Programmation graphique 3D
M1 Informatique I3D - S1 Automne 2020–2021
Travaux Pratiques

TP 5 : Environment map, reflection et transparence

Introduction

Dans ce TP nous allons "plonger" un objet dans un environnement et ajouter à cet objet des effets de réflection de la lumière venant de l'environnement. Nous manipulerons une carte d'environnement (environment map). C'est une image 360°, permettant de simuler un environnement dans toutes les directions. Concrètement, ce sont des textures que l'on peut donc échantillonner pour récupérer une information.

Exercice 1 : Carte d'environnement

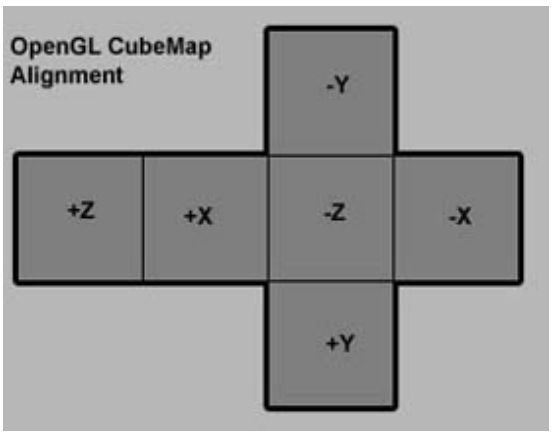
Vous allez partir du fichier *tp3_correction_02_Shading.js* du TP3.

Texture :

Vous allez commencer par ajouter une carte d'environnement à la scène. Pour faire cela en OpenGL, on utilise des textures CubeMap. Dans le précédent TP vous avez manipulé des textures du type `gl.TEXTURE_2D`, ici il s'agit de texture du type `gl.TEXTURE_CUBE_MAP`. Nous vous simplifions la tâche avec la bibliothèque easyWGL avec laquelle vous pouvez créer une texture CubeMap (doc README lignes 587-612) en utilisant :

```
var tex = TextureCubeMap();
tex.load(["cubeMap1.png", "cubeMap2.png",
"cubeMap3.png", "cubeMap4.png", "cubeMap5.png", "cubeMap6.png"]);
```

Un CubeMap est une texture construite à partir de 6 plan d'une scène 360°, en suivant l'orientation suivante :



Cette texture CubeMap sera donc plaquée sur un cube ! La bibliothèque easyWGL fourni des maillages pour certaines primitives de bases (cube, tore, sphere, cylindre ...) et le "renderer" qui va avec. Avec cette solution, pas besoin de gérer les VBO, EBO, VAO ... tout est fait pour vous ! Vous pouvez ensuite accéder dans le vertex shader aux positions, aux normales, aux coordonnées de texture et à la couleur. Regardez dans la doc (fichier README) à partir de la ligne 715 pour le maillage et à partir de la ligne 753 pour le renderer.

Shader :

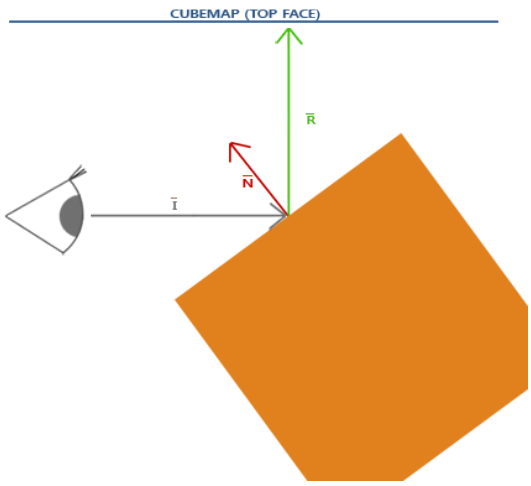
Vous allez ensuite créer les shaders pour afficher le cube avec la texture cubeMap plaquée dessus. Vous savez maintenant comment accéder à une texture dans un shader et comment l'échantillonner (récupérer une valeur dedans). Ici, il faut utiliser un uniform de type `samplerCube` et non pas `sampler2D` comme au TP précédent. L'échantillonnage ce fait ensuite par un vecteur 3D et non pas une position 2D dans la texture. Pour plaquer la texture correctement sur votre cube, il vous suffi de récupérer la position interpolée des sommets du cube (c'est enfait un vecteur direction dont l'origine est le centre du cube).

