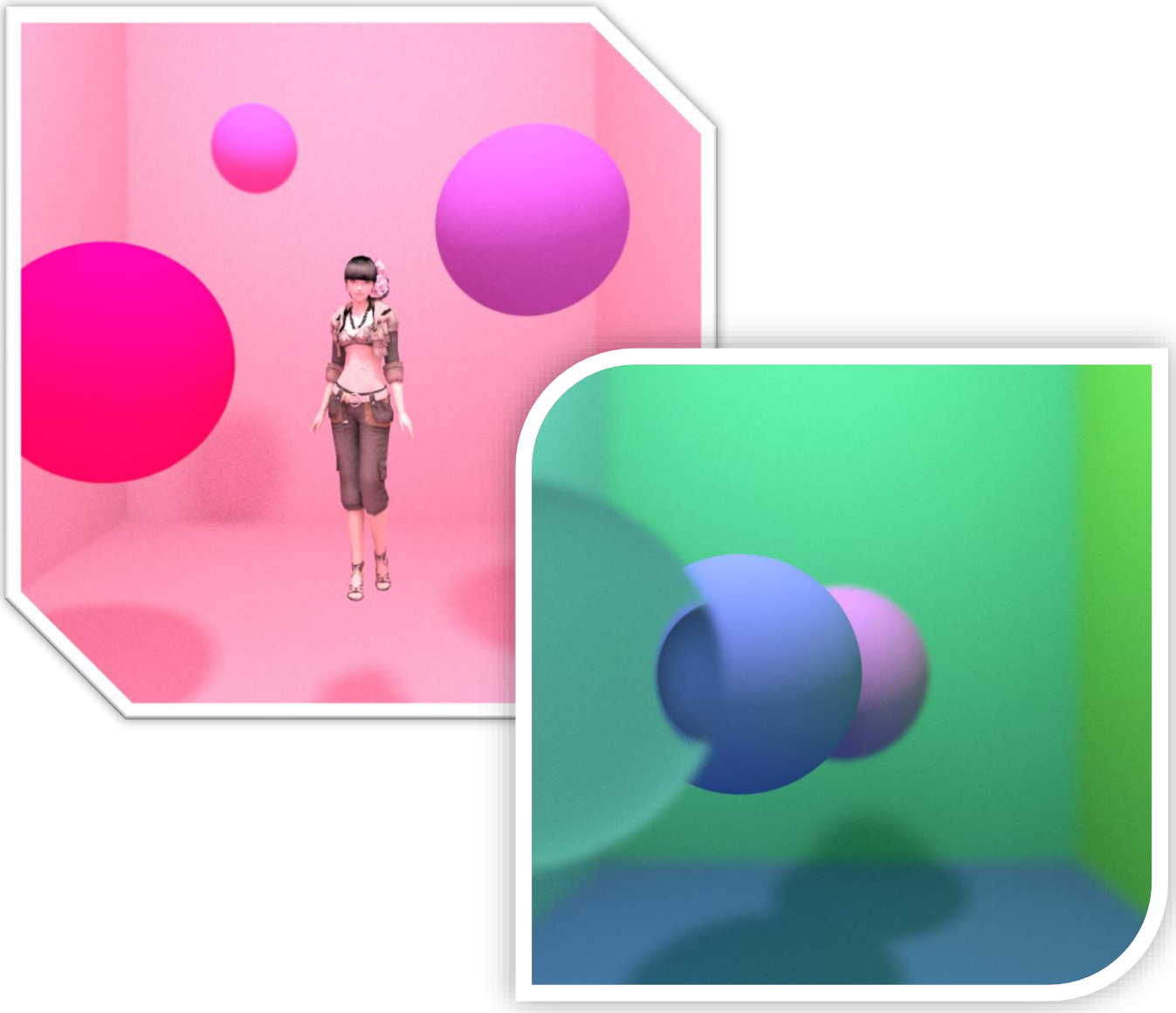
Philippe POEUNG

Ecole Centrale de Lyon

RAPPORT DE PROJET D'INFORMATIQUE GRAPHIQUE

Rendu réaliste par raytracing

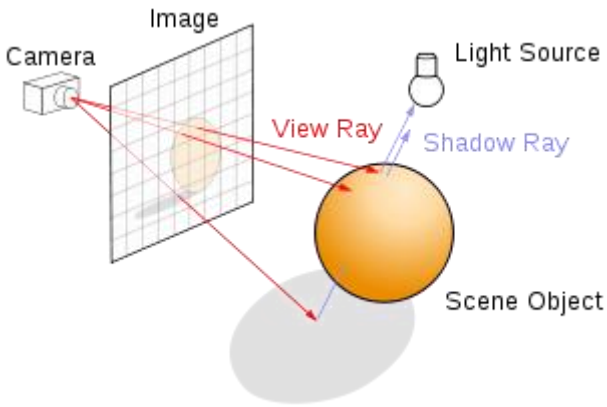


Semestre 9 2017-2018

Enseignant : Nicolas BONNEEL

Introduction

Nous allons réaliser un programme de rendu réaliste par méthode dite de raytracing backward. Elle consiste à tracer des rayons de lumière depuis la caméra en direction de l'écran de pixel, et la trajectoire (entre autres) de ces rayons va, selon des règles approchantes celles de la physique du monde réel, déterminer la couleur du pixel associé aux rayons.



Mise en place des éléments de base

On va écrire ce programme en C++.

Les 4 premiers objets que l'on va définir sont :

- Vector : correspond à un triple de double. On va l'utiliser aussi bien pour décrire un vecteur de l'espace 3D qu'un point de l'espace 3D ainsi qu'une couleur (coordonnées RGB)
- Rayon : correspond un point d'origine (Vector) ainsi qu'un rayon directeur (Vector de norme euclidienne unitaire)
- Sphère : correspond à un centre (Vector) ainsi qu'un rayon (double) ainsi qu'un albedo (Vector) correspondant à la fraction de chaque composante de la lumière RGB qui est renvoyée par la sphère
