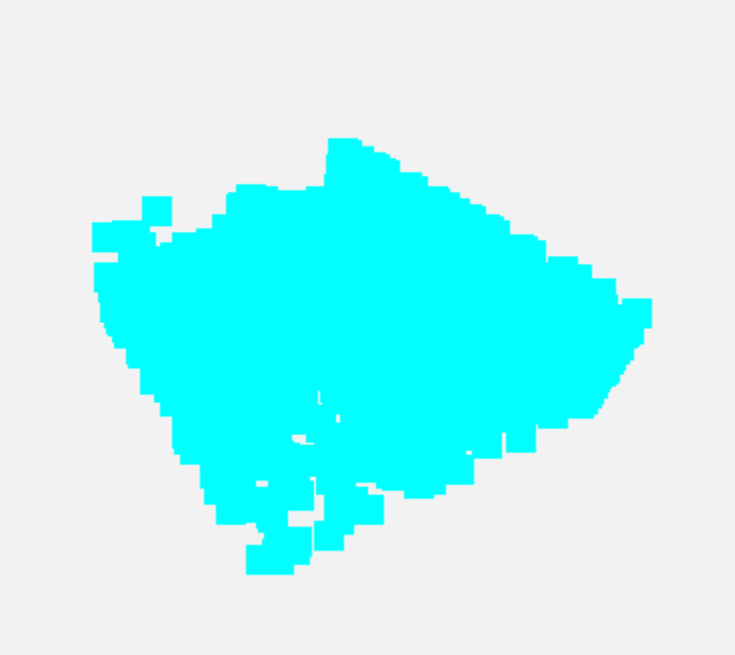
Simulation de fluide

PREMIER PROJET OPENGL

Amaury Lekens | Synthèse d'image | Q2 2018



Travail réalisé

Le travail demandé consistait en la réalisation d'une simulation de fluide la plus réaliste possible en utilisant openGL et la méthode PIC FLIP.

La première partie du travail qui a été réalisé est un travail sur la caméra. L'application permet de se déplacer dans tous les sens à l'intérieur de la scène et de changer l'angle de vision de la caméra.

La deuxième partie du travail est une première implémentation de simulation de fluide sans utiliser la méthode PIC-FLIP. Cette simulation génère simplement 10 000 particules et les projette dans un cube prédéfini avec une vitesse et une position aléatoire, lorsqu'une particule cogne une paroi, elle rebondit sur celle-ci en perdant une partie de sa vitesse. En plus de cela, on applique une force externe sur l'ensemble des particules, la gravité. (Projet 3D-Engine-Base-WPF)

La troisième et dernière partie consiste en une tentative d'implémentation de la méthode PIC FLIP. Malheureusement, elle n'est pas fonctionnelle. Le rapport se contentera pour cette partie de montrer les différents essais d'implémentation des shaders correspondants aux différentes étapes de la méthode PIC FLIP. (Projet 3D-Engine-Base-PF)

1. Caméra

Toute l'implémentation des fonctions liées à la caméra se trouve dans les classes :

- Input : cette classe gère les événements claviers et souris
- Camera : cette classe gère le déplacement et l'angle de vision de la caméra

Il y a deux déplacements de la caméra :

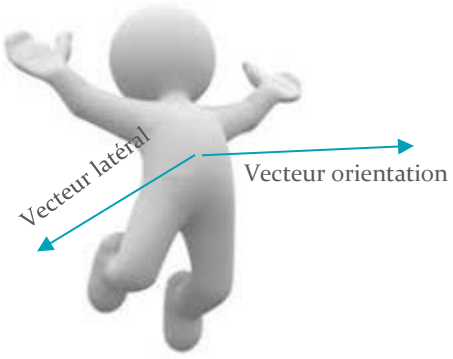
- Le mouvement en tant que tel dans l'espace 3D, selon les axes x, y, z. Ce mouvement est effectué avec les flèches du clavier.
- Le mouvement d'orientation de la caméra. Ce mouvement se fait grâce à la souris.

