Graphique 3D : For The King

Benoit Jehanno, Thomas Dias-Alves

7 mai 2013

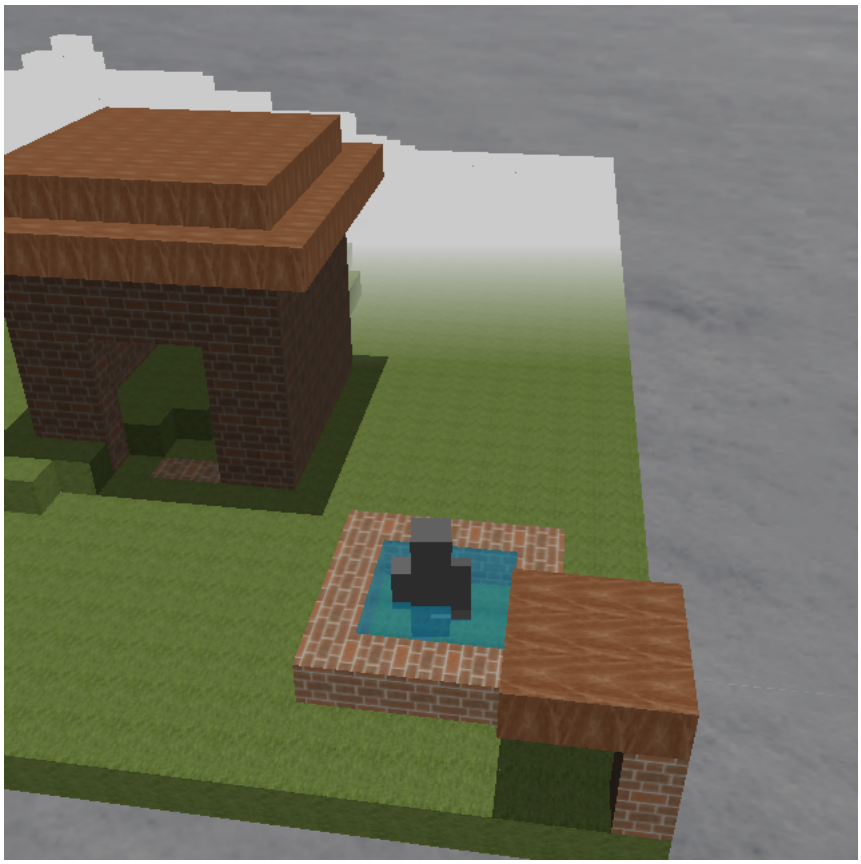

Figure 1 – Un projet de 3D les deux pieds dans l'eau

1 Introduction

Nous avons profité de l'opportunité d'un sujet assez libre en graphique 3D pour essayer de réaliser une base de jeu de platformes. C'est quelque chose que nous voulions essayer de réaliser avant le projet pour avoir une idée de la facon d'implémenter la chose. Nous nous sommes donc bien documenté sur le sujet avant de nous lancer dans l'implémentation.
Ce programme utilise les bibliothèques OpenGL et SDL (et boost pour la création simple de fichiers de configuration) qui sont toutes liées en externe (Le programme tourne sur Les EnsiPC), le nombre de classes est assez conséquent c'est pourquoi nous allons détailler les points relatifs au graphique 3D que nous avons implémenté en précisant où ils se situent dans le code.

Pour utiliser le programme et connaître les touches voir commandes.txt

2 Le decors

2.0.1 Génération de la carte

La génération de la carte est largement améliorable, c'est un des derniers points que nous avons abordé et le temps nous a manqué. Le principe de cette algorithme et de d'"empiler" aléatoirement des cercle de rayon aleatoire sur la carte de maniere a creer un relief assez homogène. De l'eau a ensuite été ajouté avant une certaine profondeur.

Voir le constructeur de Carte dans carte.cpp partie "Generation aleatoire de la carte"

La partie "constante" de la map vous sera utile pour tester les rendu dynamique d'ombre et de face des blocs.

2.0.2 Affichage optimisé des blocs

Les blocs qui constituent notre carte sont affichés intelligement selon leurs voisins. C'est à dire qu'aucune face inutile n'est créée sur le bloc (donc affichée). Cette méthode est appelée à la création de la carte mais également dynamiquement lors de la destruction d'un bloc.
On notera qu'on tient compte de la transparence des bloc ou non dans cette methode. L'affichage des blocs se fait d'ailleurs en deux temps, d'abord les blocs non transparents puis ceux transparents, pour éviter des effets de superposition.

Voir le constructeur de Carte dans carte.cpp partie "Mise en place des faces"

2.0.3 Ombres statics par lancé de rayon simplifié

Nous nous sommes renseigné sur des méthodes pour faire des ombres en OpenGL, notamment le shadow buffer et le lancé de rayon, mais ces méthodes nous semblaient très lourdes à implémenter pour notre structure de Carte, qui est une simple matrice 3D.Nous avons donc opté pour une méthode simplifiée de lancé de rayon qui s'accordait plus à notre structure.
Chaque face des blocs contient un booléen pour déterminer si celle-ci est éclairé ou non (*voir bloc.h et face.h*) . Grace à ceci on a pu simuler un éclairage vertical en "lancant des rayons" partant de tout les cubes en haut de la carte et s'arrétant au premier bloc opaque rencontré.
Cette méthode à l'avantage d'être assez légère elle peut donc être appelé dynamiquement lors

de la destruction d'un cube !
On pourra tester cet effet en essayant detruire le toit de la maison et en regardant le sol.

On notera qu'on tient compte de la transparence des blocs ou non dans cette methode.

