Projet - INF443

Dario Shariatian

1er juin 2021

Dossier

Introduction

La scène animée en elle-même est constitué d'un terrain déformé procéduralement à l'aide d'un bruit de perlin, de plusieurs **tornades** se baladant aléatoirement ou selon une trajectoire prédéfinie, **d'arbres** placées aléatoirement sur le terrain et munie d'une hitbox hiérarchique, une **arme** dont la canon génère de la **fumée** et lançant des **balles** soumises à la gravité et pouvant rebondir sur la scène notamment à l'aide des hitbox et des structures de collision sous-jacentesun, **oiseau** muni d'une hitbox se déplaçant selon une trajectoire prédéfinie qui pers ses couleurs lorsque touché par une balle de l'arme. Il se trouve aussi une structure de **sphère fractale** afin de jouer avec la structure de collision. Une **skybox** entoure la scène. On peut librement switcher entre la vue première personne et la vue globale. Il est possible de changer quelques paramètres d'exécution de la scène à l'aide du GUI, dont j'ai pu fournir une classe `GuiWindow` permettant d'en faciliter l'intégration. Enfin, il faut appuyer sur `ECHAP` afin que le mouvement de la souris sois capturé par l'écran et pour pouvoir lancer les balles. Nous allons progressivement décrire les structures mises en place pour générer ces éléments.

Je donne la description de quelques classe de base réutilisées dans tout le code. On fait principalement usage de la généricité en évitant de définir des classes abstraites dans un soucis de performance, en particulier dans la structure de collision où j'ai voulu éviter les fonctions virtuelles.

0.1 Classe `DefaultMeshDrawable`

Cette classe permet l'affichage de maillage basique comme utilisé dans le TD, avec le shader par défaut étant celui de phong. Comme on a voulu garder des appels de type `draw(drawable, scene)`, on définit cette fonction avec un mécanisme de template, en utilisant les méthodes d'une classe SCENE que je présenterai juste après. Cette fonction est généralement redéfinie pour chaque spécialisation de cette classe.

0.2 Classe `DescriptiveAnimation`

Permet de suivre l'évolution périodique d'un point passant par des positions spatio-temporelles clés. On peut utiliser une interpolation linéaire ou une interpolation par spline cardinale avec coefficient $k = 0.5$ (Catmull-Rom). Ses spécialisations doivent fournir un callback éventuellement égale à la fonction nulle pour gérer l'animation de leur structure.

0.3 Classe `CollisionStructure`

Utilise plusieurs primitives de collision (boule, plan, cuboïde) pour construire une structure hiérarchique. Permet ensuite de vérifier s'il y a collision entre deux structures. Cette classe hérite de `DefaultMeshDrawable`, et peut être affichée si on lui fournit un mesh (en particulier utile pour représenter les hitbox).

0.4 Classe `ParticleSystem`

Permet de représenter un système de particule. Ses spécialisations pourront redéfinir la méthode `update(float dt)` pour faire évoluer les particules. On ne spécifie pas forcément des forces car par exemple la classe `Smoke` utilise une modélisation différente pour l'évolution de ses particules. Chaque particule est dôtée d'un temps de vie au-delà delaquelle elle n'est pas mise à jour.

1 Gestion Camera FPS

Voir fichiers `scene.*pp` Pour pouvoir switcher de manière simple entre les différentes cameras j'ai généralisé la définition de la scène en une classe mère `scene_environment` de manière à en faire hériter la scène en vue fps ou en vue globale. La caméra fps change de position et de rotation en prenant pour input le clavier et la souris. On utilise un callback sur le mouvement de souris, et on recalcule le `pitch` et le `yaw` de la caméra en prenant en compte la nouvelle position `p1` de la souris et son acienne position `p0`. La méthode suivante permet d'éviter tout effet de roll :

```
rotation r_pitch = rotation({ 1,0,0 }, head_scene.pitch);
rotation r_yaw = rotation({ 0,1,0 }, head_scene.yaw);
camera.orientation_camera *= inverse(r_pitch) * inverse(r_yaw);
head_scene.pitch += p1.y - p0.y;
head_scene.yaw -= p1.x - p0.x;
r_pitch = rotation({ 1,0,0 }, head_scene.pitch);
r_yaw = rotation({ 0,1,0 }, head_scene.yaw);
```

Pour la position de la caméra, on la place à une hauteur fixée au dessus du sol en utilisant une **interpolation bilinéaire** sur les sommets du terrain représenté sous forme de maillage (voir classe `Terrain`).

2 Animation

Duex objets suivent une animation descriptive : l'oiseau et une des tornades. Cette dernière décrit un mouvement de cercle avec une oscillation radiale. Les deux objets utilise une interpolation par spline cardinale. Les autres objets suivent

3 Collisions

Les collisions avec le terrain consistent à vérifier la hauteur du sol et la distance à ses côtés extremaux(voir fichier `terrain.hpp`)

Pour le reste, on utilise la classe `CollisionStructure`. Celle-ci permet de définir un graphe dont les noeuds sont des `CollisionPrimitive` et les enfants d'autre `CollisionStructure`. Un parcours de graphe permet de vérifier les collisions entre deux structures, selon le code suivant, qui conserve la hiérachie de parcours en effectuant les tests de collision niveau par niveau.

On fait en sorte de renvoyer un pointeur vers le CollisionPrimitive de la structure passée en appel.

