Université Mohamed Khider-Biskra
Faculté des Sciences Exactes et Sciences de la Nature et de la Vie
Département d'informatique

Master 1 IVA **Module: SGPU**
Durée: 2 séances. **2019/2020**

TP N°04

But : manipuler la technique de Normal Mapping " bump mapping".
Rappel :

Le **bump mapping** (Textures de normales ou normal-mapping appelé aussi méthode de perturbations de normales) fait partie des techniques d'éclairage par pixel c'est à dire que tous les calculs d'éclairage (application des équations de la lumière) sont effectués pour chaque pixel. Une **bump mapping**, est une texture dans laquelle chaque **texel** contient un vecteur normal XYZ.

En gros, **vous prenez une texture classique et le but est de donner à cette texture une illusion de relief.** Le mot illusion est important puisque c'est justement parce que c'est une illusion que le truc est très intéressant.

Pourquoi modéliser un mur de pierre bout par bout, polygones par polygones alors qu'une simple couche peut donner la même impression ? Car je rappelle tout de même que modéliser un objet 3D complexe (comme un mur) demande pas mal de ressource machine. Et c'est là que le **bump mapping** trouve son utilité...

Généralement, une normal-mapping est dérivée d'une color-map. Cette dérivation se fait à l'aide d'utilitaires de création de normal-maps. Cet utilitaire, nmapgen.exe est un tout petit utilitaire en ligne de commande.
http://www.ozone3d.net/downloads/tutorials/normal_map_tutor.zip
La puissance des GPUs nous permet d'atteindre cette précision tout en conservant des frames rates acceptables. Dans chaque texel **RGB** du la texture de normales est encodé un vecteur **XYZ** : chacune des composantes d'une couleur est comprise entre **0** et **1** et chacune des composantes d'un vecteur est entre **-1** et **1**, donc la conversion d'un texel en normale s'effectue ainsi :

```
normal = (2*color)-1    // pour chaque composante
```

Pour appliquer cette technique; le gros problème est la conversion de notre normale, qui est exprimée dans l'espace de chaque triangle (espace tangent, aussi appelé espace de l'image), vers l'espace modèle (car c'est ce que l'on utilise dans notre équation d'ombrage).

Le problème est que le vecteur normal se trouve exprimé dans l'espace tangent alors que les autres vecteurs intervenants dans les calculs (vecteur lumière et vecteur vision) sont exprimés dans un autre repère (le repère de la caméra). Il faut donc exprimer tous ces vecteurs dans un même et unique repère afin que les calculs (produit scalaire principalement) aient un sens. Ce repère est l'espace tangent.

Les vecteurs normaux stockés dans la normale map « Textures de normales » sont exprimés dans l'espace tangent. D'où la couleur bleu de la normal-map car la plupart des vecteurs sont orientés suivant l'axe des Z « l'extérieur de la surface ».

La technique du bump mapping est toute simple puisqu'elle consiste à utiliser un vecteur normal déformé au niveau du pixel en cours de traitement.

Les équations de la lumières que nous allons implémenter :

If = Ia + Id + Is

Où $\mathbf{I_f}$ est l'intensité de la couleur finale du pixel, $\mathbf{I_a}$ est l'intensité de la couleur ambiante, $\mathbf{I_d}$ est l'intensité de la couleur diffuse et **Is** celle de la couleur spéculaire. Pour plus d'explications sur ces différentes composantes.

Attention : le bump mapping est par nature une technique multitexturing. Dans notre cas, 2 textures sont utilisées et que tous les calculs d'éclairages sont réalisés pour chaque pixel. La première texture utilisé est : la texture des

Normales (exemple : "**brickNormal.bmp**") et la deuxième est : la texture de l'objet (exemple : "**brickCouleur.bmp**")

Pour simplifier :

I_f = Ia + Id + Is devient → I_f = (Ka + Kd* NL)* TextureObjet + pow(NL, 30.0) ;
TextureObjet : la texture de l'objet (exemple : "**brickCouleur.bmp**")
vec4 Ka = vec4(0.6, 0.6, 0.6, 1.0);
vec4 Kd = vec4(0.8, 0.8, 0.8, 1.0);

Questions :

1ére étape :

- Afficher une sphère (Model .Obj).
- Utiliser le code du TP des textures pour réaliser ce TP. Compléter le code du fragment Shader ci-dessous pour implémenté le "bump mapping".
- Utiliser les deux images "**brickNormal.bmp**" et "**brickCouleur.bmp**" qui sont dans le répertoire image.
- Plaquer la texture "**brickCouleur.bmp**" sur la sphère et utiliser la texture "**brickNormal.bmp**" pour récupérer les normales nécessaires pour le calculer de l'éclairage.
- La position de la source de lumière doit être donnée du code c++ au shader, puis vous devez faire bouger cette source de lumière. La position de la source est initialisé par (x,y,z) = (5,5,5).
- Utiliser modèle de Phong comme modèle d'éclairement.

2éme étape :

- Afficher une autre sphère (Model .Obj).
- Plaquer la texture "**brickCouleur.bmp**" sur la sphère (utiliser les shaders pour le plaquage de texture).
- Utiliser le même modèle de Phong comme modèle d'éclairement (comme précédemment).

// Vertex Shader

```
varying vec3 lumiere; // variable globale vecteur lumière
varying vec2 texcoords; // coordonnées de texture
uniform vec4 PL; // (x,y,z) = (5,5,5) de Opengl
void main()
{
// Projection sur l'écran par un produit avec la matrice de
//transformation
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
// calculer les coordonnées globales mais non projetées
vec4 pospoint = gl_ModelViewMatrix * gl_Vertex;
// position de la lumière
vec4 poslum = PL;
// calculer le vecteur lumière en ce sommet
lumiere = vec3(poslum - pospoint);
// calculer les coordonnées de texture de ce vertex
texcoords = vec2(gl_MultiTexCoord0);
}
```

// Fragment Shader

```
……
……
uniform sampler2D Texture1; // Texture Normale
uniform sampler2D Texture2; // Texture Couleur
void main()
{
// Récupérer la texture de l'objet
vec4 TextureObjet = …….. ;
// couleurs diffuses et ambiante
……
// calculer la valeur du bump «normale » à partir d'une variable
//uniform
vec3 bump = normalize( ……. * 2.0 - 1.0);
// normalisé le vec3 bump
……..
// Calculer la formule de Lambert et empêcher les
//dépassements
vec3 L = normalize(…….);
float NL = ……..……;
// calculer la couleur diffuse finale et simuler un terme
// reflet spéculaire
        gl_FragColor = ……..;
}
```