28 avril 2014

Master Informatique

Programation Graphique Haute-Performance

Documents non autorisés - Durée 1h30

Vos réponses doivent être claires et concises (respect des maxima indiqués). L'examen est long mais noté sur 25, il n'est donc pas indispensable de répondre à toutes les questions.

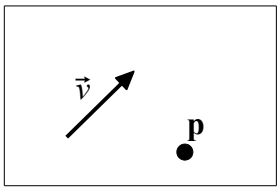
A Transformations de l'espace [5pts]

Dans cet exercice nous supposerons que nous avons accès à une fonction *rot(θ,a)* qui retourne une matrice de rotation d'angle θ autour de l'axe a, une fonction *trans(t)* qui retourne une matrice effectuant une translation d'un vecteur t, et une fonction *scale(sx,sy,sz)* qui retourne une matrice effectuant une mise à l'échelle non uniforme. Ces trois fonctions retournent des matrices 4x4.

A.1 Préliminaires.

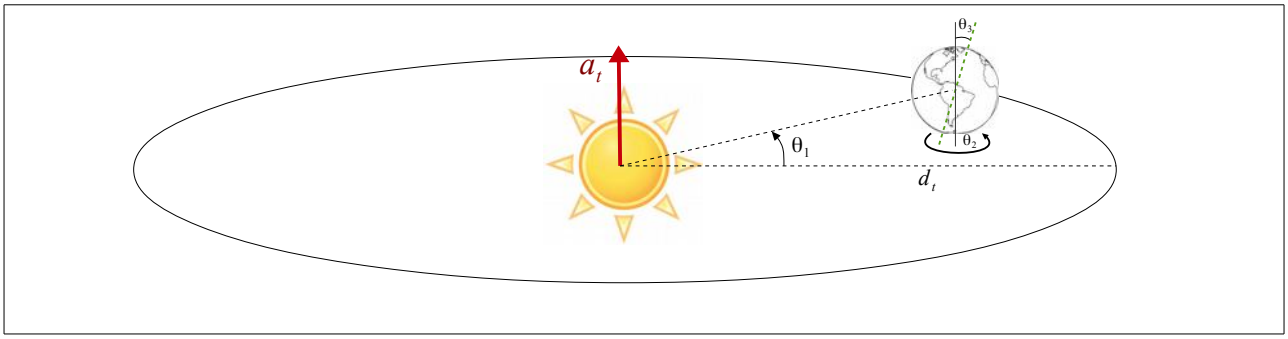
A.1.1 Donner la matrice retournée par *scale(sx,sy,sz)*.

A.1.2 Un *vecteur* 3D $\vec{v}$ ou un *point* 3D **p** peuvent chacun être représenté par leur coordonnées $[v_x, v_y, v_z]$ et $[p_x, p_y, p_z]$ respectivement. Donner une différence fondamentale entre la notion de *vecteur* et de *point*.



A.2 Système solaire.

Nous supposons que nous disposons du maillage d'une sphère texturée centrée en *[0,0,0]*, et de rayon *1*. Nous souhaitons maintenant simuler un système solaire où le soleil est centrée en *[0,0,0]* comme illustré ci-dessous. Nous considéreront la planète Terre de rayon r_t, située à une distance d_t du soleil, et tournant autour du soleil selon l'axe a_t.



A.2.1 Dans un premier temps nous négligeront l'inclinaison de la Terre, c'est à dire, que nous supposeront que les pôles de la Terre sont alignés sur l'axe a_t. A l'aide des fonctions précédentes, donner la transformation T1 à appliquer au sommet de notre maillage afin d'afficher la planète Terre à instant t donné par un angle de rotation θ_1 autour du soleil, et θ_2 autour d'elle même.

A.2.2 A partir de T1, donner la transformation T2 permettant de considérer une inclinaison de θ_3 degré de l'axe de la Terre dans la plan contenant le Soleil, la Terre, et l'axe a_t.

B Rendu temps-réel / OpenGL [13 pts]

B.1 Rastérization [10 pts]

B.1.1 Décrire l'algorithme permettant l'élimination des parties cachées mis en œuvre par les cartes graphiques. Il s'agit de l'algorithme permettant un rendu correct de la scène quelque soit l'ordre de tracé des polygones. (6 lignes max)

B.1.2 A quelles étapes du pipeline OpenGL peut-on appliquer cet algorithme ? Quel est-le choix le plus efficace ? (3 lignes max)

B.1.3 Comment rendre cet algorithme compatible avec les objets semi-transparents, justifier. (4 lignes max)

