Le BumpMapping

François DEVIC
Serge Alexandre FIEVET

24 mai 2004

Table des figures

Table des matières

Chapitre 1

Théorie sur le BumpMapping

1.1 Qu'est ce que le Bump Mapping

Le Bump Mapping est une technique qui permet d'ajouter du relief à un surface sans ajouter de la complexité dans la géométrie de la surface. Ici la géométrie d'une surface est en fait les points qui définissent les polygones de cette même surface. Cet ajout se fait grâce une texture qui est en ... 2 dimensions. Nous allons voir comment cela est possible grâce à son inventeur, James Blinn. Ensuite nous verrons deux méthodes modernes pour réaliser du Bump Mapping.

1.2 Le BumpMapping de Blinn

Pour ajouter du relief à une surface, il est possible d'utiliser du Displace Mapping. Cependant, cette technique une grande compexité dans la géométrie de la scène : elle ajoute autant de point qu'il y a de perturbations.

Blinn, en 1978, a proposé de perturber les normales de la surface plutôt que de perturber les points de la surface. Ainsi moins de calculs étaient nécessaires. Pour réaliser cela, il faut donc une texture de déformation ou *Bump map*. Après on applique cette texture sur l'objet.

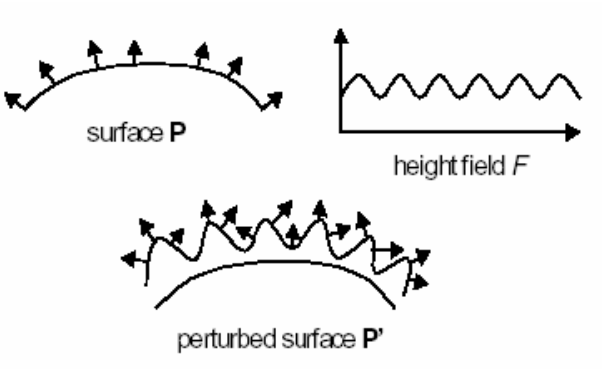

FIG. 1.1 – Application d'une texture de déformation

Nous allons voir le côté mathématique de cette technique.
Soit un point **P**. La normale en ce point est définie par :

$$N(u,v) = \frac{\partial P(u,v)}{\partial u} * \frac{\partial P(u,v)}{\partial v}$$

Soit la fonction $F(u,v)$ qui définie la force de l'élévation du relief à partir du point $\mathbf{P}(u,v)$. La direction du déplacement est définie selon la normale N. A partir de $\mathbf{P}$ et de N, on peut en déduire le nouvel emplacement **P'** :

$$\mathbf{P'} = \mathbf{P}(u,v) + F(u,v)\frac{N(u,v)}{|N(u,v)|}$$

De là, on obtient aussi la normale au point **P'** :

$$N'(u,v) = \frac{\partial \mathbf{P'}(u,v)}{\partial u} * \frac{\partial \mathbf{P'}(u,v)}{\partial v}$$

En simplifiant, on obtient :

$$N' = N + \frac{\frac{\partial F}{\partial u}(N * \frac{\partial P(u,v)}{\partial v}) - \frac{\partial F}{\partial v}(N * \frac{\partial P(u,v)}{\partial u})}{|N|}$$

Voici donc toute la théorie que Blinn a développé pour arriver à ajouter du relief sans trop de calculs. Ainsi on peut obtenir des images comme les suivantes :

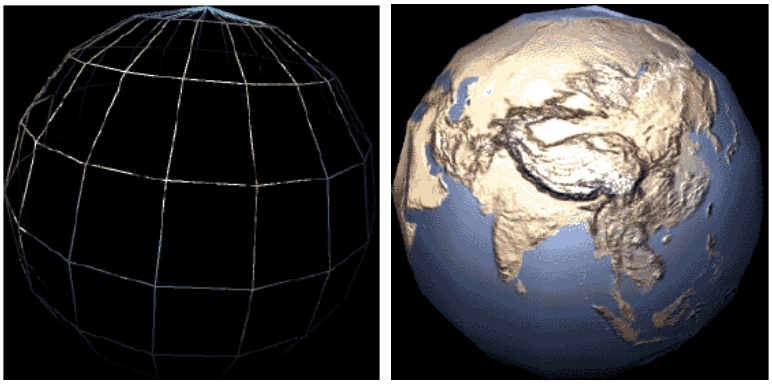

Fig. 1.2 – Une terre moins plate (*nVidia*)

Après avoir vu la première méthode pour réaliser du BumpMapping, nous allons maintenant voir des méthodes améliorées.

1.3 Les améliorations

Nous allons voir deux méthodes pour réaliser du Bump Mapping sans utiliser autant de calculs que la méthode décrite par Blinn. En effet, lorsqu'il a édité sa théorie en 1978, les ordinateurs ne pouvaient effectuer tant de calculs. Des nouvelles techniques sont donc apparues.

1.3.1 L'emboss BumpMapping

L'emboss Bump Mapping repose sur un décalage de texture. La technique d'emboss BumpMapping utilise du multitexturing et une opération de blending pour simuler la perturbation des normales. Dans cette technique, on va décaler la texture dans le sens de la lumière pour donner l'impression de relief. Plus l'intensité de la lumière va être importante, plus le décalage va être important. Ensuite on va superposer la texture décalée sur la texture de base en effectuant une soustraction de couleurs.

