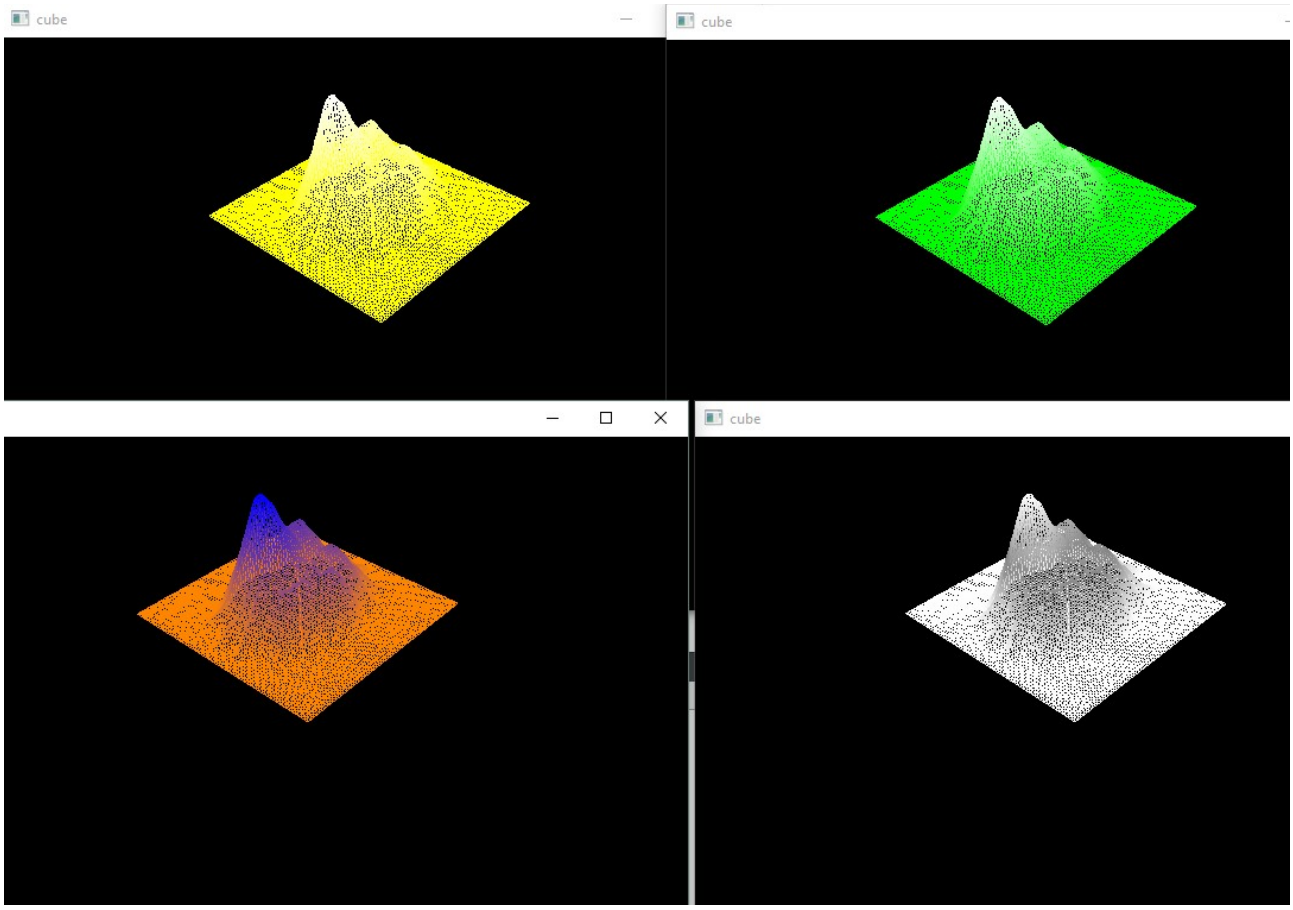
Synchroniser les fenêtres entre elles



L'application principale ouvre 4 fenêtres qui instancient chacune un plan ayant une hauteur basée sur une heightmap. Une classe Calendar gère le timer et la simulation de saison. Chaque instance de MainWidget est connectée au Calendar, qui va envoyé un signal lorsque la saison change.

```
Calendar calendar(5);

MainWidget widget1; widget1.setSeason(0);
widget1.show();
MainWidget widget2; widget2.setSeason(1);
widget2.show();
MainWidget widget3; widget3.setSeason(2);
widget3.show();
MainWidget widget4; widget4.setSeason(3);
widget4.show();

//calendar to widgets
QObject::connect(&calendar, SIGNAL(seasonChanged()),
                 &widget1, SLOT(changeSeason()));
QObject::connect(&calendar, SIGNAL(seasonChanged()),
                 &widget2, SLOT(changeSeason()));
QObject::connect(&calendar, SIGNAL(seasonChanged()),
                 &widget3, SLOT(changeSeason()));
QObject::connect(&calendar, SIGNAL(seasonChanged()),
                 &widget4, SLOT(changeSeason()));

//widgets to calendar
QObject::connect(&widget1, SIGNAL(seasonChanged(int)),
                 &calendar, SLOT(season(int)));
QObject::connect(&widget2, SIGNAL(seasonChanged(int)),
                 &calendar, SLOT(season(int)));
QObject::connect(&widget3, SIGNAL(seasonChanged(int)),
                 &calendar, SLOT(season(int)));
QObject::connect(&widget4, SIGNAL(seasonChanged(int)),
                 &calendar, SLOT(season(int)));

calendar.startCalendar();
```

J'ai cependant un problème que je n'ai pas su résoudre ; seulement la fenêtre actuelle s'actualise, même si l'intégralité des widget reçoivent le signal.