- Scène : correspond à une liste de sphères, ainsi qu'à la position d'une source lumineuse (Vector) ainsi qu'une intensité de la source lumineuse (double)

On définit également les méthodes et fonctions de base, par exemple le produit scalaire/vectoriel/de Hadamard de deux Vector, la normalisation d'un Vecteur, l'ajout d'une sphère dans la scène, etc.

On va également utiliser un timer pour connaître le temps d'exécution.

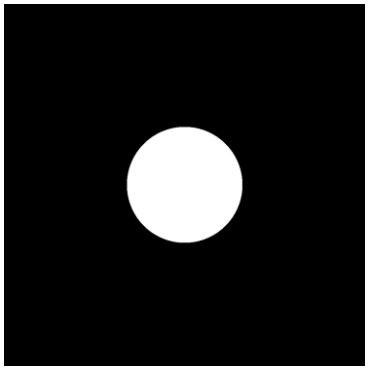
Calcul d'intersection rayon-sphère

La première routine à implémenter consiste à dire sur un rayon intersecte une sphère. cette fonction va renvoyer le point d'intersection P, la normale N en ce point ainsi que la distance t entre l'origine du rayon et P.

```
bool intersect(const Ray &ray, Vector &P, Vector &N, double &t) {
	double a = ray.u.getNorm2(); //est égal à 1 normalement
	double b = 2 * prodSca(ray.u, ray.C - O);
	double c = (ray.C - O).getNorm2() - r*r;
	double delta = b*b - 4 * a*c;
	if (delta < 0)
		return false;
	double t1 = (-b - sqrt(delta)) / (2 * a);
	double t2 = (-b + sqrt(delta)) / (2 * a);
	if (t1 > 0)
		t = t1;
	else if (t2>0)
		t = t2;
	else return false;
	P = ray.C + t*ray.u;
	N = getNormalized(P - O);
	return true;
};
```

