Programmation graphique 3D
M1 Informatique I3D - S1 Automne 2020–2021
Travaux Pratiques

TP 1 : Introduction à la programmation sur GPU

Introduction

Dans ce premier TP, nous allons voir les fonctionnalités et techniques de base à la programmation graphique sur GPU. Pour cela, nous aborderons :

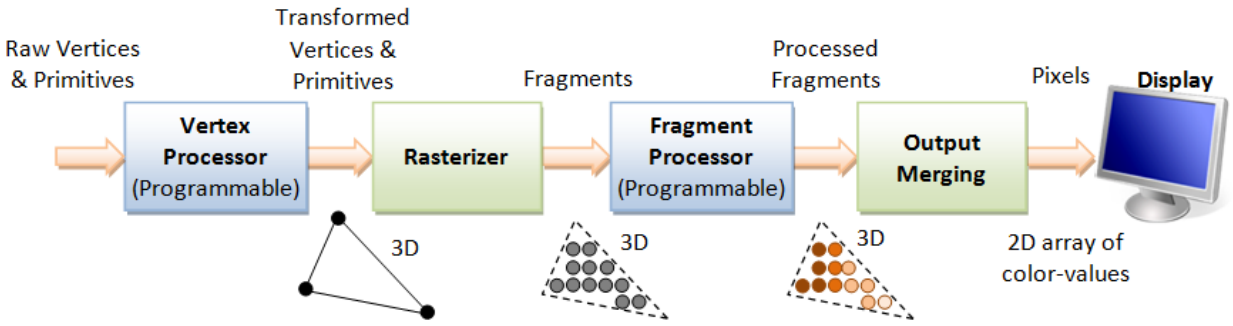
[1] Les données : **VBO** (vertex buffer object), **VAO** (vertex array object)

- création d'une géométrie
- affichage : points, lignes => **glDrawArrays()**

[2] Les shaders programmables :

- **vertex shader** : appelé pour chaque vertex en parallèle => écrit une position (après transformations ou non) dans la variable prédéfinie **gl_Position**
- **fragment shader** : appelé pour chaque fragment (pixel rastérizé) en parallèle => écrit la couleur de chaque fragment dans le buffer de couleurs associé à l'écran
- différents types de variables : **in / out** (traversée du **pipeline graphique**, communication entre shaders), **uniform** (pour les constantes)
- exemples : modification de l'apparence (couleur), animation (temps), génération procédurale (**gl_VertexID**, **gl_InstanceID**)

Rappel du pipeline graphique :



Exercice 1 : Premier programme OpenGL (ou plutôt WebGL)

A partir du fichier *base.js*, prenez connaissance de l'ensemble des fonctionnalités de bases qui y sont implémentées.

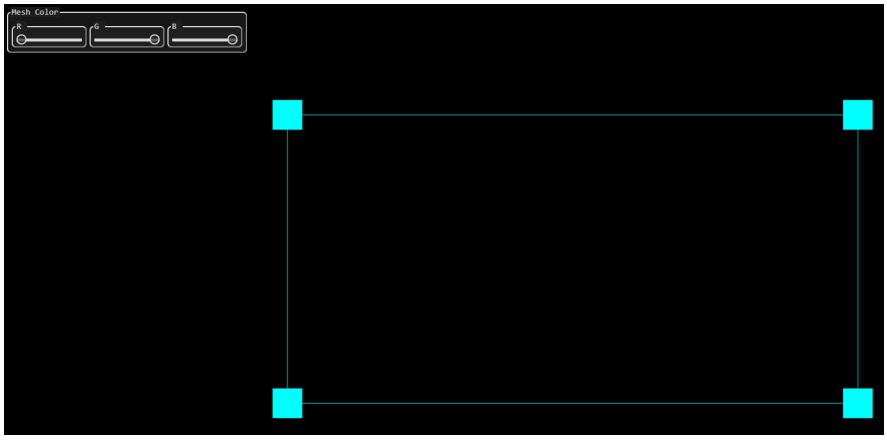
- Initialisation : création de la géométrie, création d'un VBO puis d'un VAO.
- Affichage : fonction qui determine comment on fait le rendu de la scène.
- Shaders : programmation du pipeline GPU.

Prenez le temps de lire ce code en détail !

