***MORE - Squelette de projet***

Nom du projet: Rope Man

Nom des étudiants: Margot Pellegrin | Louan Ourvouai | Nathan Vetter

Ce document a pour but de clarifier la structure, les étapes à implémenter pour le projet, histoire de ne pas partir sur une mauvaise route. Je vous demande de vous concerter et remplir ce document (1 par groupe), ce qui décrira comment vous comprenez votre projet et nous permettra de discuter précisément !  
 Ne craignez pas de préciser si vous n’êtes pas sûr, ce n’est pas un document d’évaluation. Je donne ce template remplis avec des exemples des TPs/activité que nous avons vu ensemble.

***Description des classes utiles pour le projet (cela exclu les classes « OpenGL » de la base de code) :***

- Tous les TPs : Classe « freefly camera », classe « projection matrix »

- TP2 : Classe « brain tétraèdre » pour lire le fichier et loader les buffers GPU

Pour mon projet RopeMan:

- Classe player:

o Attributs : position courante, matrice de camera, taille max des cordes, position bout de la corde 1, position bout de la corde 2

o Méthodes : init, update\_position\_from\_rope\_and\_time\_frame, test\_collision, lancer\_de\_corde, game\_over

- Classe plateforme:

o Attributs : position courante, longueur, hauteur, largeur

o Méthodes : init

- Classe lava:

o Attributs : position courante

o Méthodes : init, update\_position\_from\_time

Je vois l’idée et ça pourrait marcher comme ça, mais je vous donne ma proposition :

* Classe Game : pour gérer les ressource et ce qu’il se passe
* Classe Rope : pour gérer la physique d’une corde (qui est une ligne de masse-ressorts), contient aussi les données pour les buffers (VBOs) pour le rendu des cordes
* Classe Player : events clavier, physique du joueur, matrices de camera, pointeurs vers cordes « actives »
* Classe Plateform : Gère les collisions & paramètres des plateformes

Note :

- La simulation des cordes / du joueur est sous CPU au lieux de compute shader, c’est plus simple et vu le faible nombre de paramètres, rentable !

- les plateformes ne sont pas gérés individuellement, mais toutes ensemble, procéduralement (je vous explique ça lundi après-midi). Le seul point négatif de cette représentation est qu’il faudra « copier-coller » le code GLSL de la map (dans le raymarcheur de platform), sur CPU dans la classe platform pour query les collisions

- La gestion des cordes utilise une « banque », pour éviter que des cordes disparaissent : Le fait que le joueur décide de lâcher une corde ne doit pas la faire disparaitre (pour gérer la nouvelle corde lancée), pour cela, il faut un buffer de cordes, d’où la nécessité d’une classe Rope

***Description des ressources mémoire pour le projet, côté CPU et GPU :***

Pour mon projet RopeMan :

- UBO pour les matrices + paramètres (GPU)

- Structure CPU pour l’UBO

- SSBO (x3) pour les positions player + cordes (GPU) OK, ça marcherait. la position du player va dans l’UBO des paramètres, c’est plus simple. Et pour les cordes, puisque la simulation est sur CPU, pas besoin de SSBO, mais il faut un VBO pour pouvoir la rendre

- Instances de classe « player» et « plateforme»

- Texture2D pour les contours des plateformes (GPU) C’est-à-dire ? Pour avoir des plateforme texturée ? On pourra faire ça (avec une technique appellée triplanar mapping) mais c’est pas prioritaire

***Description en pseudo-code du projet. Faire figurer les shaders nécessaires et utilisé:***

Initialisation :

- Chargement/compilation shaders

- Allocation des 3 SSBOs

- Allocation UBO pour les paramètres & matrices

- Initialisation d’une instances de classe « player »

- Initialisation de plusieurs instances de classe « plateforme »

- Initialisation d’une instances de classe « Lava »

- « Clear » des 3 SSBO et écriture position initiale

- Remplissage CPU et écriture GPU de l’UBO

- Allocation et remplissage GPU du VBO de la map La map/plateformes ne passe pas par un VBO mais une surface implicite (ray marching)

En regardant la base de code, vous verrez les initialisations des classes et ressources

Boucle de jeux :

- Détection des cliques de la souris => actualisation des positions courantes et de la camera du joueur

- Actualisation et écriture GPU de l’UBO avec les nouvelles données (des matrices ont changé)

- Écriture GPU des nouvelles positions dans les 3 SSBOs

- DrawCall pour les plateformes [Shader plateforme: Tessellation – Geometry- Fragment shader]

o Avec un VAO vide, toutes les données seront lues depuis les SSBO

o Le drawcall est paramétré pour remplir les 2 moitié d’écran (chaque joueur)

- DrawCall pour la Lava [Shader lava: Vertex – Fragment shader]

o Avec un VAO qui utilise le VBO de la map

- DrawCall pour les Cordes [Shader corde: Vertex – Fragment shader]

o Avec un VAO qui utilise le VBO de la map

- Test des collisions

o Si pas de collision, la corde reviens vers le joueur

o Si collision entre une plateforme et une corde, le joueur est projeté vers le point d’impact

o Si collision entre le joueur et la lave, le joueur perd la partie

(Note, dans cette implémentation, le test de collision est fait sur CPU. On aurait pu le faire sur GPU avec un compute shader). C’est très bien sur CPU !

C’est légèrement mieux de mettre la physique et le collision avant le rendu (dans une boucle synchrone comme celle-ci). Je suis d’accord dans l’idée même si vu les classes, ça s’écrira un peu différement

***Questions/point importants/de difficulté à voir ensemble :***

- Voir les méthodes de DrawCall pour les différents éléments.

J’ai déjà fait les méthodes pour les drawcall, vous pourrez regarder… mais le gros du boulet est dans les shaders associés aux drawcall