GPGPU
CPE

5ETI IMI

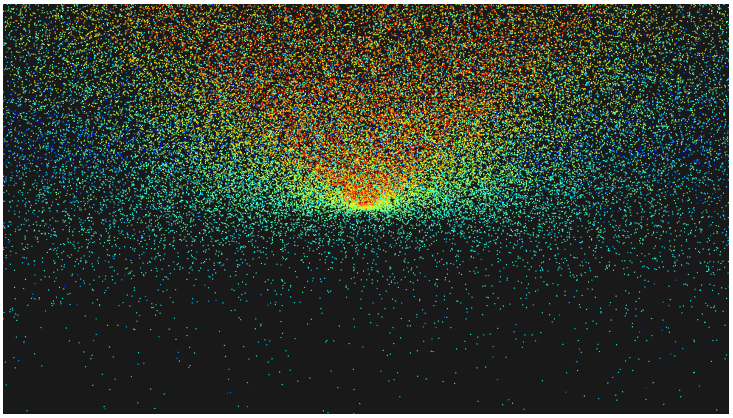

Figure 1: 100’000 particules volantes

1 But

L’objectif de ce TP est de mettre en avant deux grandes techniques utilisées pour faire de la GPGPU :

1. les Transform Feedbacks (TF),
2. les Computes Shaders (CS)

Ce TP sera noté. Il est composé de deux programmes relativement similaires qu’il faudra completer.

2 Prise en main de l’environnement

2.1 Compilation

Question 1 *Compilez les codes, assurez-vous pour chacun de voir un écran noir avec un groupe de pixel allumé au centre.*

Question 2 *Observez les fichiers du projet (hors dossier external) afin de comprendre le fonctionnement des projets. Vous devez comprendre l’ensemble du code.*

3 Transform Feedbacks

Les transform Feedbacks permettent de modifier les vertices contenus dans un buffer. Seul le vertex shader (et dans certains cas le geometry shader) sont utilisés. Les transform feedbacks ne servent pas à faire de l'affichage. Il est cependant possible d'utiliser les buffers modifiés pour de l'affichage. Les vertices ne sont pas obligatoirement représentatifs d'une géométrie et les TF sont un moyen de faire de GPGPU.

Pour effectuer un TF:

3.1 À l'initialisation

- Créer les shaders
- Créer un programme et lier les shaders
- ▲ IL FAUT PRÉCISER LES ÉLÉMENTS DE SORTIE AVANT DE LINKER LE PROGRAMME.

  `glTransformFeedbackVaryings(...)`
- Linker le programme
- Créer et utiliser les VAO, VBO et EBO normalement (voir `mesh.cpp` pour plus d'information). Il faut créer un/des VBO pour le stocker l'information issue du TF.

3.2 À l'affichage

- Désactiver l'affichage : `glEnable(GL_RASTERIZER_DISCARD)`
- Préciser le programme de TF
- Préciser le VAO
- Préciser le VBO
- Préciser le buffer d'écriture du TF : `glBindBufferBase(...)`
- Faire la transformation :

  ```
  glBeginTransformFeedback(...);
  glDrawArrays(...);
  glEndTransformFeedback();
  ```
- Attendre le buffer `glFlush(...)`
- Réactiver l'affichage `glDisable(GL_RASTERIZER_DISCARD)`

Attention, il faut décrire de nouveau le fonctionnement des buffers après utilisation des TF ! De plus, les attributs pouvant être différents entre la visualisation et le TF, il est bien de préciser le fonctionnement des VBO avant chaque utilisation de ceux-ci.

Pour lire le contenu d'un buffer, vous pour utiliser : `glGetBufferSubData(...)`. Dans le cas où un geometry shader est utilisé, le nombre de vertices peut changer. Il est alors nécéssaire d'utiliser un *query objects*. Plus d'information sont disponible : https://open.gl/feedback.

Dans un premier temps, commencez avec seulement les positions.

Question 3 *Créez un vertex shader (appelé ici* `tf.vs`*) qui déplace les points de manière uniforme (de l'ordre de 0.01).*

Question 4 *Ajoutez dans le code principal, la création d'un programme qui s'occupera de changer la position des particules grâce au shader* `tf.vs`*. Pensez à préciser les variables à capturer.*

Question 5 *Avant la demande d'affichage des particules, effecuter la modification de la position des points. Puis échanger les buffers d'entrée et de sortie (seulement leurs identifiants).*

4 Compute Shaders

Les compute shaders ne font pas partie du pipeline de rendu. Ils permettent de dispatcher des calculs sur la carte graphique en contrôlant les unités de calcul du GPU.

Pour les compute shaders, il n'y a donc pas de notion d'attribut de vertex. On peut en revanche utiliser des buffers accessible dans les compute shaders, les buffers les plus utilisées sont les Shader Storage Buffer Object (SSBO) et les textures (on utilisera les SSBO pour ce TP). Pour connaitre les données à traiter dans le compute shader on peut utiliser l'identifiant du travail (`gl_GlobalInvocationID`). Attention, il faut autant de travaux que d'objet : `gl_GlobalInvocationI` donc `local_size * gl_WorkGroupSize == N` avec N le nombre de particules.

Les SSBO sont intrinsequement des buffers tout comme les VBO et il est ainsi possible d'utiliser le même buffer pour le VBO et pour le SSBO. Il faut juste au moment de la section du buffer choisir son rôle (`glBindBuffer()`). Il est possible de lire et écrire dans le même SSBO avec un CS.

Pour utiliser un CS avec un SSBO:

4.1 À l'initialisation

- Créer un compute shader
- Créer un programme et lier le shader
- Linker le programme
- Créer un SSBO (dans le cas du TP, on utilisera les VBO utilisés pour l'affichage).

4.2 À l'affichage

- Préciser le programme de CS
- Préciser les buffer à utilsier : `glBindBufferBase(...)`
- Faire la transformation : `glDispatchCompute(...);`
- Assurer l'écriture de l'ensemble des données : `glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_B`

4.3 Compute shader en GLSL

- Préciser le nombre de groupe de travail local : `layout(local_size_x = ...) in;`
- Préciser les buffers en entrée :

```
layout(std430, binding = 1) buffer positionBlock
{
  float pos[];
};
```

On accède aux éléments avec `pos[i]` en lecture et écriture. Attention, ne pas utiliser de vec3 mais des float (à cause du packing, gestion interne de stockage de données). Il faudra donc $3N$ travaux.

On peut récuperer le contenu d'un buffer avec :

```
std::vector<float> buff(3*NB_PARTICULES); // n is the size
glGetNamedBufferSubData( ... , 0, 3* NB_PARTICULES * sizeof(float), buff.data(
```

Question 6 *Avant la demande d'affichage des particules, effecuter la modification de la position des points. Pensez à repréciser au VAO comment lire les données.*

5 Chute libre et visualisation

Question 7 *Prenez en compte la vitesse et ajoutez la gravité:*

$$p = p + v * dt; \quad v = v + m * g * dt;$$

On peut dans un premier temps, ne pas prendre des valeurs physiques.

Question 8 *Faites revenir les particules à leur position d'origine si elles sont en dessous d'une certaine valeur en y et remettez des valeurs aléatoires de vitesses. L'aléatoire en GLSL est difficile à obtenir, vous pouvez regarder : http://www.science-and-fiction.org/rendering/noise.html.*

Question 9 *Modifiez le fragment shader pour avoir une couleur qui dépend de la vitesse. Un lien vers des colormaps pour GLSL est dans le fragment shader.*