Thomas BARTHELEMY
Antoine LAROYE
Option RV – 2018/2019

18/02/2019

CONAV
-
RAY TRACING

Introduction

I. Créer une image

Le tutoriel explique assez clairement le code et la façon dont on crée une image avec le ray-tracing, mais n'explique pas clairement comment voir l'image que l'on peut obtenir. Sous Windows, le plus simple qu'on ait trouvé est d'inclure <fstream> et de remplacer les *cout* dans son code par la variable du fichier :

```
ofstream pic("pic.ppm");
pic << "P3\n" << nx << " " << ny << "\n255\n";
[…]
pic << ir << " " << ig << " " << ib << endl;
```

Ainsi on obtient un fichier *pic.ppm* dans le dossier du projet Visual Studio. La visionneuse Windows ne permet pas d'afficher l'image, mais on peut la faire glisser sur la page du site : http://paulcuth.me.uk/netpbm-viewer/ (ou avec Gimp ou Photoshop) pour en avoir un aperçu et vérifier que notre code fonctionne bien.

Sous Linux, on peut directement compiler le projet pour obtenir un fichier .ppm visualisable avec je sais plus quel logiciel.

Une fois que ceci est fait, on obtient l'équivalent du «hello world» en synthèse d'images :

II. La classe vec3

Ici, aucune difficulté majeure. En cours, on a déjà vu et utilisé cette classe un certain nombre de fois. Le code est un peu long à recopier (sur le pdf le code est sous forme d'image et on ne peut pas copier/coller, mais l'auteur du tutoriel donne le lien de son GitHub pour le cours, ce qui permet de récupérer les gros morceaux de codes comme celui-ci).

III. Rayons, caméra simple et background

De même que pour le chapitre précédent, cette partie traite de sujets que nous avons étudié en cours, et permet donc de mettre en pratique (via le code) la théorie que nous avons vue.

Ici pour créer le background, on crée une fonction color(ray r) qui affecte une couleur en fonction de la hauteur de la variable y du pixel pour avoir un background dégradé blanc et bleu. L'auteur du tutoriel explique de toute façon toujours ce que fait son code dans les paragraphes du chapitre en cours, donc on est rarement perdus. Le seul inconvénient conséquent de ceci est l'absence de commentaires dans son code, qui force à aller chercher l'information dans le texte (c'est un peu moins commode mais ça se fait).

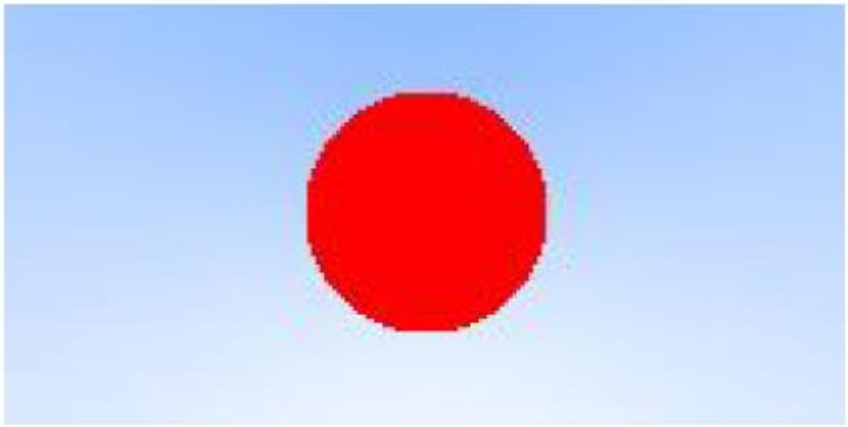
Avec la fonction color, appelée pour chaque fragment avec un rayon tracé entre l'origine de la caméra et le fragment concerné, on remplit donc l'image de ce fort joli bleu-ciel dégradé :

```
for (int j = ny - 1; j >= 0; j--)
{
	for (int i = 0; i < nx; i++)
	{
		float u = float(i) / float(nx);
		float v = float(j) / float(ny);

		ray r(origin, lower_left_corner + u * horizontal + v * vertical);

		vec3 col = color(r);

		[…]
	}
}
```

IV. Ajout d'une sphère

Pour bien comprendre cette partie, il faut bien comprendre les équations détaillées dans le tutoriel. Heureusement, l'équation d'un point sur une sphère est assez simple à paramétrer, et Peter Shirley détaille bien les calculs (avec même des schémas !!).