Pour pouvoir réaliser ces déplacements, l'objet doit retenir des données :

- La position : la position actuelle de la caméra → vecteur de taille 3
- Le point cible : le point que la caméra observe → vecteur de taille 3
- Le vecteur orientation : l'orientation de la caméra en termes de vecteur → vecteur de taille 3
- Le vecteur latéral : le vecteur orthogonal au vecteur orientation
- L'angle phi : l'angle vertical d'orientation de la caméra
- L'angle thêta : l'angle horizontal d'orientation de la caméra

1.1. DÉPLACEMENT DANS L'ESPACE 3D

Comme le clavier comporte 4 flèche, on peut se déplacer de 4 façons : en avant, en arrière, vers la gauche vers la droite.



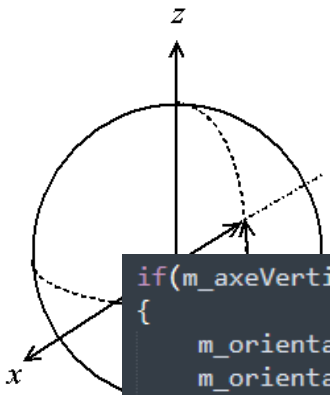
Quand on se déplace vers l'avant ou l'arrière le déplacement se fait selon la direction indiquée par le vecteur orientation. Quand on se déplace vers la droite ou la gauche le déplacement se fait selon la direction indiquée par le vecteur latéral.

Pour effectuer ces déplacements, en fonction des inputs du clavier, on met à jour la position et le point cible grâce aux deux vecteurs.

```
if(input.getKey(SDL_SCANCODE_UP))
{
    m_position = m_position + m_orientation * 0.5f;
    m_targetPoint = m_position + m_orientation;
}
```

1.2. ORIENTATION DE LA CAMÉRA

Pour orienter la caméra, on va modifier le contenu du vecteur orientation grâce aux coordonnées polaires. En fonction des mouvements de la souris, on va mettre à jour les angles ϕ et θ (! ils ne doivent jamais dépasser 89°, sinon il y aura des problèmes pour passer en coordonnées cartésiennes).



Ensuite pour calculer le nouveau vecteur orientation, en fonction de l'axe vertical qu'on a choisi, on fait la transformation vers les coordonnées cartésiennes.

```
if(m_axeVertical[0] == 1.0)
{
    m_orientation[0] = sin(phiRadian);
    m_orientation[1] = cos(phiRadian) * cos(thetaRadian);
    m_orientation[2] = cos(phiRadian) * sin(thetaRadian);
}
```

Ensuite comme le vecteur orientation a changé, le vecteur latéral doit faire de même. Il faut simplement calculer le vecteur orthogonal au vecteur orientation et normaliser celui-ci.

```
void Camera::lookAt(mat4 &modelview)
{
    modelview = lookat(m_position, m_pointCible, m_axeVertical);
}
```

Pour terminer, on met à jour le point cible.

```
m_deplacementLateral = cross(m_axeVertical, m_orientation);
m_deplacementLateral = normalize(m_deplacementLateral);

m_pointCible = m_position + m_orientation;
```

1.3. METTRE LA MATRICE MODELVIEW À JOUR

Pour que le mouvement de la caméra soit effectif, il faut mettre à jour la matrice modelview :

2. Implémentation basique d'une simulation de fluide

2.1 PASSAGE DES PARTICULES AUX SHADERS

Pour passer l'ensemble des particules à un shader, il faut d'abord les mettre dans un buffer openGL :

```
particules = new Buffer(pp, sizeof(pp));
```

Ensuite, on attache le buffer au shader :

```
computer_gravity->setData(1, particules);
```

2.1. GÉNÉRATION DES PARTICULES

On génère 10000 particules avec des vitesses v_x, v_y, v_z et des positions p_x, p_y, p_z aléatoires (mais compris dans des intervalles) :

```
for(int i(0); i < 10000; i++)
{
    float x = (rand() % 20) - 10;
    float y = (rand() % 15);
    float z = (rand() % 20) - 10;

    float vx = (rand() % 20) - 10;
    float vy = (rand() % 20) - 10;
    float vz = (rand() % 20) - 10;

    pp[i].position = vec4(x, y, z, 1.0f);
    pp[i].velocity = vec4(vx, vy, vz, 1.0);
}
```

