TP02 : ray casting

Initialisation

Vous implémenterez le ray casting dans Gratin dans un noeud “genericImage”. Dans un tel noeud, un quad est plaqué sur le viewport. Le fragment shader est donc appliqué indépendemment sur chacun des pixels de l'image à synthétiser. Le ray casting peut alors s'effectuer naturellement sans même avoir à parcourir les pixels.

Modifiez les settings du noeud pour enlever la texture d'entrée (mettez une taille par défaut, par exemple 512x512). Mettez ensuite une couleur par défaut sur l'image de sortie. Notez que la variable texcoord contient la coordonnée du pixel courant (normalisée entre 0 et 1).

Objectifs

Les objectifs de ce TP consistent à reproduire une scène simple en ray-casting avec les outils vus en cours :

- Définition d'une caméra perspective pour la génération des rayons
- Définition d'un moins une surface planaire dans la scène
- Définition d'au moins une sphère dans la scène
- Intersection du rayon avec la scène (plan(s) + sphere(s))
- Affichage des normales et des profondeurs

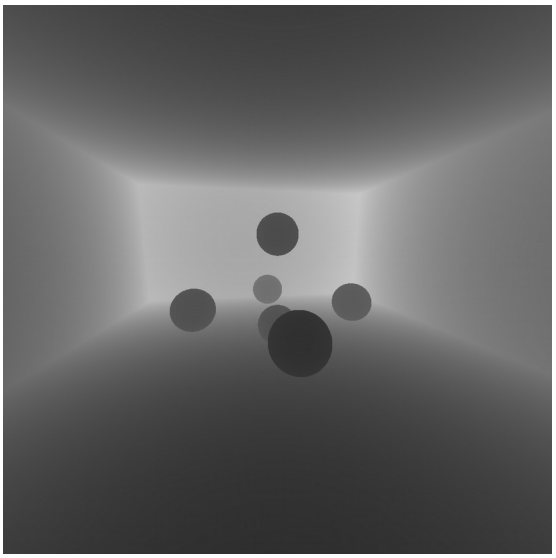


Illustration 1: Profondeurs

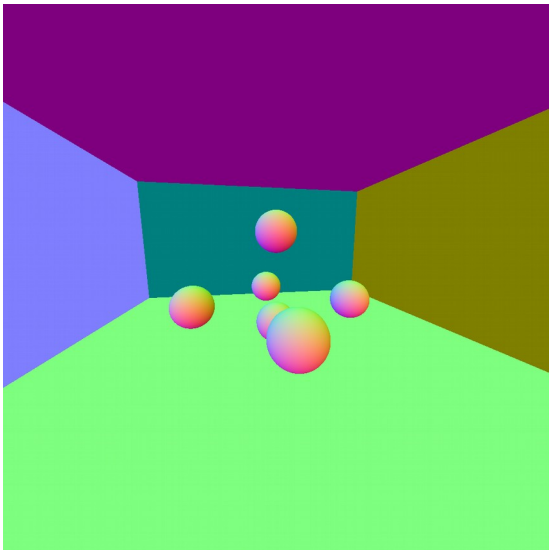


Illustration 2: normales

Rappel ray-casting

- Pour chaque pixel
  - generer un rayon avec votre caméra
  - pour chaque objet de la scène (plans+spheres)
    - tester l'intersection entre le rayon et l'objet
    - se rappeler de cet objet si la distance est inférieure aux tests précédents
  - calculer la normale de l'objet intersecté
  - calculer la couleur du pixel (affichage normale ou profondeur ici)

GLSL tips

Pour utiliser une plus grande précision des flottants, ajouter cette ligne en haut de votre shader :

```
precision highp float;
```

Vous pouvez définir des structures en GLSL (comme en C). Par exemple, la définition d'un rayon et d'un plan peuvent se faire de la manière suivante :

```
// ray structure
struct Ray {
vec3 ro; // origin
vec3 rd; // direction
};

// plane structure
struct Plane {
vec3 n; // normal
float d; // offset
};
```