Cette technique est implémentée dans toutes les cartes graphiques actuelles, c'est pourquoi elle est très utilisée. Cependant, elle possède des limitations :

- une seule source lumineuse est possible ;
- un angle d'incidence trop important donne des résultats non voulus.

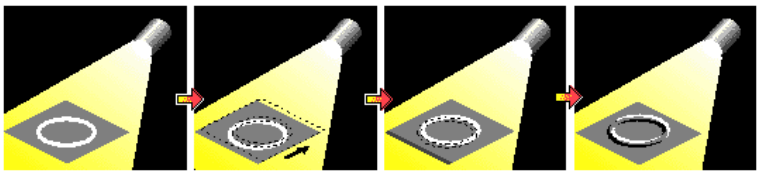

Fig. 1.3 – Décalage et soustraction de l'image

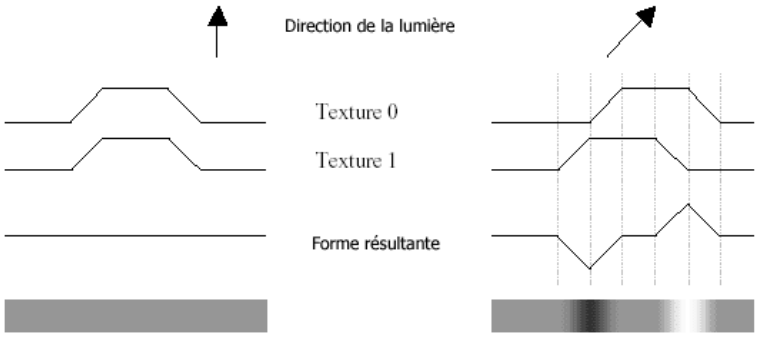

Fig. 1.4 – Emboss Bump Mapping (*3DLabs*)

Fig. 1.5 – Limitation de l'Emboss Bump Mapping

1.3.2 Dot product bump mapping

Le Dot Product Bump Mapping est une technique qui se rapproche de l'esprit original du bump mapping dans le sens où il produit un per-pixel shading : les calculs sont appliqués à chaque pixel. Dans cette technique la texture de déformation donne les normales en chaque point, cette texture sera donc appelée *normal map*. La direction de la lumière est stockée pour chaque point à la place des informations RGB : on obtient donc la *light map*.

On multiplie les vecteurs de la *light map* avec ceux de la *normal map* point par point. Le résultat fournit l'intensité lumineuse à appliquer à la texture de base.

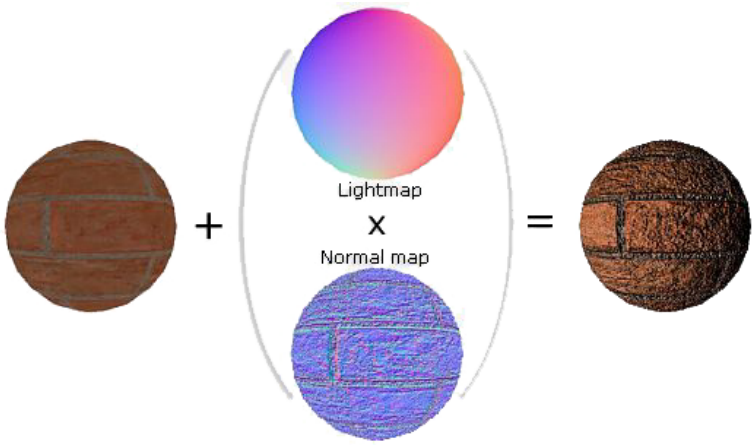

Fig. 1.6 – Le Dot Product BumpMapping

Chapitre 2

Implémentation du Bump Mapping

2.1 Sans accélération graphique

Il s'agit ici d'effectuer un bump mapping de Blinn.

Les opérations suivantes sont effectuées pour chaque pixel. D'abord, il faut commencer par calculer la normale perturbée :

```
// Bosselage :
if ( perturbationNormale ) {
    // Calcule la perturbation de la Normale
    VecteurHomogene perturb = texBoss.accesDeriveTexelU(tu,tv) * (nrm ^ plg.accesTangenteV()) -
                texBoss.accesDeriveTexelV(tu,tv) * (nrm ^ plg.accesTangenteU()) ;
    // Ajoute la perturbation
    enrm = nrm + perturb * forceBosselage_ ;
    // Normalise la normale effective
    Scalaire dist = enrm.norme() ;
    if ( dist != 0 )
        enrm = enrm / dist ;
}
else {
    enrm = nrm ;
}
```