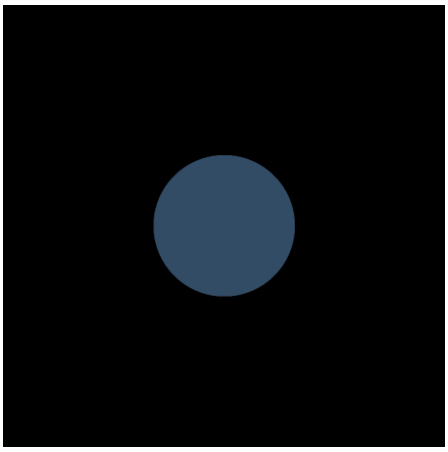
On va ensuite créer une fonction d'intersection mais à l'échelle de la scène : l'intersection réelle sera pour la première sphère intersectée (celle dont le facteur t est le plus petit). Cette fonction renverra également l'indice de cette sphère intersectée.

Dans le cas d'une intersection, on renvoit un pixel blanc. Sinon, on renvoie un pixel noir.On va ensuite la teste en mettant une unique sphère face à la caméra.



Temps d'éxécution : ~1 sec

On peut aussi légèrement modifier le programme pour que couleur du pixel dépende de l'albédo de la sphère ainsi que l'intensité de la scène :



Temps d'éxécution : ~1 sec

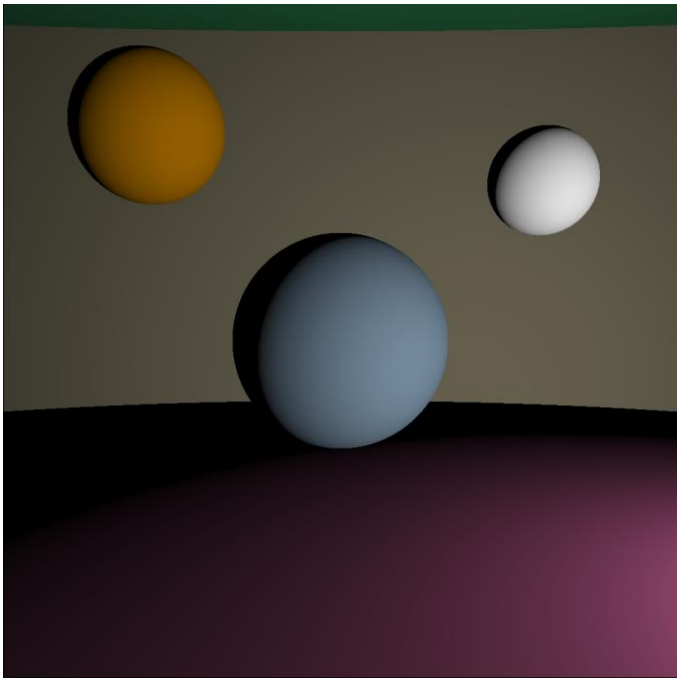
Modèle d'éclairage lambertien

On va maintenant dans le cas où il y a intersection, chercher une couleur plus réaliste. Pour cela, on va utiliser la loi du cosinus de Lambert (https://fr.wikipedia.org/wiki/Loi_de_Lambert) et faire dépendre la couleur d'un point selon l'albédo du point de la surface, l'intensité de la lumière, le carré de la distance entre le point et la lumière, ainsi que du cosinus entre la normale au point et la direction du point vers la lumière :

```
Ipix = I*spheres[numSph].rho/3.1415926
*std::max(0.,prodSca(getNormalized(sourceLum - P), N))
/(sourceLum - P).getNorm2()
```

On va emballer tout ça dans une fonction *getColor*, appelé pour chaque pixel dans la fonction main().