L'instantiation d'un objet peut se faire de la manière suivante :

```
Plane pl = Plane(vec3(0,1,0),0); // plan y=0
```

Vous pouvez créer des tableaux en GLSL. Attention néanmoins, la taille des tableaux doit être constante (les tableaux dynamiques et les pointeurs n'existent pas) :

```
Plane pls[2] = Plane[](Plane(vec3(0,1,0),0), Plane(vec3(0,-1,0),5)); // tableau de 2 plans
```

Enfin, il est possible d'écrire des fonctions (très conseillé) et de faire des boucles, exactement de la même manière qu'en C :

```
for(int i=0;i<2;++i)
    computeIntersectionWithPlane(pls[i]);
```

Caméra

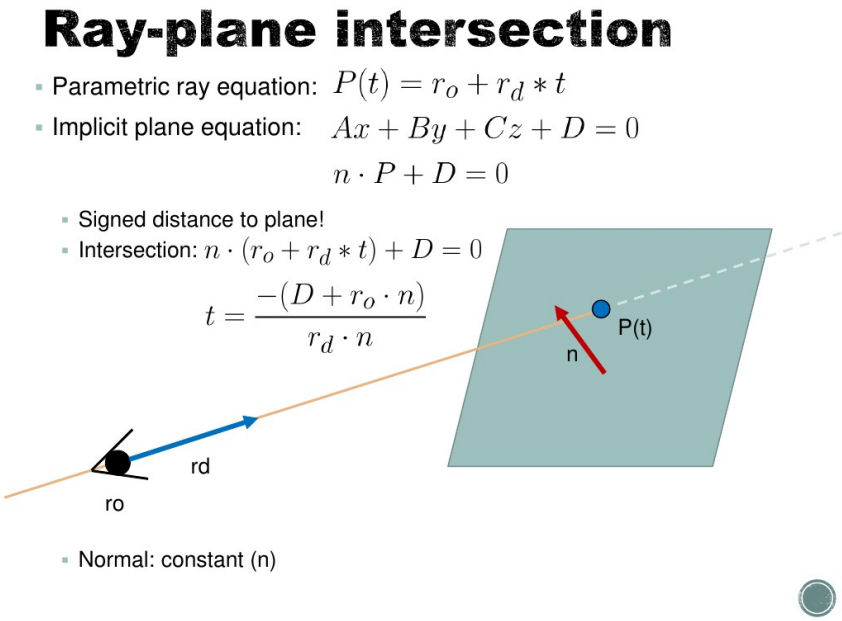
Rappel de la formule de génération d'un rayon :

x et y : coordonnées du pixel (entre -1 et 1)
u, v et w : repère orthogonal de la caméra
ro = e : position d'origine du rayon (= eye)
rd = vecteur unitaire de la caméra vers la scène
D = 1/tan(alpha/2), alpha étant l'angle de vue