Au départ, les scalaires *tu* et *tv* contiennent les coordonées relatives du pixel sur la texture et le vecteur *nrm* contient la normale théorique du pixel. Celle-ci peut être obtenue à partir de la normale du polygone ou extrapolée à partir des normales des sommets du polygone (modèle de Phong). Aprés le calcul, le vecteur *enrm* contient la normale effective (perturbée) du pixel.
`plg.accesTangenteU()` donne le vecteur dérivé du polygone par rapport au paramètre u de la texture ($\frac{\partial P(u,v)}{\partial u}$) et pareillement pour `plg.accesTangenteV()`.
`texBoss.accesDeriveTexelU(tu,tv)` donne la valeur dérivée de la hauteur par rapport au paramètre u de la texture ($\frac{\partial F}{\partial u}$) et pareillement pour `texBoss.accesDeriveTexelV(tu,tv)`.
L'opérateur ^ a été redéfini sur les vecteurs comme produit vectoriel (cross product).
La dérivée de la hauteur selon u est calculée comme suit. Le calcul est le même pour la dérivée de la hauteur selon v.

```
const Scalaire TextureHauteur::accesDeriveTexelU( Ratio x, Ratio y ) const {
    int texU = (int)(x * accesLargeur()) % accesLargeur() ;
    int texV = (int)(y * accesHauteur()) % accesHauteur() ;
    Scalaire precedent = ( texU > 0 ) ? (*this)( texU - 1, texV ) : (*this)( texU, texV ) ;

    Scalaire suivant = ( texU < accesLargeur() - 1 ) ? (*this)( texU + 1, texV ) :(*this)( texU, texV ) ;
    return (suivant - precedent) / 2  * accesLargeur() ;
}
```

Les paramètres x et y, qui contiennent les coordonées sur la texture, sont traités comme des réels compris entre 0 et 1. Ils sont transformés pour obtenir les coordonées entières du texel (élément de base de la texture) correspondant dans les variables *texU* et *texV*.
La dérivée est obtenue avec une différence finie entre le texel précédent et le texel suivant : $\frac{F(u+\delta u,v)-F(u-\delta u,v)}{2*\delta u}$ avec $\delta u = \frac{1}{accesLargeur}$, $accesLargeur$ étant le nombre total de texel selon u.

Ensuite, il suffit de calculer l'eclairage en accord avec la normale effective :

```
// Calcul ombrage :
if ( calculOmbrage ) {
    // Valeur de l'eclairage. Contient les composantes RVB de l'eclairage.
    lm = pscene_->accesAmbiente() ;
    // Pour chaque Lumiere :
    for ( size_t n=0; n<lrefsLumieres.size(); ++n ) {
        // Calcule le vecteur d'eclairage de cette Lumiere
        VecteurHomogene vectLm = llumieresDsCam[n].accesPosition() - pos ;
        // Normalise le vecteur d'eclairage
        Scalaire dist = vectLm.norme() ;
        if ( dist != 0 )
            vectLm = vectLm / dist ;
        // Calcule l'intensite de l'eclairage de cette Lumiere
        Ratio coeffLm = enrm.T() * vectLm ;

        if ( coeffLm > 0 ) {
            // Calcule la contribution a l'eclairage de cette Lumiere
            Couleur ecl = lrefsLumieres[n]->accesCouleur() * coeffLm ;

            // Ajoute cette contribution
            lm = lm + ecl ;
        }
    }
    // Applique l'eclairage calcule
    pixval = pixval * lm ;*
}
```

Il apparaît que, par définition, le bump mapping de Blinn s'intègre au mieux dans les algorithmes de rendu qui effectuent un calcul de l'éclairage pour chaque pixel (par exemple l'algorithme de Phong).

2.2 En utilisant OpenGL

Nous allons voir comment implémenter les 2 techniques améliorées grâce à OpenGL.

2.2.1 Initialisation

Pour les 2 techniques, il y a une phase d'initialisation assez importante. En effet, pour faire du bump mapping, OPenGL se sert d'extensions. De plus, il faut faire les calculs décrits dans 1.1 page 3.
Voici donc les extensions qu'il faut charger :

```
if(strstr(extension, "GL_ARB_multitexture") == NULL)
{
    QMessageBox::critical(this,tr("Qvue3D"),
    tr("Multi-texturing not available."));
    exit(1);
}

#ifdef WIN32
// si c'est supporté, alors on met les fonctions en place...
    glActiveTextureARB       = (PFNGLCLIENTACTIVETEXTUREARBPROC)wglGetProcAddress("glActiveTextureARB");
    glMultiTexCoord2fARB     = (PFNGLMULTITEXCOORD2FARBPROC)wglGetProcAddress("glMultiTexCoord2fARB");
    glClientActiveTextureARB = (PFNGLACTIVETEXTUREARBPROC)wglGetProcAddress("glClientActiveTextureARB");

// controle des ereurs...
if(!glActiveTextureARB || !glMultiTexCoord2fARB || !glClientActiveTextureARB)
{
    QMessageBox::critical(this,tr("QVue3D"),
    tr("Error setting up multitexturing."));
    exit(1);
}
#endif //WIN32

if(!IsExtensionSupported("GL_EXT_texture_env_combine"))
{
    QMessageBox::critical(0,tr("OpenGL Error"),tr("GL_EXT_texture_env_combine not found!"));
    dot3_=false;
}
```

```
if(!IsExtensionSupported("GL_EXT_texture_env_dot3"))
{
    QMessageBox::critical(0,tr("OpenGL Error"),tr("GL_EXT_texture_env_dot3 not found!"));
    dot3_=false;
}
```

Il faut donc charger les extensions qui gérent le multitexturing ainsi que celle pour le dot3.