On va également mettre des sphères de grands rayons à grand distance qui vont représenter des murs de part et droite de la scène. On obtient :

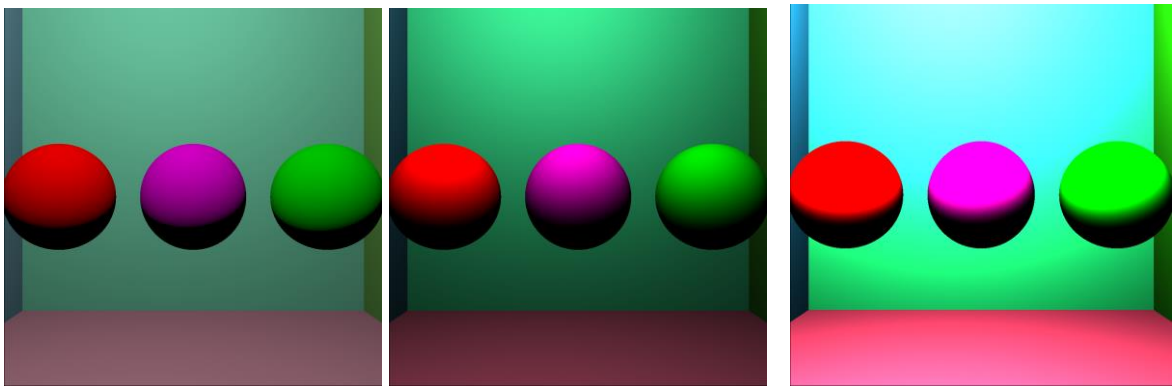


Temps d'éxécution : ~1 sec

On voit qu'il y a un dégradé de couleur depuis la surface la plus exposées à la source lumineuse jusqu'à la la latitude de la sphère qui n'est plus exposée du tout à la source lumineuse et apparaît en fait totalement noire.

Correction gamma

Nous allons maintenant implémenter la correction gamma. Il s'agit d'élever la couleur du pixel à une certaine puissance pour augmenter (>1) ou réduire (<1) le contraste de l'image.



Gamma = 0.45 // Gamma 1 // Gamma = 1.5

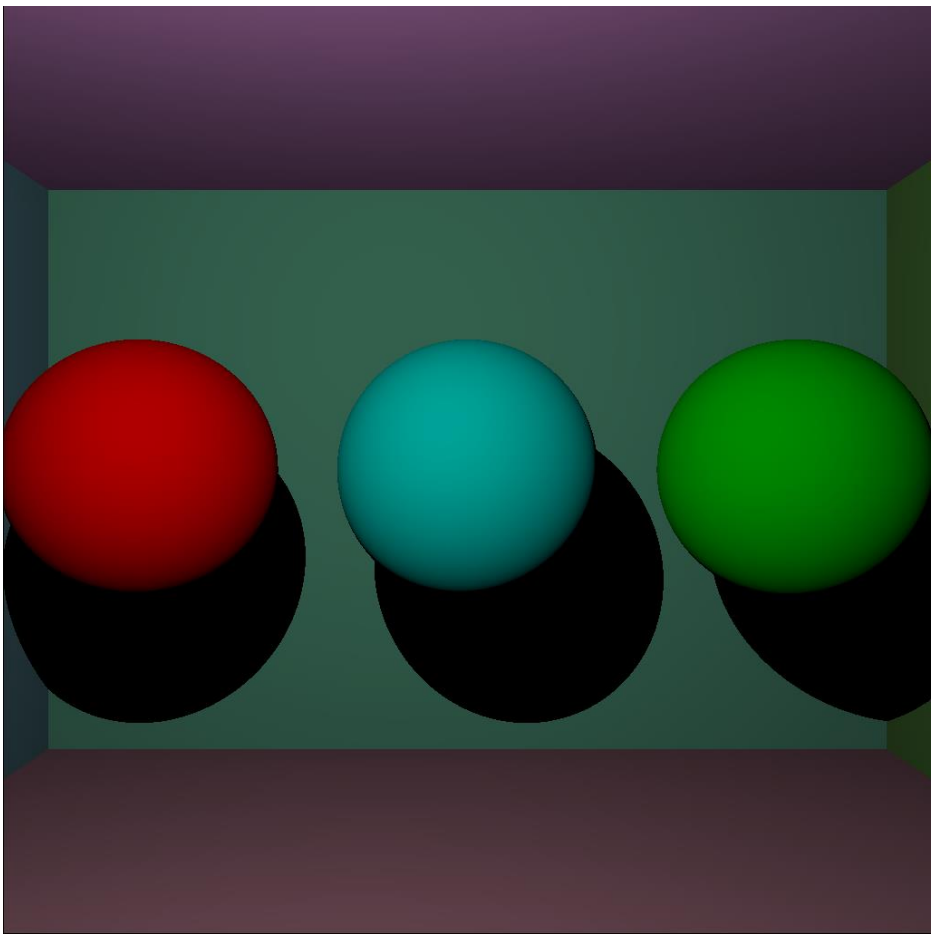
Nous allons dès lors utiliser un gamma de 0.45 pour obtenir des couleurs moins "criardes".