Donc, dans le code on crée une fonction booléenne qui vérifie si un rayon touche la sphère que l'on crée (fonction hit_sphere), et si c'est le cas on colorie le fragment en rouge pour ''afficher'' la sphère :



V. Normales de surfaces et objets multiples

Dans cette partie, on écrit d'abord un mini vertex-shader pour donner un peu de volume à notre sphère. On modifie donc la fonction hit_sphere pour qu'elle retourne non pas un booléen mais un flottant, dont la valeur déterminera la couleur du fragment dans la fonction color (si le rayon touche la sphère).

```
vec3 color(const ray& r)
{
	// Valeur de la racine solution de l'équation paramètre At + B = R
	float t = hit_sphere(vec3(0, 0, -1), 0.5, r);
	if (t > 0.0)
	{
		//Calcul de la normale au point touché par le rayon
		vec3 N = unit_vector(r.point_at_parameter(t) - vec3(0, 0, -1));

		//Attribution d'une couleur selon la normale (shader)
		return 0.5*vec3(N.x() + 1, N.y() + 1, N.z() + 1);
	}
	// Si le rayon ne touche pas, alors on garde la couleur du fond
	vec3 unit_direction = unit_vector(r.direction());
	float t = 0.5*(unit_direction.y() + 1.0);
	return(1.0 - t)*vec3(1.0, 1.0, 1.0) + t * vec3(0.5, 0.7, 1.0);

}
```

On obtient ceci :

La théorie de cette partie est assez simple, mais le code associé n'est pas toujours très clair… En fait, la ligne où la couleur est définie correspond juste à rendre les composantes de la normale (entre -1 et 1) entre 0 et 1 (d'où l'ajout de 1 et la multiplication par ½ pour les composantes de la normale) afin les ajouter dans un vecteur color.

Si on joue un peu avec ces paramètres, par exemple en augmentant la valeur ajoutée aux composantes :

Vec3(
N.x() + 1,
(N.y() +1)**0.3**,
N.z() +1) ;

On réduit la composante *green* des couleurs : la sphère est plus violette (rouge+bleu)

Vec3(
N.x() + **1.4**,
N.y() +1,
N.z() +1) ;

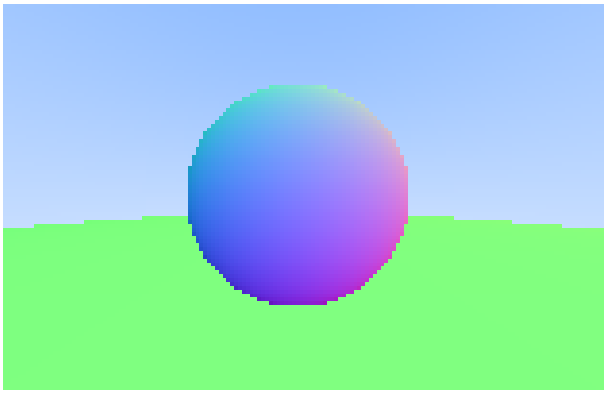
On sature la composante *rouge (x)* de la sphère. L'axe x étant dirigé vers la droite, cette modification apparaît sur la droite de la sphère (vers le haut pour y et vers la caméra pour z)

Ensuite, le tuto nous amène à faire afficher 2 sphères. Pour faire cela, on crée :

- une classe *hitable* d'objets qui peuvent être touchés par un rayon
- une classe *hitable_list* (hérite e *hitable*) qui liste les objets touchés
- une classe par forme, qui redéfinit la fonction *hit* selon l'équation paramétrique de sa surface

Dans cette partie, l'auteur utilise des pointeurs de pointeurs (**list) ainsi que des déréférencements avec l'accesseur ->… Le code est ici plus compliqué que la théorie ! On comprend ce que l'on fait, mais on comprend pas vraiment comment on le fait.

Par ailleurs, Peter Shirley fait appel à une variable MAXFLOAT qui est une variable déjà définie par le système, mais qui n'existe pas sous Windows… Avec Visual Studio, il faut mettre FLT_MAX. On obtient finalement 2 sphères (une centrale et une grande, le ''sol'').



VI. Anti-aliasing

Comme l'image créée est toute crénelée, on applique un anti-aliasing. Pour chaque pixel (associé à un couple (i,j) des boucles de la fonction main), on randomise un échantillonnage autour du centre du pixel, et pour chacun de ces échantillons, on stocke la couleur dans le vecteur associé. A la fin du sur-échantillonnage, on divise par le nombre d'échantillons pour avoir la moyenne.