Ensuite, pour notre part, osont calculés tous les points, les normales, les tangentes des modèles qui vont être affichés. C'est pourquoi, on ne peut changer la taille des objets. Nous n'expliquerons pas ici les calculs des points, des normales pour aucun de ces objets car ils ne sont pas directement reliés au Bump Mapping mais sont des techniques d'infographie générale ([MALGOUYRES 2002]).

Pour chaque technique nous avons redéfini les méthodes d'affichages pour le cube, le plan et la sphère.

2.2.2 Emboss Bump Mapping

Pour réaliser de l'Emboss Bump Mapping, il y a aussi des initialisations à faire.

```
if(lightEnable_)
    glEnable(GL_LIGHTING);
else
    glDisable(GL_LIGHTING);

// mise en place du blending.
glEnable(GL_BLEND);
//alpha destination a 0
glBlendFunc(GL_SRC_ALPHA, GL_ZERO);

// on indique la texture 0 est utilisée par la texture normale
glActiveTextureARB(GL_TEXTURE0_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, bumpMap.ID);

// mis en place de l'environnement pour la première texture
//  Modulate the RGB by 2.
glTexEnvf( GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE, GL_COMBINE_EXT );
glTexEnvf( GL_TEXTURE_ENV,  GL_COMBINE_RGB_EXT, GL_MODULATE );
glTexEnvf( GL_TEXTURE_ENV,  GL_RGB_SCALE_EXT, 2.0f );

// On indique qu'on va remplacer le canal alpha.
glTexEnvf(GL_TEXTURE_ENV, GL_COMBINE_ALPHA_EXT, GL_REPLACE );

// La texture numéro 1 est utilisée comme bump map
glActiveTextureARB(GL_TEXTURE1_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, bumpMap.ID);

// mis en place de l'environnement pour la texture de bump map
// Modulate by 1.
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_EXT );
glTexEnvf(GL_TEXTURE_ENV, GL_COMBINE_RGB_EXT, GL_REPLACE );
glTexEnvf(GL_TEXTURE_ENV, GL_RGB_SCALE_EXT, 1.0f );

//utilise les résultats de la texture précédente comme
// source pour la suivante
glTexEnvf(GL_TEXTURE_ENV, GL_SOURCE0_RGB_EXT, GL_PREVIOUS_EXT );
glTexEnvf(GL_TEXTURE_ENV, GL_COMBINE_ALPHA_EXT, GL_ADD_SIGNED_EXT );

// ici nous allons utiliser les résultats de la précédente texture
// pour combiner l'image avec son bump map correctement
glTexEnvf(GL_TEXTURE_ENV, GL_OPERAND0_ALPHA_EXT, GL_ONE_MINUS_SRC_ALPHA );
glTexEnvf(GL_TEXTURE_ENV, GL_SOURCE0_ALPHA_EXT, GL_TEXTURE );
glTexEnvf(GL_TEXTURE_ENV, GL_SOURCE1_ALPHA_EXT, GL_PREVIOUS_EXT );
```

Nous voyons qu'il y a de très nombreuses initialisations à réaliser ici aussi. Ces initialisations sont réalisées avec les extensions OpenGL. Une fois ces initialisations réalisées à chaque image dessinée, on peut dessiner effectivement le modèle. La première étape est de recalculer l'emboss, en effet, la lumière peut avoir changé ainsi que l'orientation du modèle. Ensuite on dessine le modèle normalement. Dans la fonction decaleTexture, on décale la texture dans le sens de la lumière.

2.2.3 Dot Product Bump Mapping

Ici aussi, avant de pouvoir dessiner une image, de nombreuses initialisations sont nécessaires.

```
glActiveTextureARB(GL_TEXTURE0_ARB);
if(dot3ID_)
{
        glEnable(GL_TEXTURE_2D);

        //initialisation normale d'une texture
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
        glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

        glBindTexture(GL_TEXTURE_2D, dot3ID_);
}
// mise en place du dot3
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_EXT);
glTexEnvf(GL_TEXTURE_ENV, GL_COMBINE_RGB_EXT, GL_DOT3_RGB_EXT);
glTexEnvf(GL_TEXTURE_ENV, GL_SOURCE0_RGB_EXT, GL_PRIMARY_COLOR_EXT);
glTexEnvf(GL_TEXTURE_ENV, GL_OPERAND0_RGB_EXT, GL_SRC_COLOR);
glTexEnvf(GL_TEXTURE_ENV, GL_SOURCE1_RGB_EXT, GL_TEXTURE);
glTexEnvf(GL_TEXTURE_ENV, GL_OPERAND1_RGB_EXT, GL_SRC_COLOR);

glActiveTextureARB(GL_TEXTURE1_ARB);
if(dot3IDIm_)
{
        glBindTexture(GL_TEXTURE_2D, dot3IDIm_);
}
glEnable(GL_TEXTURE_2D);
```

Ici aussi ce sont les extensions qu'il faut mettre en place. Une fois ceci fait, il faut calculer les *lightmap* et *normalmap*. Il y a 2 types de calculs de *normalmap* : un pour les images en couleurs et un pour les images en niveaux de gris. Tous ces calculs sont effectués grâce à la classe Dot3.