Ombres portés

Nous allons faire en sorte que les points qui sont cachés de la lumière apparaissent en noir, et ce en modifiant la fonction *getColor*. Pour cela, on va effectuer un deuxième test d'intersection, cette fois ci entre le point P pour lequel on cherche à savoir si il est caché par une sphère et la position de la lumière. Il n'y ombre portée que si ce deuxième test d'intersection donne l'existence d'une intersection ET si cette intersection se trouve entre le point P et la source de lumière (et non derrière), donc que la distance t est inférieure à la distance entre P et la source de lumière. Dans ce cas, on renvoie un pixel noir.

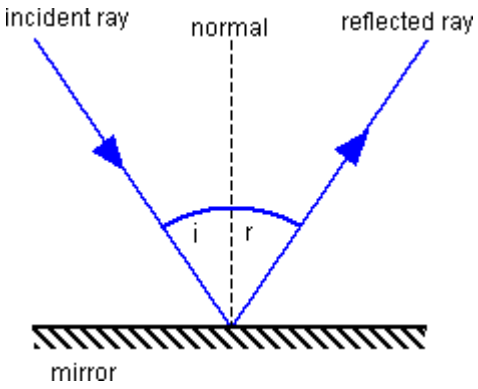
```
Vector Ipix = Vector(0, 0, 0);
bool CS = intersect(Ray(P + epsilonCS*N, sourceLum - P), PCS, NCS,
numSphCS, tCS);
if (!CS || (CS && tCS > std::sqrt((sourceLum - P).getNorm2()))) {
      Ipix = //éclairage selon modèle lambertien
}
```

On obtient le résultat escompté :



Temps d'exécution : ~1.5 sec

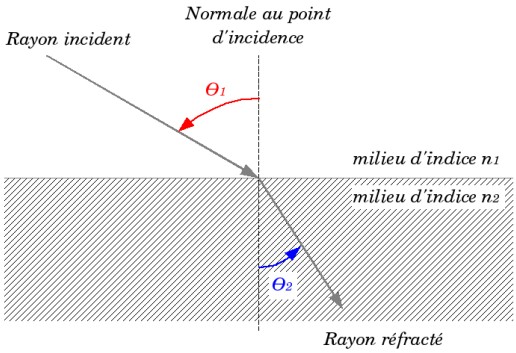
Sphères miroirs



On va introduire des sphères ayant le comportement d'un miroir. D'après la loi de Snell-Descartes pour le rayon réfléchi, la couleur d'un point P de cette sphère va être celle du rayon réfléchi par la sphère à angle égal.

On va donc dans ce cas rendre la fonction getColor récursive, en veillant à limiter le nombre de récursion (par exemple dans le cas de deux points de miroir parfaitement face à face).