Peter Shirley utilise la fonction drand48(), qui retourne un entier entre 0 et 1 (1 exclus). Sous Windows, nous avons choisi de faire comme dans le code à droite ➔

```
for (int s = 0; s < ns; s++)
{

	int intRU = rand() % 9999;
	float ru = float(intRU) / 10000;
	float u = float(i + ru) / float(nx);


	int intRV = rand() % 9999;
	float rv = float(intRV) / 10000;
	float v = float(j + rv) / float(ny);

	ray r = cam.get_ray(u, v);
	vec3 p = r.point_at_parameter(2.0);
	col += color(r, world);
}
```

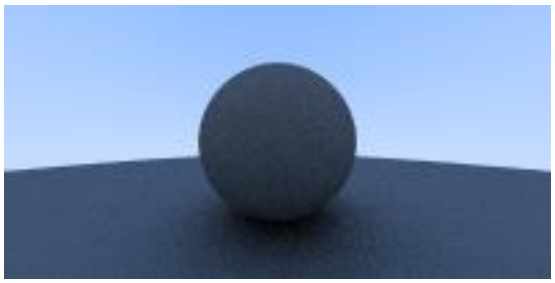
C'est pas très joli mais ça marche

VII. Matériaux mats

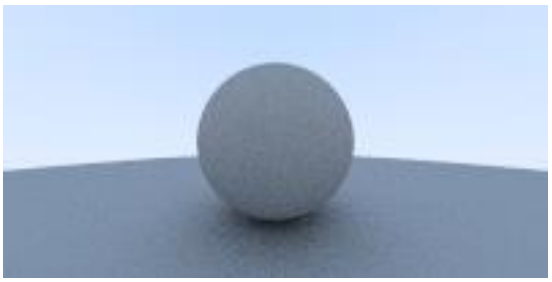
Ce chapitre permet d'ajouter un *material* à notre sphère. La théorie n'est pas aussi évidente que pour les autres chapitres, mais reste abordable… On a eu un gros obstacle lors de cette étape : après avoir lu et réécrit le code du tutoriel, on a voulu tester le programme mais une exception s'est levée ! Après de longs moments de debug, il nous a fallu comparer notre fichier cpp avec celui d'un autre groupe pour comprendre. L'erreur venait indirectement de notre façon (un peu moche) de calculer un nombre aléatoire entre 0 et 1. L'appel de la fonction srand(time(NULL)) dans la fonction color (qui est récursive dans cette étape) provoquait visiblement un problème de mémoire après un trop grand nombre d'itérations. Du coup, on l'a enlevé et on en a profité pour réécrire les parties avec *rand* de manière plus compacte :

```
vec3 random_in_unit_sphere()
{
	vec3 p;
	//srand(time(NULL));
	do {
		p = 2.0*vec3(double(rand()) / RAND_MAX, double(rand()) / RAND_MAX,
double(rand()) / RAND_MAX) - vec3(1, 1, 1);
	} while (p.squared_length() >= 1.0);
	return p;
}
```

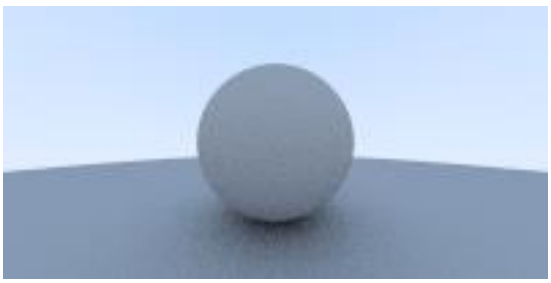
Ensuite, c'était tutto buono :



Avec la correction gamma :



Et la correction des ombres :



VIII. Métaux

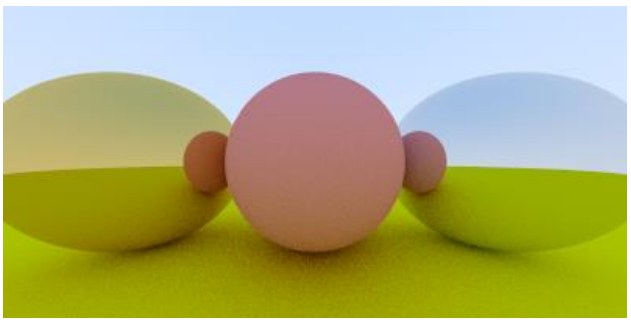
Pour créer les matériaux métalliques, on définit des sous-classes de *material* qui vont déterminer la texture de l'objet. On a donc *lambertian* (diffus) et *metal* (réflexion). Dans le main, on crée donc des sphères avec des propriétés différentes : il faut rajouter une variable qui pointe vers un *material* au constructeur de *sphere,* ainsi que l'indique Peter Shirley. Seulement, il ne dit pas quoi modifier et c'est ça la difficulté de cette partie. On a rajouté dans sphere.h :