On fait en sorte de mettre à jour les positions successives des structures de la même manière que pour la classe `vcl::hierarchy_mesh_drawable`, en passant de positions globales à locales. On aurait pu utiliser des matrices 4x4 pour représenter toutes les transofrmations linéaires mais par souci de temps je suis resté à des translations.

```
CollisionPrimitive* CollisionStructure::intersects(CollisionStructure* structure, int&
    max_sub_calls) {
  // If too much time, we consider there is no collision
  if (!max_sub_calls) return nullptr;

  // primitive global_pos is currently set right

  // First testing against parent structure given on input
  if (primitive->intersects(structure->primitive)) {
    // we continue on children if they exist
```

```
      if (structure->subStructures.size() == 0) {
         // we are done
         return primitive;
      }
      else {
         // Now launching the other's children on this, so we get ping pong effect
         // that makes a hierarchical parcours, as intended
         // We must be careful to update the children global_pos from here
         for (auto& substructre : structure->subStructures) {
            substructre->primitive->global_pos = structure->primitive->pos();
            CollisionPrimitive* prim = substructre->intersects(this, --max_sub_calls);
            if (prim) return substructre->primitive; // we return the other
         }
      }
   }
   return false;
}
```

Pour les calculs d'intersections de primitives (voir fichier `CollisionStructure.cpp`), nous avons collisions entre :

- Boule et boule : simple calcul entre les distances aux centres
- Boule et plan : comparaison de de position des deux points extremaux de la boule (séparé par le vecteur normal au plan) et la position du plan
- Boule et cuboïde : je n'ai pas trouvé de test simple et ai donc approximé la boule par un cuboïde
- Plan et plan : colinéarité des normales
- Plan et cuboïde : on tranforme le plan de manière à ce que le cuboïde devienne un cube. Si on appelle la matrice de changement de base M (qui est la matrice des vecteurs définissant le cuboïde) alors le vecteur normal se transforme via $M^{-1})^\dagger$ (ce qui préserve les propriétés de produit scalaire). Il n'y a plus qu'à comparer avec l'intersection avec un cube dans $[0, 1]$ ce qui se fait en inspectant l'intersection du plan avec les diagonales du cube.
- Cuboïde c et cuboïde c' : on place les diagonales de c' dans l'espace où c est un cube normalisé. On vérifie l'intersection de chacune de ces lignes (PQ) avec le cube en calulant la projection disons de P sur l'hyperplan confondu avec la face du cube le plus proche de P, et en vérifiant que cette projection est bien sur la face. Dans le code j'effectue ce calcul coordonnée par coordonnée.

Cette structure de collision hiérarchique permet de définir assez simplement des objets dont la construction s'effectue en création de sous-structures contenues dans le volume défini par leur parent. Voir la classe `FractalSphere` pour un exemple.

4 Fumée

Voir la classe `Smoke`. Permet la modélisation de fumée via l'équation de mouvement

$$v = \begin{pmatrix} -2 * y * \text{vorticity} * z \\ 2 * x * \text{vorticity} * z \\ |z| + \text{offset} \end{pmatrix}$$

La classe utilise la primitive openGL `GL_POINT` pour créer un quadrangle autour de la position d'une particule de fumée, et on vient plaquer une texture dessus. Ce quadrangle fait toujours face à la caméra, et requiert donc 4 fois moins de données à envoyer au gpu. On a redéfinit un vbo, et un vao (données entrelacées) pour uploader les données au gpu. Les fichiers `smoke_*.glsl` définissent les shaders utilisés pour ces données-là (les particules sont davantage transparentes et blanches au début, puis noir et opaque à la fin). J'ai aussi modélisé l'atténuation en taille des particules au fur et à mesure que l'on s'éloigne, avec l'équation $\frac{\text{taille\_default}}{\texttt{coef}[0] + \texttt{coef}[1] * d + \texttt{coef}[2] * d^2}$. Par défaut on a $\texttt{coef} = \{1, 0, 0.5\}$, mais c'est une variable uniforme susceptible d'être changée.

Selon les paramètres que l'on choisit cette classe modèlise la fumée sortant du canon de l'arme et les tornades sur la scène.

5 Balles

Voir classe `GravityBalls`. Ce sont simplement des balles avec un certain temps de vie qui suivent l'équation de la gravité et munit d'une structure de collision. Elles peuvent rebondir sur le terrain et sur les autres objets, avec une certaine perte en énergie à chaque rebond, spécifiée par `BALLS_REBOUNCE_LOSS`. Pour le rebond sur une surface elles utilisent l'équation $v := v - 2 * (v \cdot n)n$ où n est la normale de norme 1 du plan tangent à la surface au point de rebond.

6 Boule Fractale

Voir classe `FractalSphere`. Cette boule fractale utilise la structure de collision hiérarchique. On génère cette boule fractale en replaçant chaque boule par 6 boules plus petites translatés selon chaque direction de l'espace.

Enfin on retrouvera dans le code des classes et des fonctions pour le `skybox` (plaquage d'une texture sur un cube centré en le personnage, il a fallu bien gérer les coordonnées uv sur le patron), le `terrain`, les `tree`, le `Flamethrower` (qui ne jette pas encore de flammes, mais qui fume déjà) et un `object loader` qui permet de charger des fichiers `.obj` afin d'avoir des maillages importés d'autres logiciels (la petite bibliothèque utilisée pour cela à été trouvée en ligne).

Compilation

Exactement comme pour les TD, via cmake. Je laisse le projet `Visual Studio` que j'ai généré sur ma machine ainsi qu'un executable compilé en release sur ma machine sous Windows 10. Les options de compilation devraient permettre de le lancer sur une machine similaire.