2.2. APPLICATION DE LA GRAVITÉ

L'application de la gravité se fait via un compute shader (gravity.comp). Il va modifier la vitesse selon l'axe vertical (y) de toutes les particules (sauf celles qui se trouvent sur le sol virtuel).

```
if(p.y > 0){
    v += (1.0/60.0) * vec3(0, -9.81, 0);
}
```

2.3. VÉRIFICATION DES FRONTIÈRES

Pour être sur que les particules restent dans un volume prédéfini, on utilise aussi un compute shader (boundary.comp). Dans celui-ci, on vérifie la position de toutes les particules et si une de celles-ci se trouve au-delà du volume, on applique deux transformations :

- La particule est déplacée sur la frontière (définie par le volume) qu'elle a dépassée
- La vitesse selon l'axe, avec lequel la particule a effectué un déplacement est multiplié par -1 et par un facteur compris entre 0 et 1 (au plus il est élevé, au plus le fluide se stabilisera vite)

```
if(p.y > 20) {
    p.y = 20;
    v.y = 0.95*(-v.y);
}
```

3. Tentative d'implémentation du PIC-FLIP

3.1. CRÉATION ET VIDAGE DE LA GRILLE

La méthode PIC-FLIP est basée sur une grille qui discrétise le volume. On va donc créer une structure de donnée (Voxel) qui représente un élément de cette grille. Cette structure contient une vitesse, une vitesse décalée et une nouvelle vitesse décalée (chaque fois selon les trois axes). On ne retient pas de position car un voxel se trouve déjà à une certaine position de la grille. Ensuite on crée un buffer qui contiendra cette structure autant de fois qu'il y a de voxel dans la grille (ici 27000). Il faut ensuite associer le buffer au shader.

Quand on lance l'exécution du shader, on définit un certain nombre de workgroups selon les 3 axes (ici on définit 30 workgroups selon l'axe vertical).

```
computer_empty->compute(1, 30, 1);
```

A l'intérieur du shader, on définit la taille des workgroups. Dans ce projet, la taille du workgroup est de 30 selon l'axe x et 30 selon l'axe z. Un workgroup contient donc 900 éléments. Comme il y a 30 workgroups, la shader va bien travailler sur 27000 éléments.

```
layout (local_size_x = 30,
        local_size_z = 30) in;
```

Le shader « emptyGrid.comp » va simplement mettre les différentes vitesses de chaque élément de la grille à zéro.

```
uint index = (gl_GlobalInvocationID.y*30*30) + (gl_GlobalInvocationID.z*30) + gl_GlobalInvocationID.x;
voxels[index].velocity = vec4(0.0);
voxels[index].staggeredVelocity = vec4(0.0);
voxels[index].newStaggeredVelocity = vec4(0.0);
```

Le shader « grid.comp » va placer les particules dans la grille.

3.2. STAGGERED GRID

Le shader staggered grid va calculer les vitesses décalées qui sont simplement des interpolations de vitesses de deux particules voisines. Pour trouver les particules voisines

```
uint index = gl_GlobalInvocationID.x;
vec3 v = particles[index].velocity.xyz;
vec3 p = particles[index].position.xyz;

uint voxel_index = uint((30.0*30.0*p.y) + (30.0*p.z) + p.x);
voxels[voxel_index].velocity = vec4(v, 1.0);
```

on utilise les fonctions openGL :

- gl_LocalInvocationIndex : index de l'élément de travail à l'intérieur du workgroup
- gl_WorkGroupID : index du workgroup en cours

3.3. GRAVITÉ ET FRONTIÈRE

Ces deux shaders font la même chose qu'au point 2 mais ils agissent sur la grille.

3.4. MISE A JOUR DES VITESSES ET POSITIONS

Le shader « updateVelocity.comp » met à jour les vitesses en faisant les interpolations « inverse » du point 3.2. La méthode PIC interpole directement les nouvelles vitesses tandis que la méthode FLIP interpole les différences de vitesses. La nouvelle vitesse est calculée avec la formule suivante :

$$v = \alpha.\mathrm{PIC} + (1\text{-}\alpha).\mathrm{FLIP}$$

Le shader « position.comp » met à jour la position des particules (et les vitesses) dans le buffer « particules ».