$$r = (x.u, y.v, D.w)$$
$$r_d = r/||r||$$
$$r_o = e$$
$$P(t) = r_o + r_d * t$$

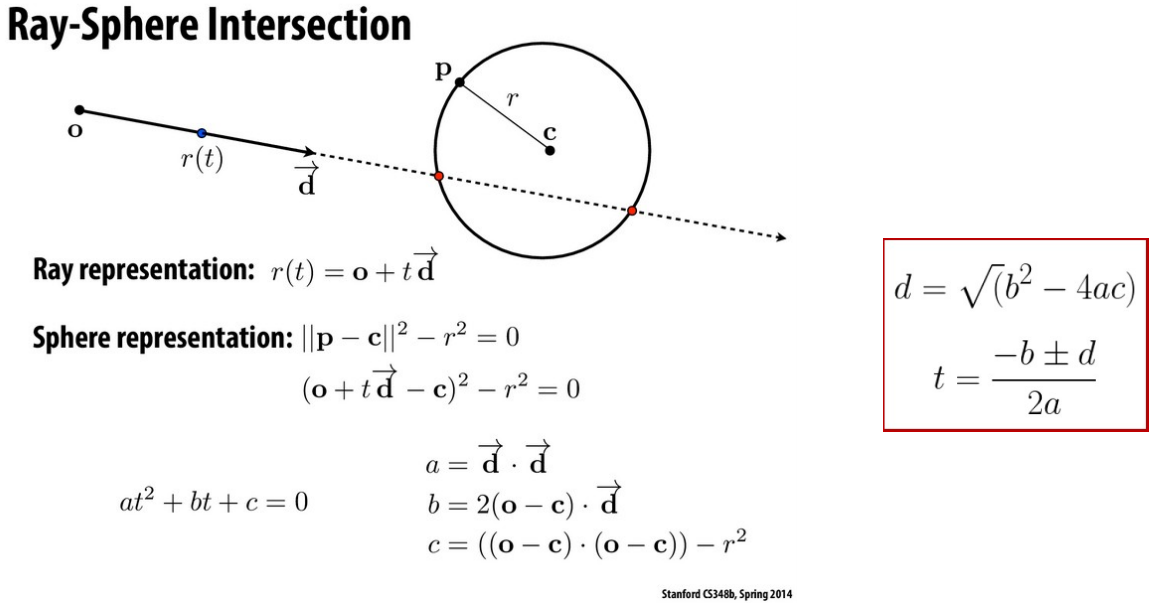
Le plus simple est de déterminer ces variables en partant de la position de la caméra (à vous de décider) et du point vers lequel la caméra regarde (à vous aussi de décider). Le vecteur w peut alors être facilement déterminé (rappel : u,v et w sont unitaires). Le vecteur v peut être initialisé à (0,1,0) pour dire que la caméra est orientée en haut (à l'endroit). Le vecteur u peut alors être déduis avec le produit vectoriel des 2 autres. Attention à refaire cette opération pour déterminer le vrai vecteur v (sans quoi votre repère ne sera pas orthogonal). Une fois le repère déterminé, les autres paramètres sont immédiats.

Intersection rayon – plan



Le test d'intersection entre le rayon généré et le plan doit renvoyer une distance (la valeur t calculée à partir de l'equation du plan et des paramètres du rayon comme décrit dans le slide ci-dessus). Si jamais votre scène possède plusieurs objets, il est important que votre fonction renvoie non seulement la distance, mais aussi un identifiant qui permettra de retrouver l'objet testé par la suite (pour calculer sa normale, sa couleur, etc).

Intersection rayon – sphère



$$d = \sqrt{(b^2 - 4ac)}$$

$$t = \frac{-b \pm d}{2a}$$

Note : cette version est légèrement différente de celle vue en cours (pas entièrement développée) et permet de calculer l'intersection avec moins de calculs.

Pour rappel, la distance à conserver est la distance positive minimum. Si le déterminant est négatif, il n'y a pas de solution. S'il est égal à 0, il n'y en a qu'une (silhouette).

Dernier conseils

Découpez bien votre shader en sous fonctions pour pouvoir tester facilement. Si vous devez débugger, sachez que le printf n'existe pas en GLSL. Il faut alors improviser et débugger en couleurs (i.e. afficher du rouge ou vert pour tester une condition par exemple) !

Bonus

- Animer vos éléments, animez votre caméra grâce aux keyframes et à un timer (ou aux positions de la souris)
- Ajoutez des éléments à votre scène
  - spheres / plans
  - tenter un nouvel objet (par exemple un cylindre → comment le représenter et comment gérer l'intersection ?)
- Essayer d'effectuer un shading simple, comme au TP01.

A rendre

Enregistrer et nommer votre pipeline prenom-nom-tp02.gra.

Envoyer votre pipeline à benoit.arbelot@gmail.com (titre du mail : [GICAO SIA] TP02)

Dans le mail, expliquer brièvement les étapes que vous avez effectuées et les problèmes rencontrés.