B.1.4 Décrire brièvement le principe des *Shadow-Map* pour le rendu des ombres portées. Donner un point fort et un point faible de cette méthode. (7 lignes max)

B.1.5 Expliquer comment adapter le principe des *shadow-maps* pour simuler un vidéo projecteur projetant une image sur une scène non plane. (3 lignes max)

B.1.6 Peut-on utiliser le mip-mapping sur une carte de normales ? Justifier. (4 lignes max)

B.1.7 Voici un exemple de shader effectuant le rendu d'un objet blanc, purement diffus pour une source lumineuse ponctuelle. Il y a au moins deux erreurs au niveau du calcul de l'éclairage. Quelles sont elles ?

```
/* vertex shader */
in vec3 vtx_pos;
in vec3 vtx_normal;
out vec3 normal;
out vec3 pos;
uniform mat4 Mobj,  // position/orientation
                    // de l'objet
             Mcam,  // transf. du repère monde
                    // au repère de la caméra
             Mproj; // matrice de projection
void main() {
  gl_Position = Mproj*Mcam*Mobj*vec4(vtx_pos,1);
  pos = vtx_pos;
  normal = vtx_normal;
}
```

```
/* fragment shader */
in vec3 pos;
in vec3 normal;
uniform vec3 light_pos; // position de la caméra
                        // dans le repère monde

void main() {
  vec3 light_dir = normalize(light_pos - pos);
  float d = max(0,dot(normal,light_dir));
  gl_FragColor = vec4(d,d,d,1);
}
```

B.2 Scènes complexes [3 pts]

L'on souhaite mettre en œuvre un moteur de rendu temps réel d'une scène de forêt composée de

milliers d'arbres. Chaque arbre est initialement représenté par des millions de polygones pour le tronc et les détails de l'écorce, ainsi que des milliers de feuilles, chaque feuille étant elle même représentée par quelques centaines de polygones pour reproduire fidèlement leur dentelure. On suppose que les détails de l'écorce sont visuellement répétitifs.

B.2.1 Expliquer comment représenter de manière plus efficace un tel arbre, c'est à dire de la manière la plus compacte possible (cout mémoire) tout en permettant un rendu efficace avec un maximum de détails à la fois au niveau de l'écorce et des feuilles. (7 lignes max)

C CUDA [7 pts]

C.1 Voici un exemple de code CUDA calculant la valeur minimal d'un tableau :

```
 1 : __GLOBAL__ void kernel(float *data, float *min, int n) {
 2 :   int tid = threadIdx.x + blockIdx.x * blockDim.x;
 3 :   if(tid<n && data[tid] < *min)
 4 :     *min = data[tid];
 5 : }
 6 :
 7 : float compute_min(float *data, int n) {
 8 :   float *d_data;
 9 :   cudaMalloc(&d_data, sizeof(float)*n);
10 :   cudaMemcpy(d_data, data, sizeof(float)*n, cudaMemcpyHostToDevice);
11 :
12 :   float min = data[0];
13 :   dim3 DimBlock((n+255)/256,1,1);
14 :   dim3 DimGrid(256,1,1);
15 :   kernel<<<DimGrid,DimBlock>>>(d_data, &min, n);
16 :
17 :   return min;
18 : }
```

C.1.1 Ce code calcule-t-il la bonne valeur ? (justifier en 1-2 phrases)

C.1.2 Décrire, sans fournir de code, une meilleure stratégie pour calculer la valeur minimale d'un tableau en CUDA. (5 lignes max)

C.2 Dans le contexte d'un logiciel de retouche photo, on propose d'appliquer un filtre complexe `filter()` d'antialiasing sur les silhouettes de l'image. Pour ce faire, on propose une implémentation CUDA où chaque thread est responsable d'une ligne entière de l'image. Nous proposons les deux kernels suivants (certaines parties ont été omises par souci de concision) :

```
__GLOBAL__ void
kernel1(float *img, int width, int height) {
  int i = threadIdx.x + /* … */;

  for(int j=0;j<width;++j)
    if(is_silhouette(img,i,j))
      filter(img,i,j);

}
```

```
__GLOBAL__ void
kernel2(float *img, int width, int height) {
  int i = threadIdx.x + /* … */;
  int j=0;
  while(j<width) {
    while( j<width && !is_silhouette(img,i,j) )
      ++j;
    if(j<width)
      filter(img,i,j);
  }
}
```

C.2.1 Quelle version est la plus performante ? Pourquoi ? (4 lignes max)

C.2.2 En remarquant que le nombre de pixels silhouettes est très variable d'une ligne à l'autre, proposer une stratégie encore plus performante. (donner le principe général en 6 lignes max)