Chapitre 3

Les programmes

3.1 Rendu logiciel

Le programme testQTRendu permet d'afficher un effet de Bump Mapping avec la méthode de Blinn. Il utilise la bibliothèque QT pour son interface graphique et génère lui-même l'affichage de l'effet.

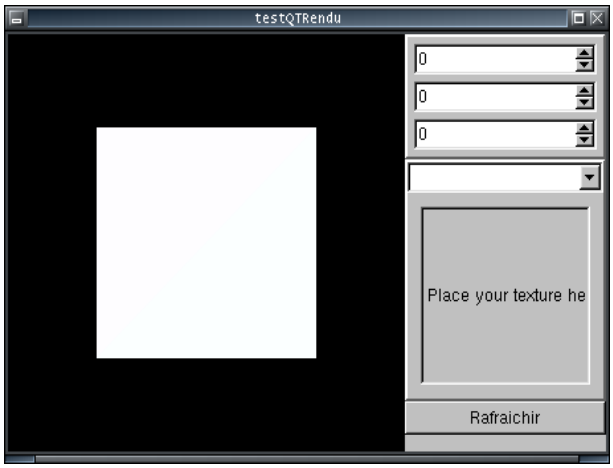

Fig. 3.1 – Présentation génerale

Il présente une zone d'affichage, trois champs de contrôle d'orientation de la scène et une zone de sélection de texture.

3.1.1 Traitements effectués

testQTRendu affiche un cube eclairé par une forte lumière blanche et trois faibles lumières colorées éloignées les unes des autres.

On peut appliquer une texture et un bosselage sur ce cube (en fait, sur la face avant et arrière). testQTRendu ne disposant pas d'un mécanisme pour extraire une carte de texture ou de bosselage à partir de n'importe quelle image, il ne peut traiter que des images dont les canaux RVB contiennent la texture et le canal Alpha contient le bosselage.

On peut aussi appliquer des couleurs au cube, ces couleurs étant définies arbitrairement dans le code du programme.

Enfin, on peut choisir un traitement de l'eclairage par facette, avec interpolation de Gourraud (interpolation des eclairages entre les sommets) ou avec interpolation de Phong (interpolation des normales entre les sommets).

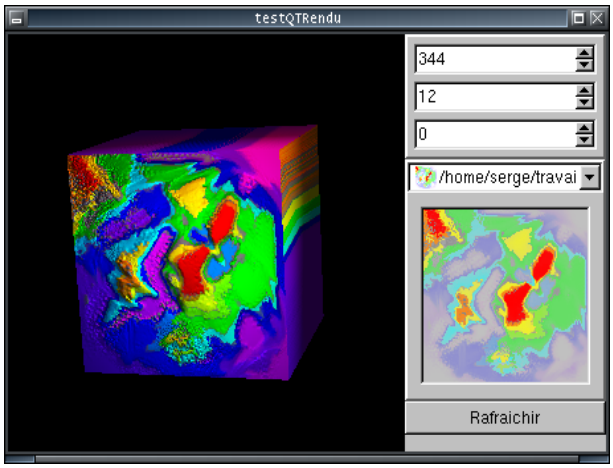

Fig. 3.2 – Une texture, un bosselage, le tour est joué

Pour les deux derniers modes d'eclairage, il est possible d'utiliser comme données de départ les normales des sommets ou celles des facettes. Ce deuxième choix peut sembler idiot, car il revient (presque) à faire un eclairage par facette, mais il peut être nécessaire pour, par exemple, afficher du bump mapping sur une forme qu'on ne désire pas arrondir. En effet, testQTRendu n'affiche le bump mapping que dans le mode d'interpolation de Phong.