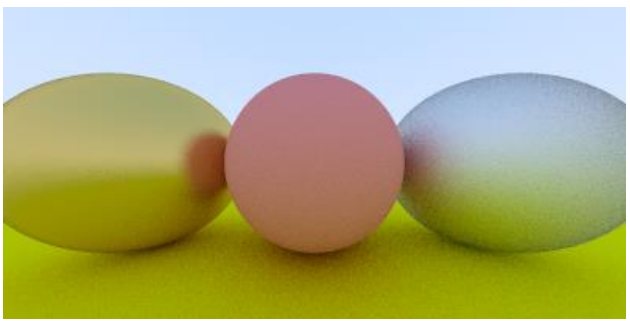
```
class sphere : public hitable
{
	public:
		sphere() {};
		sphere(vec3 cen, float r, material * mat) : center(cen), radius(r), mat_ptr(mat) {};
		virtual bool hit(const ray& r, float tmin, float tmax, hit_record& rec) const;
		vec3 center;
		float radius;
		material * mat_ptr;
};
```

Pour correspondre à l'appel du main :

```
list[1] = new sphere(vec3(0, -100.5, -1), 100, new lambertian(vec3(0.8,0.8,0.0)));
```

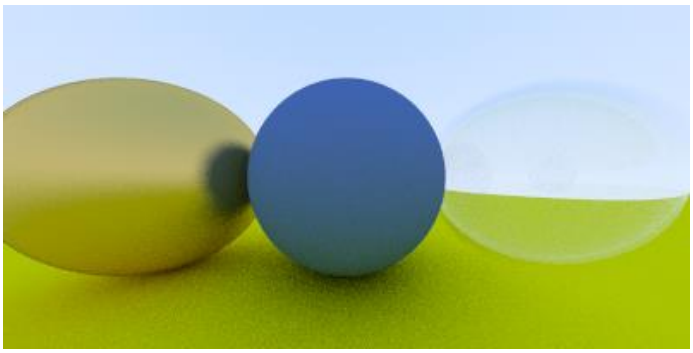
En faisant cela, on a eu une erreur de pointeur (qui ne pointait vers rien du tout). En effet, on avait oublié de rajouter dans la méthode *hit* l'affectation du pointeur du *hit_record rec*. Après cette correction, ça a marché comme sur des petites roulettes :





IX. Corps transparents (diélectriques)

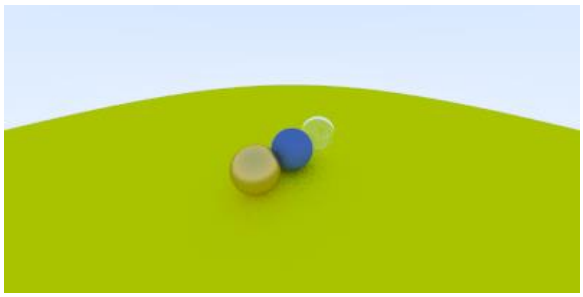
Pour cette partie sur la réfraction, pas de grande difficulté majeure si ce n'est la compréhension des calculs effectués dans le code. La plupart des classes sont déjà définies donc ces nouveaux matériaux que l'on ajoute sont juste des sous-classes avec des calculs spécifiques pour déterminer leur apparence. Ici, pour les matériaux diélectriques, on détermine l'éclatement d'un rayon et selon son angle, on sait s'il est réfracté ou réfléchi.



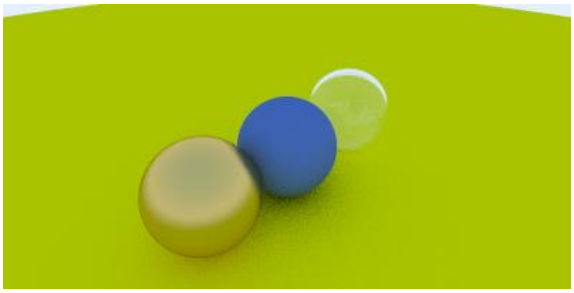
X. Positions et fov de caméra

Les 2 dernières parties sont assez simples de compréhension. On modifie les caractéristiques de la caméra pour déterminer le frustrum dans lequel on envoie nos rayons. On peut ainsi changer de FOV et de position pour la caméra. C'est un peu galère car on ne peut accéder à ces variables que dans le code, et on est obligés d'attendre d'avoir l'image totalement rendue avant de voir le résultat.