ATTENTION ! Les matrices de vue et de projection ne sont pas tout à fait les mêmes pour afficher ce cube, pour avoir l'effet d'un environnement qui nous entoure à l'infini (skybox), ces matrices sont calculées un peu différement (en particulier, on retire les translations de la matrice de vue). Vous pouvez récupérer la matrice "$projection \times vue$" avec la fonction `ewgl.scene_camera.get_matrix_for_skybox();`.

Exercice 2 : Reflections

Maintenant que l'environnement est en place, nous allons l'utiliser pour ajouter de la réfléction sur notre modèle de voiture. Ces calculs se font dans le fragment shader. Vous allez donc avoir **2 programmes de shader** : un qui contient le vertex shader et le fragment shader pour le cube avec la texture CubeMap plaquée dessus (exercice 1) et un qui contient le vertex shader et le fragment shader pour la voiture avec l'éclairage et les réflections. Pour ce dernier, vous devez :

- Envoyer la texture de la carte d'environement au fragment shader;
- Calculer le vecteur de réflction $R = I - 2.0 * dot(N, I) * N$. A partir du rayon incident I (vecteur direction de la caméra vers le fragment) et de la normal N. Pour cela, vous utiliserez la fonction GLSL `reflect(I, N)` qui renvoi un *vec3*.
  ATTENTION : I et N doivent être **normalisés** et **exprimés dans l'espace monde** pour calculer l'échantillon de la carte d'environement qui est plaquée sur le cube. Il vous faut donc **la position de la caméra dans l'espace monde**, sachant qu'elle est en [0, 0, 0] dans l'espace vue, il suffit de multiplier cette position par l'inverse de la matrice view (il y la fonction `.inverse()` sur une matrice dans la lib easyWGL) pour revenir dans le repère du monde.
- Echantillonner la texture CubeMap avec le vecteur R précédement calculé.



Pensez à la fonction GLSL `mix()` pour que la couleur du fragment soit **un mélange de la couleur calculée par l'éclairage de Blinn-Phong et de la réflection de l'environement**. Si vous ne prenez que la réflection, votre modèle sera un miroir parfait !

Vous pouvez modifier le modèle d'éclairage pour utiliser une **"directional light"**, plutôt qu'une **"point light"**. Il suffit pour cela de définir la direction de la lumière (vec3) (par exemple vec3(200, 200, -200) dans l'espace monde) plutôt que de la calculer par rapport à la position de la source lumineuse. C'est ce qui est fait pour simuler un "soleil" plutôt qu'un éclairage d'intérieur. C'est donc mieux ici dans notre environement.

Exercice 3 : Transparence

Vous allez maintenant ajouter un effet de transparence sur les vitres de la voiture :

- Dans la fonction JavaScript qui s'occupe de l'affichage, il faut penser à activer le "blending" et à décrire dans quel ordre gérer les élements transparents :

```
gl.enable(gl.BLEND);
gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
draw ...
gl.disable(gl.BLEND);
```

- Il faut envoyer le "flag de transparence" T au shader via un uniform. Ce flag est récupéré à la lecture des fichiers .json du modèle comme les informations des matériaux ka, kd, ks et ns. Il faut donc :

  1. Ajouter une variable globale `var asset_material_T_list = [];`
  2. Y mettre le flag de transparence lu dans les fichiers .json à la fin de la fonction `SceneManager_add()` : `asset_material_T_list.push( object.T );`
  3. Ajouter un uniform dans la fonction de rendu (dans la boucle qui affiche toutes les parties du maillage de la voiture) : `Uniforms.uT = asset_material_T_list[ i ];`

- Dans le fragment shader, si le flag de transparence est égale à 1 : on peut donner une valeur inférieure à 1.0 pour le canal alpha de la couleur finale (le 4ème composant du vec4).