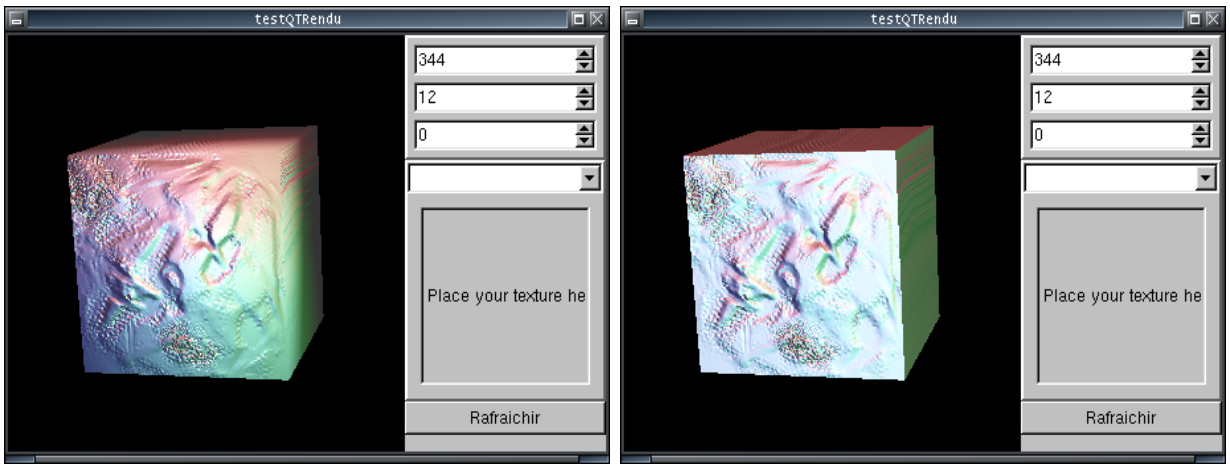

Fig. 3.3 – Avec les normales des Sommets, et avec la normale de la facette

Il est à noter que les jonctions de la texture et du bosselage entre les polygones présentent un phénomène étrange et indésirable. Cela vient de ce que l'algorithme utilisé pour interpoler les coordonées des pixels sur la texture est excessivement approximatif, au point d'être visiblement incorrect. Ce problème difficile n'a pas été traité plus profondément.

3.1.2 Selection des textures

Pour ouvrir des textures, il faut cliquer droit dans la zone de selection des textures et choisir *Ajouter une texture*. Cette option va ouvrir un boîte de dialogue qui permet de séléctionner des fichiers images (pgm, pbm, ppm, png, bmp, xpm). Une fois les fichiers choisis, ils sont ajoutés à la liste des textures. Ensuite, on peut les choisir dans le menu déroulant. Lorsqu'une texture est choisie, il faut encore cliquer sur *Rafraichir* pour l'appliquer à la scène.

3.1.3 Contrôle de la Scène

L'orientation de la scène peut être contrôlée de deux façons :

1. en cliquant sur la zone d'affichage, et en déplaçant la souris (un clic droit et un clic gauche permettent de contrôler des axes différents) ;
2. en modifiant les valeurs des champs de contrôle de l'orientation.

Le mode de traitement de l'eclairage peut être sélectionné avec la touche 'm', le type de normales utilisées (sommets ou facettes) peut être sélectionné avec la touche 'n', la texture peut être activée et désactivée avec la touche 't', les couleurs du cube peuvent être activées et désactivées avec la touche 'c', et le bump mapping peut être activé et désactivé avec la touche 'b'. Il peut être nécessaire de cliquer préalablement sur la zone d'affichage pour utiliser ces commandes.

Que le bump mapping soit activé ou non, il ne sera visible que dans le mode de traitement de l'eclairage de Phong (appuyer sur 'm' jusqu'à obtenir le résultat souhaité).

3.2 Avec OpenGL

Ce programme, réalisé en Qt et OpenGL, permet de visualiser les effets d'Emboss Bump-Mapping et de Dot3 Bump Mapping grâce à OpenGL. Ce programme fonctionne aussi bien sous Windows que sous Linux grâce à Qt. Nous allons montrer ici le fonctionnement de cette apllication.

Toutes les fonctionnalités de cette apllication sont disponibles dans le menu **Affichage**.

3.2.1 Le démarrage

Voici l'écran qui apparaît au démarrage de l'application :

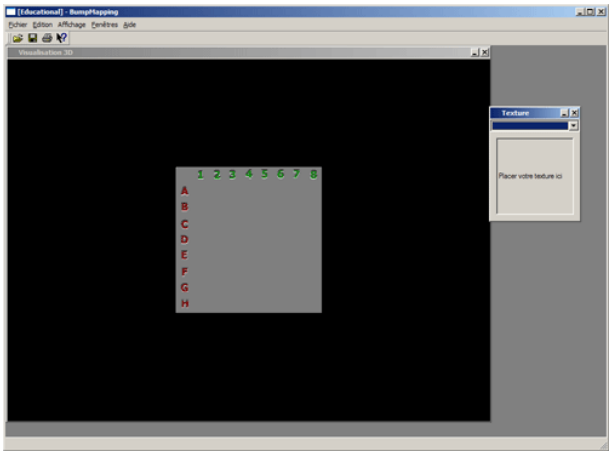

Fig. 3.4 – L'interface graphique

Dans cette interface, se trouvent le widget pour gérer les textures (3.5 page suivante) ainsi que la fenêtre principale(3.6 page suivante).

La face affichée utilise la technique d'Emboss BumpMapping. Aucune texture n'est chargée dans le widget de texture.

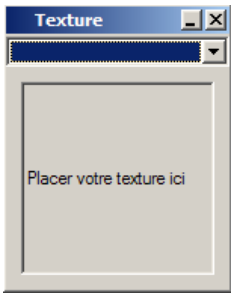

Fig. 3.5 – Le widget pour les textures

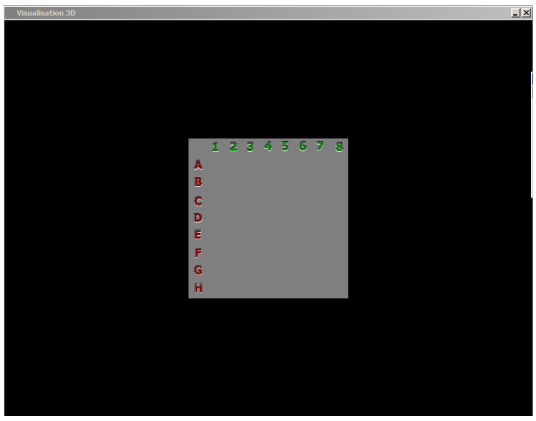

Fig. 3.6 – Le widget pour les textures

3.2.2 La gestion des textures

Il y a 3 moyens pour ajouter des textures dans l'applications :

1. en cliquant droit dans le widget des textures et en choisissant *Ajouter une texture*. Cette option va ouvrir un boîte de dialogue qui permet de séléctionner des fichiers images (pgm, pbm, ppm, png, bmp, xpm). Une fois les fichiers choisis, ils sont ajoutés à la liste des textures ;
2. en cliquant-déposant la texture directement dans le widget des textures ;
3. en cliquant-déposant la texture dans la fenêtre principale.