FOV = 90°



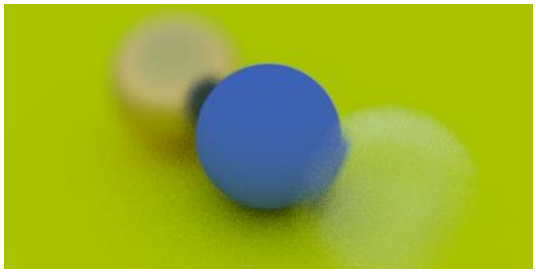
FOV = 45°



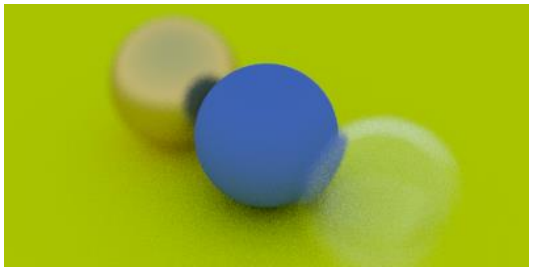
XI. Focus et profondeur de champ

Ici, pareil : pas grand-chose à dire car le tutoriel est assez simple et clair. On envoie les rayons depuis un disque unitaire au lieu d'un point (depuis la lentille quoi) et on définit une longueur de focus qui détermine le plan de netteté de l'image. Le fait que les rayons soient lancés depuis un disque rend flous les objets qui ne sont pas dans le focus de la caméra, et tadaaaaaa :

Ouverture = 2



Ouverture = 1

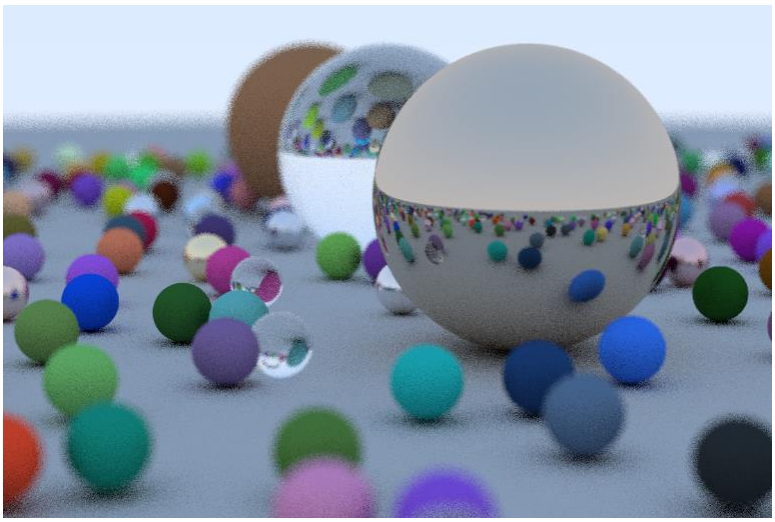


Conclusion

Ce TP est très intéressant, car si l'on voit le principe du ray-tracing en IMAGRV, on ne sait pas ce que ça donne en termes de code. Ici, on a un bon aperçu des fondamentaux de cette technique de rendu : les classes à créer, comment lancer les rayons, comment ils réagissent et pourquoi ils doivent réagir de cette façon pour que ça donne un certain résultat…

En tout, on a dû passer une douzaine d'heures dessus. C'est plutôt raisonnable et on avance assez simplement. Quand on bloque trop, on peut toujours comparer son code avec celui disponible sur le GitHub qui est fournit dans le pdf. Ce tutoriel est donc bien fourni en explications, mais n'est pas toujours très clair (surtout qu'il est en anglais, avec parfois des termes compliqués) et de temps à autres, on recopie le code sans trop comprendre pourquoi derrière l'image réagit de cette façon…

La dernière image du tuto (1200x800 px) générée aléatoirement :



C'est d'ailleurs là qu'on se rend compte que le RT temps réel, c'est compliqué avec des CGU ''classiques'' !