Exercice 2 : Un peu de fléxibilité

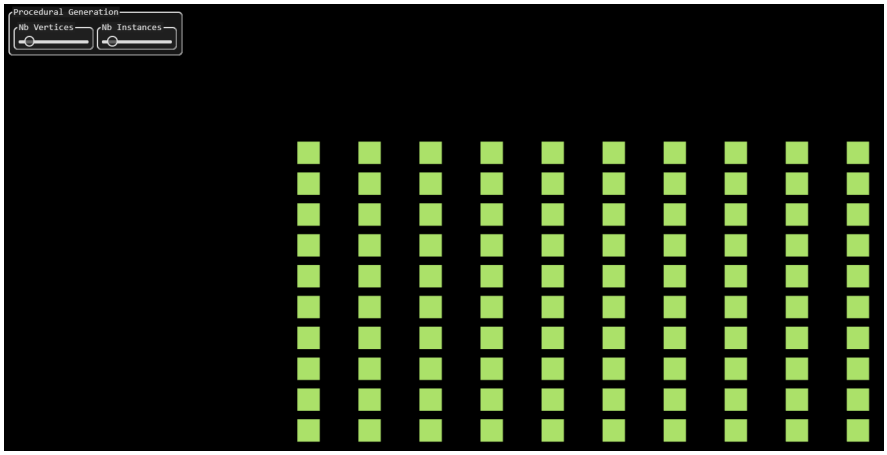
A partir du fichier *base.js*, que vous maitrisez maintenant sur le bout des doigts, vous allez faire les modifications suivantes :

1. on voudrais pouvoir définir une couleur depuis le CPU et l'envoyer vers le GPU (variable uniform);
2. maintenant, à travers une interface graphique (UserInterface de l'API easywgl);
3. puis ajouter une animation sur la taille des points en fonction du temps. Dans le shader, utilisez la fonction `mix(min, max, value)`. Elle retourne une interpolation linéaire de la valeur (comprise entre 0 et 1) entre `min` et `max`. Si `value` vaut 0 alors la fonction retourne `min`, si `value` vaut 1 alors la fonction retourne `max`.



Exercice 3 : Sans géométrie...

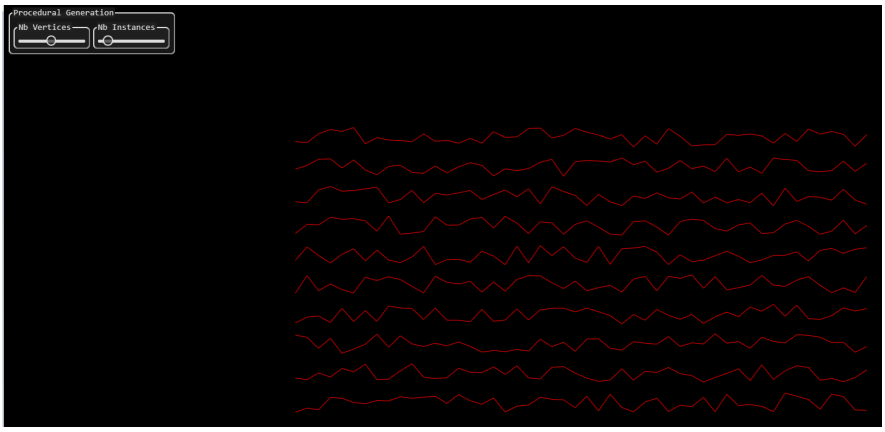
Vous allez maintenant faire de la génération procédurale. Sans définir de géométrie, (on dit aussi, de manière implicite), vous afficherez des points alignés sur une grille 2D. Pour cela, on utilisera la méthode d'affichage par instancing -> `drawArraysInstanced()`. On indiquera le nombre de sommets et le nombre d'instances en pouvant faire varier ces nombres par une interface graphique. Vous pouvez également faire varier la couleur des sommets en fonction du temps.



Vous pouvez aussi ajouter de l'animation en faisant varier la position des sommets à l'aide d'une fonction de bruit directement dans le vertex shader :

```
float noise( vec2 st )
{
    return fract( sin( dot( st.xy, vec2( 12.9898, 78.233 ) ) ) * 43758.5453123 );
}
```

Puis en passant en mode de dessin par ligne vous devriez obtenir quelque chose comme ca :



Exercice 4 : A travers le pipeline

On veut maintenant pouvoir, dans le fragment shader, attribuer une couleur en fonction de la position 2D (x,y) des sommets. Vous utiliserez les variables `in/out` pour communiquer entre les shaders.