Une fois les textures ajoutées, on peut les choisir dans la liste des textures.

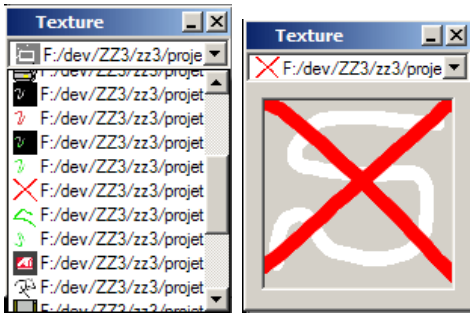

Fig. 3.7 – Choix d'une texture

Une fois la texture choisie, il suffit de la cliquer-déposer dans la fenêtre principale pour qu'elle soit affichée sur le modèle choisi.

3.2.3 Le menu affichage

le choix du mode d'affichage

Cette application permet de visualiser les modèles de trois façons différentes : en mode points, en mode ligne et mode plein. Le premier mode n'affiche que les points du modèles, le deuxième mode affiche les arrêtes et le dernier affiche le modèle complet.

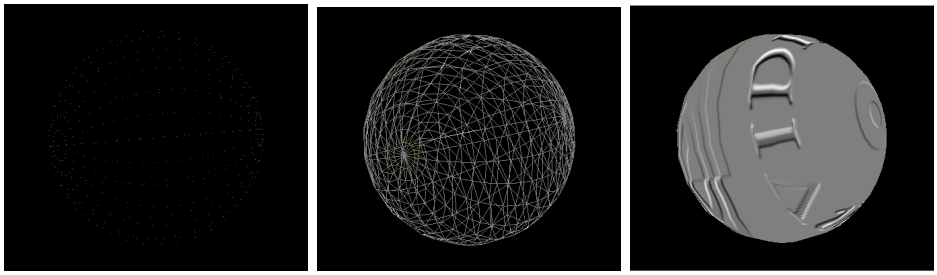

Fig. 3.8 – Les 3 modes d'affichage

Le choix de la méthode de BumpMapping

Là aussi, il existe 3 méthodes : l'emboss Bump Mapping, le dot3 et la méthode sans Bump ! !

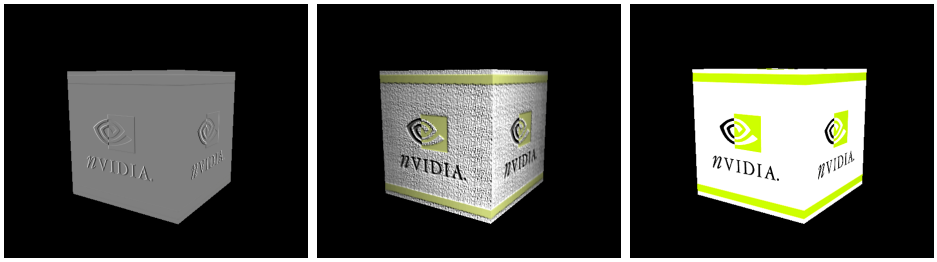

Fig. 3.9 – Les 3 méthodes de bump appliquées à un cube (emboss, dot3, sans bump)

Le choix du modèle

Cette application permet de visualiser le comportement du bump mapping sur 4 types de modèles : un triangle, une face plane, une cube et enfin une sphère.

Fig. 3.10 – Les quatre modèles géomètriques de l'application

Ce sont les 3 options les plus importantes de ce menu. On peut aussi activer et désactiver la lumière et les textures.

3.2.4 L'Emboss BumpMapping

Comme nous l'avons vu précédemment, nous pouvons choisir d'effectuer de l'Emboss Bump Mapping (3.2.3 page précédente). Pour changer la texture à appliquer sur le modèle, nous avons vu qu'il faut faire un cliquer-déposer d'une image. Il existe deux types d'images dans ce cas précis : les images avec Canal Alpha et celles sans. Ceci est très important car nous avons vu que nous nous servions du canal Alpha pour créer l'Emboss avec OpenGL. Lorsqu'une image n'a pas de canal Alpha alors l'application propose d'en créer un. Celui est en fait la moyenne des 3 composantes couleur.

