.4;
196     Vector2d center(graph.width/2, graph.height/2);
197     double rMax = distance2(center, Vector2d(0.,0.));
198     for (const auto& node: graph.nodes) {
199         double r = distance2(center, node->pos)/rMax; // between 0 and 1
200         node->elevation *= (exp(-a*r)+b);
201     }
202 }
203
204 void CudaPerlinNoise::PerlinNoise(
205     VoronoiDiagram::VoronoiGraph& graph,
206     double fundamental,
207     unsigned int nbOctaves,
208     double persistence) {
209
210     unique_ptr<float []> h_x(new float[graph.nodes.size()]);
211     unique_ptr<float []> h_y(new float[graph.nodes.size()]);
212     for (unsigned int node = 0; node < graph.nodes.size(); ++node) {
213         h_x[node] = static_cast<float>(graph.nodes[node]->pos.x);
214         h_y[node] = static_cast<float>(graph.nodes[node]->pos.y);
215     }
216
217     unique_ptr<float []> h_z(new float[graph.nodes.size()]);
218     CUDA_PerlinNoise(
219         float(graph.width), float(graph.height), graph.nodes.size(),

```

```

220     h_x.get(), h_y.get(), h_z.get(),
221     float(fundamental), nbOctaves, float(persistence));
222
223     for (unsigned int node = 0; node < graph.nodes.size(); ++node)
224         graph.nodes[node]->elevation = static_cast<double>(h_z[node]);
225 }
226
227 void CudaPerlinNoise::computeParallelMap(
228     VoronoiDiagram::VoronoiGraph& graph,
229     double fundamental,
230     unsigned int nbOctaves,
231     double persistence) {
232     CudaPerlinNoise::PerlinNoise(graph, fundamental, nbOctaves, persistence
233         );
234     makeIsland(graph);
235     computeAverageSiteElevation(graph);
236 }

```

7.0.22 PerlinNoise.cuh

```
1 #ifndef CUDA_PERLIN_NOISE
2 #define CUDA_PERLIN_NOISE
3
4 void CUDA_PerlinNoise(float , float , unsigned int , float*, float*, float*,
    float , unsigned int , float);
5
6 #endif
```

7.0.23 PerlinNoise.cu

```
1 #include <cmath>
2
3 #include <cuda.h>
4 #include <cuda_runtime.h>
5
6 #include "PerlinNoise.cuh"
7
8 __device__
9 unsigned int hash1(unsigned int a)
10 {
11     a = (a+0x7ed55d16) + (a<<12);
12     a = (a^0xc761c23c) ^ (a>>19);
13     a = (a+0x165667b1) + (a<<5);
14     a = (a+0xd3a2646c) ^ (a<<9);
15     a = (a+0xfd7046c5) + (a<<3);
16     a = (a^0xb55a4f09) ^ (a>>16);
17     return a;
18 }
19
20 __device__
21 unsigned int hash2(unsigned int a)
22 {
23     a = (a ^ 61) ^ (a >> 16);
24     a = a + (a << 3);
25     a = a ^ (a >> 4);
26     a = a * 0x27d4eb2d;
27     a = a ^ (a >> 15);
28     return a;
29 }
30
31 __device__
32 unsigned int hashPaire(unsigned int a, unsigned int b) {
33     return (a >= b)? a*a + a + b : a+ b*b;
34 }
35
36 __device__
37 float dot(float2 v1, float2 v2) {
38     return v1.x*v2.x + v1.y*v2.y;
39 }
40
41 __device__
42 float norm2(float2 v) {
43     return sqrt(dot(v, v));
44 }
```



```

45
46 __device__
47 float distance2(float2 v1, float2 v2) {
48     return norm2(make_float2(v1.x - v2.x, v1.y - v2.y));
49 }
50
51 __device__
52 float2 normalize(float2 v) {
53     float n = norm2(v);
54     if (n <= 1e-7f)
55         return v;
56     v.x /= n;
57     v.y /= n;
58     return v;
59 }
60
61 __device__
62 float2 getGradient(unsigned int xCell, unsigned int yCell) {
63     unsigned int hashX = hash1(hashPaire(xCell, yCell));
64     unsigned int hashY = hash1(hashPaire(yCell, hashX%(1 << 16)));
65     float x = float(hashX % (1 << 10)) - (1 << 9);
66     float y = float(hashY % (1 << 10)) - (1 << 9);
67     return normalize(make_float2(x, y));
68 }
69
70 __device__
71 float interpolate(float t, float a, float b) {
72     t = t*t*t*(6*t*t-15*t+10);
73     return a + t*(b-a);
74 }
75
76 __global__
77 void CUDA_noise(
78     float width,
79     float height,
80     unsigned int nbNodes,
81     float* d_x,
82     float* d_y,
83     float* d_z,
84     float frequency,
85     float amplitude) {
86
87     const unsigned int node = blockIdx.x*blockDim.x + threadIdx.x;
88     if (node >= nbNodes)
89         return ;
90
91     const float squareSize = height/frequency;
92
93     float x = d_x[node];
94     if (x < 0.f) x = squareSize/2;
95     if (x >= width) x = width - squareSize/2;
96
97     float y = d_y[node];
98     if (y < 0.f) y = squareSize / 2;
99     if (y >= height) y = height - squareSize/2;
100

```