Voir le constructeur de Carte dans carte.cpp partie "Mise ne place de la lumière"

2.0.4 Blocs texturés et transparence

Nous avons implémenté un affichage des blocs texturés (différentes textures selon leur type). Pour cela nous avons utilisé la fonction "loadTexture" de sdlglutils.h qui peut lire les textures avec MipMap pour éviter les effets de moiré.
L'affichage des blocs est visible dans le fichier util_fig.cpp dans la fichier dessinerCarre.
La texture d'eau a été ajouté avec de la transparence en utilisant GL_BLEND. On fait bien attention à dessiner les textures transparentes en dernier pour avoir vraiment cette impression que le personnage rentre dans l'eau.

3 Les objets

3.0.5 Structure hiérarchique de briques pour les mobiles

Mobile est la classe mère de tout les objets pouvants se déplacer sur la carte (Personnage, Projectile, Explosion ...) Nous avons fait le choix de les représenter grace à une structure hiérarchique de Briques. Mobile possède donc un champ arbre_brique et les objets Brique possèdent eux un vecteur de Brique* pour acceder aux suivants de l'arbre. Voir la classe mère des brique "Brique_gene.h" pour la structure hierarchique de Briques. Ceci requière d'avoir une méthode récursive pour l'affichage *(voir la fonction dessinerBriqueRec() de Brique_gene.cpp)* mais c'est un avantage énorme lors des animations puisque que chaque brique est représenté dans le repère de la brique précédente ! Cette structure est aussi utile pour la hitbox, il s'agit d'ailleurs du type de structure que nous avons vu en cours.

3.0.6 Animation du personnage

Un exemple de Mobile animé est notre Personnage. L'animation de la marche se fait dans la fonction marcher de Personnage.cpp. La structure hiérarchique précédente rend la création du personnage assez fastidieuse *(voir le constructeur de Personnage)* mais la manipulation de ce dernier est bien plus simple par la suite pour les animations.

3.0.7 Les particules

Une nouvelle brique a été créé pour afficher un effet de particule : Brique_particulaire. Elle s'intègre donc au modèle hiérarchique et permet de créer des Mobile avec des effets de particule constant *(Voir projectile.h et explosion.h)*.
Brique_Particulaire contient un tableau de particules ayant chaqu'une leur vitesse, leur position, leur couleur etc ... Lors de l'affichage de la particule leurs paramètres sont mis à jour en leur appliquant une gravité et une diminution de taille et de la couleur. Lorsque leur taille a atteint 0 elles sont réinitialisés avec une vitesse, une direction et une taille initiale aléatoire.

Voir Brique_particulaire.cpp

3.1 Affichage

3.1.1 Caméra 1ere, 3em personne et 3em personne inversée

Nous ne sommes pas partit de QGLviewer, nous avons du réimplémenter l'affichage, mais ceci nous à cependant permis de fixer la caméra que nous désirions. Ainsi trois types de vues centrées sur le personnage ont été implémentés : la caméra 1ere, 3eme personne et 3eme personne inversée.

Voir playercamera.cpp

3.1.2 Skybox

Pour ajouter un effet de réalisme nous avons créé une skybox autour de notre personnage selon le principe suivant : Nous nous mettons dans la vue du personnage grace à glLoadIdentity() et nous desactivons les éclairages et le Z buffer puis nous faisons une rotation pour compenser la rotation de la caméra et nous remettre "droit" par rapport à la carte. enfin nous affichons notre skybox. Ainsi la skybox est une boite relativement petite qui suit notre caméra mais s'affichera toujours en arrière plan.

Voir skybox.cpp, la skybox est appelé dans la fonction "look" de playercamera.cpp.

3.1.3 Autres effets graphique

D'autres effets graphiques qui ne mérite pas d'être détaillé ici ont été ajoutés tel que le brouillard pour les objets éloignés, la vision maximal et minimale.

4 Moteur Physique

4.0.4 Collision particule/Bloc

la première collision implémenté a été la collision bloc/particule. Cette collision assez simple est utilisé pour repérer le contact entre les boules d'énergie que lance le personnage et la carte. Ceci permet de détruire le bloc concerné tout en mettant à jour les faces des blocs avoisinant et leur éclairage.

4.0.5 Collision Bloc/Bloc

La collision Bloc/Bloc a été géré grace à un algorithme de collision de 2 box quelconques qui s'appuie sur le théorême de l'axe séparateur. Nous avons préférés ne pas nous restreindre à des boites alignées car dans notre cas nous avons juste ce cas à gérer. Pour cela on regarde si il existe un plan qui sépare nos box en testant des collisions de type sommet/face et arrête/arrête.

4.0.6 Moteur physique

Comme nous l'avons expliqué, une partie du moteur physique consiste à détecter les collisions. Nous avons distingué les collisions générales et celles avec le sol pour empécher le personnage de se diriger vers le bas (on projette alors son deplacement dans le plan horizontal) et gérer l'arrêt de la chute. La deuxième partie du moteur physique est la gestion de la gravité et de la viscosité du milieu. Pour cela nous avons utilisé les mêmes schémas d'intégrations qu'en TP.

5 Conclusion

En conclusion, ce projet peut continuer d'évoluer, en tout cas c'est ainsi que nous avons essyé de programmé tout du long du projet. Les autres engines du jeu sont déjà présents, il reste à les compléter.
De nombreuses améliorations sont possibles en termes de fonctionnalités et de code. Néanmoins nous avons implémentés de nombreuses idées inspirées par le cours et les TP de Graphique 3D. De plus nous devions repartir de zéro à chaque fois car nous n'utilisions pas les sources des tps, ce qui fut formateur pour s'approprier les notions déjà codées des TPs. Nous sommes arrivé à quelquechose de "jouable" qui contitue, nous pensons, une bonne base pour partir dans une infinité de directions de jeux possibles.