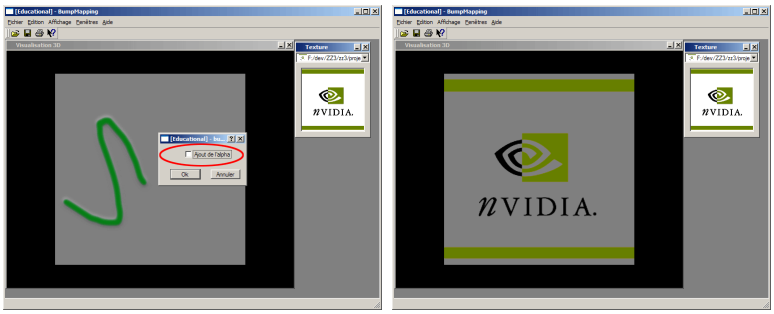

Fig. 3.11 – Insertion d'une image sans alpha

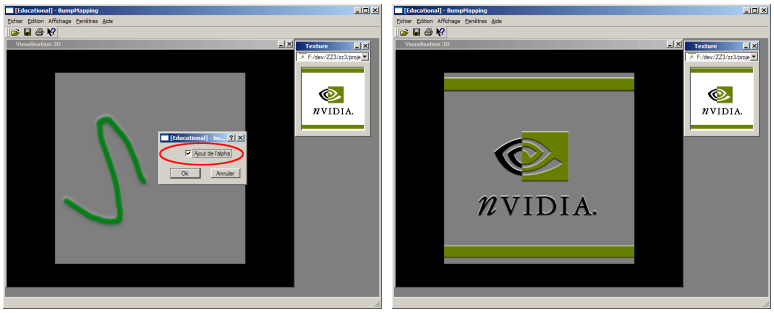

Fig. 3.12 – Insertion d'une image avec alpha

3.2.5 Le Dot3

Comme pour l'Emboss on peut changer de texture pour le Dot3 sur le modèle. Cependant, il y a 2 façons de les appliquer, il y a la texture de couleur et la texture de bump. Ainsi on peut obtenir des résultats très bizarres.

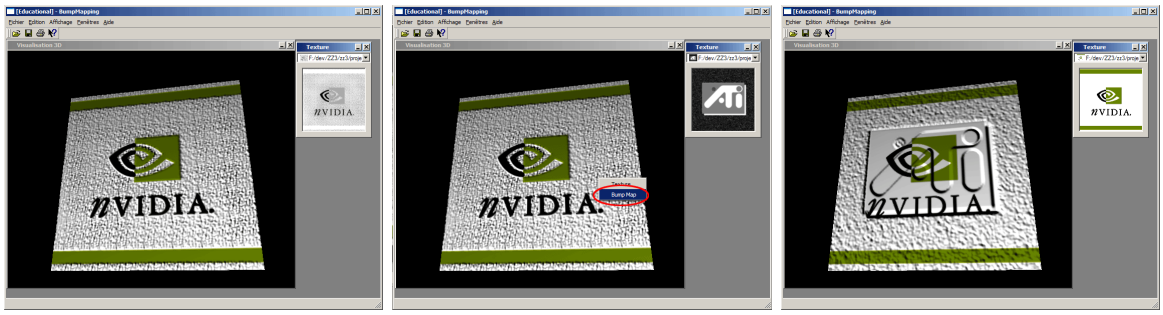

Fig. 3.13 – Transformation de nVidia grâce au Dot3

Conclusion

Ce projet nous a permis d'apprendre beaucoup de choses vis à vis du BumpMapping mais aussi de l'infographie en général et des bibliothèques graphiques. Cependant nous n'avons pas eu le temps d'approfondir et d'utiliser des bibliothèques comme Cg de nVidia, HLSL de Microsoft, RenderMonkey d'ATI qui sont des langages de shaders. Nous n'avons pas non plus utiliser DirectX de Microsoft qui est spécialement dédié à Windows. Mais ce dernier point était un choix effectué dès le début du projet car nous voulions un projet multiplateforme.

Bibliographie

[MALGOUYRES 2002] R. MALGOUYRES, *Algorithmes pour la synthèse d'images et l'animation 3D*, DUNOD, Paris, 2002

[WATT 2000] A. WATT, *3D Computer Graphics third Edition*, Addison-Wesley, Harlow, 2000

[WOO et all 2000] M.WOO, J. NEIDER, T. DAVIS, D. SHREINER, *OpenGL 1.2*, Campus-Press, Paris, 2000

[FERNANDO & KILGARD 2003] M.J. KILGARD, R. FERNANDO, *The Cg Tutorial*, Addison-Wesley, Boston, 2003

[McCUSKEY 2002] M. McCUSKEY, *Special effects Game Programming with DirectX*, Premier Press, USA, 2002

Sites Internet

Bump Mapping how it works ?,
http ://www.tweak3d.net/articles/bumpmapping/

Bump Mapping on ATI Rage 128,
http ://www.ati.com/developer/sdk/rage128sdk/Rage128BumpTutorial.html

Bump Mapping on Consumer 3D Graphics Accelerators,
http ://www.ati.com/developer/sdk/RADEONSDK/Html/Tutorials/RADEONBumpMap.html

Le site de nVidia : http ://developer.nvidia.com

Comprendre la 3D : Bump Mapping et compagnie, http ://www.onversity.com/ ultimategameprogramming
Bump Mapping, http ://www.cs.wpi.edu/ matt/courses/cs563/talks/bump/bumpmap.html

Bump Mapping, www.siggraph.org