```

101 unsigned int xCell = x/squareSize;
102 unsigned int yCell = y/squareSize;
103
104 const unsigned int directions[4][2] = {{0, 0}, {0, 1}, {1, 0}, {1, 1}};
105 float values[4];
106 for (unsigned int dir = 0; dir < 4; ++dir) {
107     unsigned int xCorner = xCell+directions[dir][0];
108     unsigned int yCorner = yCell+directions[dir][1];
109     float2 gradient = getGradient(xCorner, yCorner);
110     float2 pos = make_float2(xCorner*squareSize, yCorner*squareSize);
111     float2 arrow = make_float2(pos.x - x, pos.y - y);
112     values[dir] = dot(normalize(arrow), gradient);
113 }
114
115 float xReal = (x - xCell*squareSize) / squareSize;
116 float y1 = interpolate(xReal, values[0], values[2]);
117 float y2 = interpolate(xReal, values[1], values[3]);
118
119 float yReal = (y - yCell*squareSize) / squareSize;
120 float z = interpolate(yReal, y1, y2);
121 d_z[node] += amplitude*z;
122 }
123
124 __host__
125 void CUDA_PerlinNoise(
126     float width,
127     float height,
128     unsigned int nbNodes,
129     float* h_x,
130     float* h_y,
131     float* h_z,
132     float fundamental,
133     unsigned int nbOctaves,
134     float persistence) {
135     const unsigned int nbBytes = sizeof(float)*nbNodes;
136
137     float* d_x;
138     cudaMalloc(&d_x, nbBytes);
139     cudaMemcpy(d_x, h_x, nbBytes, cudaMemcpyHostToDevice);
140
141     float* d_y;
142     cudaMalloc(&d_y, nbBytes);
143     cudaMemcpy(d_y, h_y, nbBytes, cudaMemcpyHostToDevice);
144
145     float* d_z;
146     cudaMalloc(&d_z, nbBytes);
147     cudaMemset(d_z, 0, nbBytes);
148
149     float frequency = fundamental;
150     float amplitude = 1.;
151     float amplitudeMax = 0.;
152     for (unsigned int octave = 0; octave < nbOctaves; ++octave) {
153         amplitudeMax += amplitude;
154         const unsigned int blockSize = 128;
155         const dim3 gridSize((nbNodes+blockSize-1)/blockSize);

```

```

156     CUDA_noise<<<gridSize, blockSize>>>(width, height, nbNodes, d_x, d_y,
      d_z, frequency, amplitude);
157     frequency *= 2.5;
158     amplitude *= persistence;
159 }
160
161 cudaMemcpy(h_z, d_z, nbBytes, cudaMemcpyDeviceToHost);
162 for (unsigned int node = 0; node < nbNodes; ++node) {
163     h_z[node] /= amplitudeMax;
164     h_z[node] = (h_z[node]+1)/2;
165 }
166
167 cudaFree(d_x);
168 cudaFree(d_y);
169 cudaFree(d_z);
170 }

```

7.0.24 VoronoiDiagram.cuh

```
1 #ifndef VORONOI_DIAGRAM_CUDA
2 #define VORONOI_DIAGRAM_CUDA
3
4 namespace CUDA_VoronoiDiagram {
5
6 void naiveParallelGeneration
7     (unsigned int, unsigned int, unsigned int* const, unsigned int,
8      const unsigned int* const, const unsigned int* const);
9
10 }
11
12 #endif
```

7.0.25 VoronoiDiagram.cu

```
1 #include "VoronoiDiagram.cuh"
2
3 #include <cuda.h>
4 #include <cuda_runtime.h>
5
6 __device__
7 unsigned int distance2(unsigned int dx, unsigned int dy)
8 {
9     return dx*dx + dy*dy;
10 }
11
12 __global__
13 void naiveGeneration
14     (unsigned int width, unsigned int height,
15      unsigned int * const d_index,
16      unsigned int nbPoints,
17      const unsigned int* const d_x,
18      const unsigned int* const d_y)
19 {
20     const unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
21     const unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;
22     const unsigned int pixel = y*width + x;
23     if (x >= width || y >= height)
24         return ;
25
26     unsigned int bestPoint = 0;
27     unsigned int bestDistance = distance2(d_x[0]-x, d_y[0]-y);
28     for (unsigned int point = 1; point < nbPoints; ++point)
29     {
30         unsigned int distance = distance2(d_x[point]-x, d_y[point]-y);
31         if (distance < bestDistance)
32         {
33             bestDistance = distance;
34             bestPoint = point;
35         }
36     }
37
38     d_index[pixel] = bestPoint;
39 }
```

```

40
41 void CUDA_VoronoiDiagram::naiveParallelGeneration
42     (unsigned int width, unsigned int height,
43      unsigned int * const h_index,
44      unsigned int nbPoints,
45      const unsigned int* const h_x,
46      const unsigned int* const h_y)
47 {
48     const unsigned int nbBytesVertex = width*height*sizeof(unsigned int);
49     const unsigned int nbBytesPoint = nbPoints*sizeof(unsigned int);
50
51     unsigned int* d_index = nullptr;
52     cudaMalloc(&d_index, nbBytesVertex);
53
54     unsigned int* d_x = nullptr;
55     cudaMalloc(&d_x, nbBytesPoint);
56     cudaMemcpy(d_x, h_x, nbBytesPoint, cudaMemcpyHostToDevice);
57
58     unsigned int* d_y = nullptr;
59     cudaMalloc(&d_y, nbBytesPoint);
60     cudaMemcpy(d_y, h_y, nbBytesPoint, cudaMemcpyHostToDevice);
61
62     const dim3 blockSize(32, 32);
63     const dim3 gridSize((width+blockSize.x-1)/blockSize.x, (height+blockSize.
        y-1)/blockSize.y);
64     naiveGeneration<<<gridSize, blockSize>>>(width, height, d_index, nbPoints
        , d_x, d_y);
65
66     cudaMemcpy(h_index, d_index, nbBytesVertex, cudaMemcpyDeviceToHost);
67
68     cudaFree(d_index);
69     cudaFree(d_x);
70     cudaFree(d_y);
71 }

```