Simuler les changements de saisons

J'ai rajouté un paramètre à la structure VertexData ; un QVector3D pour la couleur. Ainsi, lors de l'initialisation, en fonction du niveau de gris calculé à partir de la heightmap et de la saison, la couleur du vertex change.

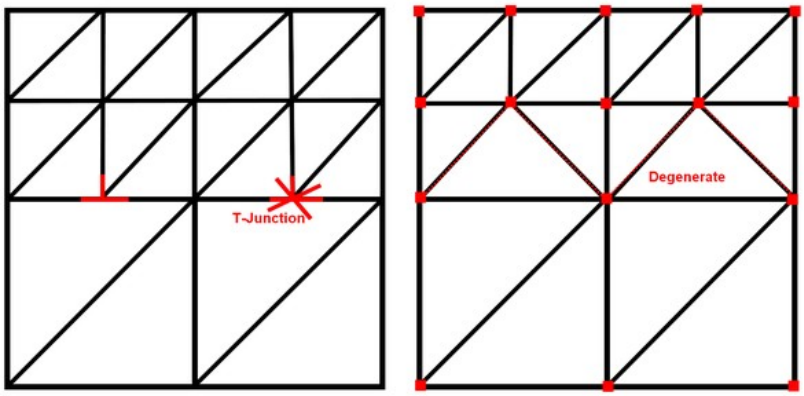
```
for(int j = 0; j < nbVertices; j++) {
    for(int i = 0; i < nbVertices; i++) {
        switch(season) {
        case 0: //summer : light on ground
            seasonColor = QVector3D(1.0, 1.0, grayLevel[compteur]);    ⚠ implicit conversion loses floating-point precision: '__gnu_cxx::
            break;
        case 1: //fall : orange and red on ground
            seasonColor = QVector3D(1.0 * ( 1 - grayLevel[compteur]), 0.5 * (1 - grayLevel[compteur]), grayLevel[compteur]);    ⚠ implici
            break;
        case 2: //winter : white on ground
            if(grayLevel[compteur] <= 0.5)                                    ⚠ implicit conversion changes signedness: 'int'
                seasonColor = QVector3D(1.0 * (1 - grayLevel[compteur]), 1.0* (1 - grayLevel[compteur]), 1.0* (1 - grayLevel[compteur]));
            else
                seasonColor = QVector3D(1.0 * grayLevel[compteur], 1.0* grayLevel[compteur], 1.0* grayLevel[compteur]);    ⚠ implicit con
            break;
        case 3: //spring : light and green on ground
            seasonColor = QVector3D(grayLevel[compteur], 1.0, grayLevel[compteur]);    ⚠ implicit conversion loses floating-point precisi
            break;
        default:
            seasonColor = QVector3D(1.0, 1.0, 1.0);
            break;
        }

        vertices[i+j*nbVertices] = {QVector3D(i*incr, j*incr, grayLevel[compteur++]),
                                    QVector2D(i*(1./(nbVertices-1)), j*(1./(nbVertices-1))),
                                    QVector3D(seasonColor[0], seasonColor[1], seasonColor[2])};
```

grayLevel[compteur] contient le niveau de gris du vertex courant.

Intégrer un QuadTree

Je me suis renseigné sur la méthode à implémenter. Je n'ai pas réussi à implémenter les problèmes de T-jonction.



Gestionnaire de scène

Je ne l'ai pas implémenté dans ce TP, mais nous voulons en implémenter un dans notre projet. En effet, comme il s'agit d'un jeu de stratégie en 2D, nous aimerions afficher uniquement ce que l'utilisateur voit sur son écran. Pour se faire, chaque élément du jeu sera organisé dans un arbre. Ainsi, il faudra vérifier les coordonnées x et y (il s'agit d'un jeu en 2D) de chaque élément (les bateaux) et afficher uniquement ceux présent dans l'affichage.

Bonus

Pour la pluie et la neige, il faudrait générer une seule particule et l'instancier autant que l'on souhaite. Pour la rendre « physique », à chaque tick il faut que l'objet soit attirer vers le sol, donc décrémenter sa coordonnée z. Pour gérer la collision avec le plan, il faudrait vérifier la hauteur du vertex en dessous de la particule, et déplacer la particule sur le vertex voisin ayant la valeur Z la plus faible. Ainsi, la particule descendrait petit à petit jusqu'à z=0.

Pour le niveau de détail des .obj, on pourrait augmenter le nombre de triangle sur les faces visible par l'utilisateur. Ainsi, on aurait moins de triangle à calculer sur les faces que l'utilisateur ne peut pas voir.

Pour garder un rendu temps réel, il faudrait mettre un cap aux fps. On pourrait utiliser la méthode sleep() pour garantir le temps entre chaque frame.

Source du projet : https://github.com/cpttrvs/hmin317-tp1