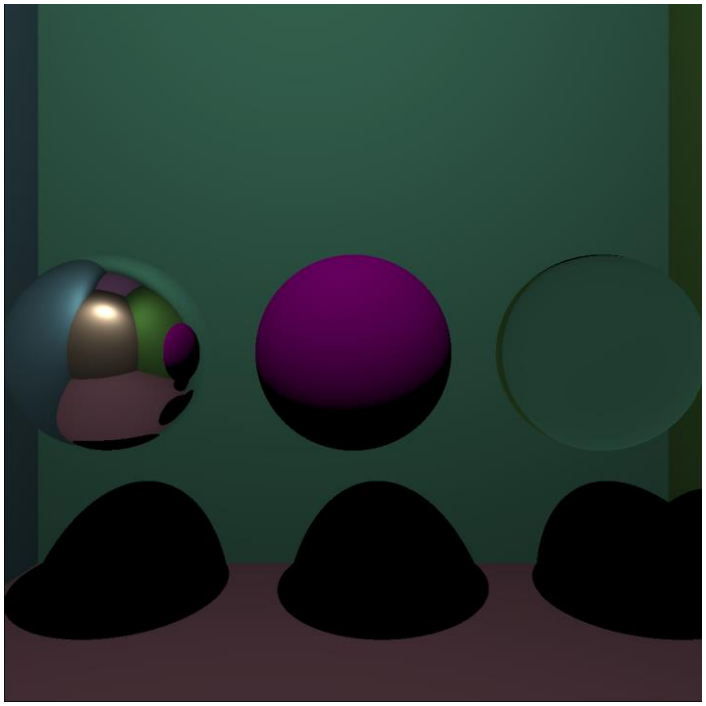
Sphères transparentes



Cette fois-ci, on va utiliser la loi de Snell-Descartes pour le rayon réfracté.

$$n_1 \cdot \sin(\theta_1) = n_2 \cdot \sin(\theta_2)$$

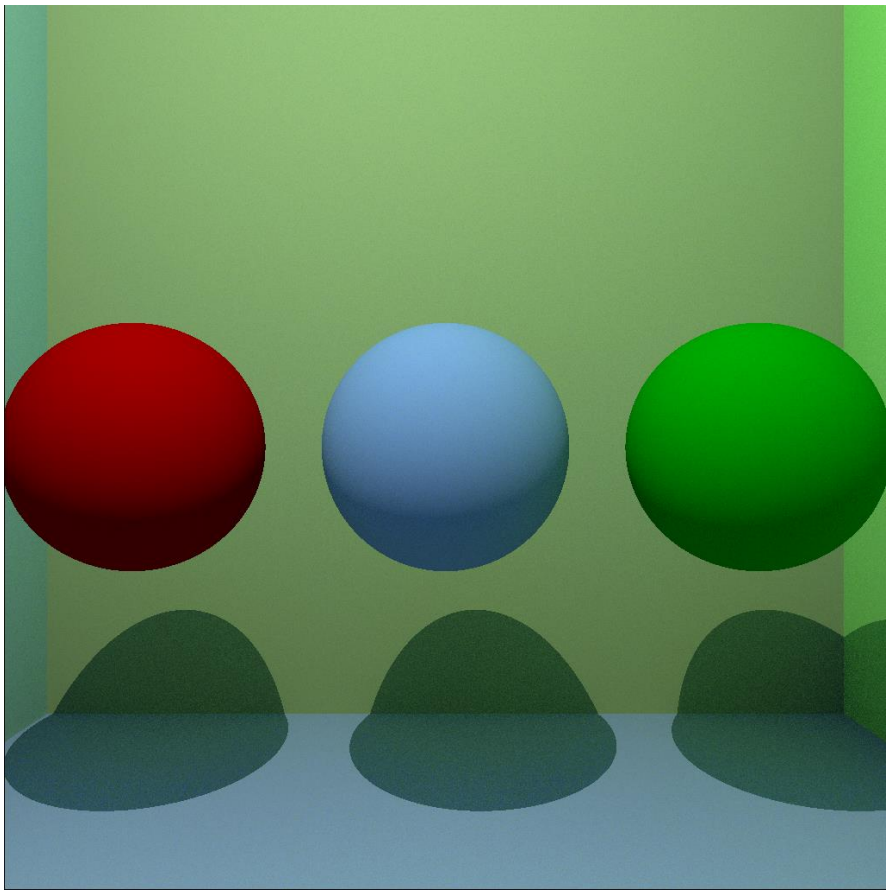
Voici le résultat de l'implémentation de ces deux sphères :



Temps d'exécution : ~1.5 sec

Eclairage indirect

On va maintenant utiliser le principe physique signifiant que chaque objet éclairé devient une source secondaire de lumière. On va ainsi pour chaque point dire que sa couleur se divise en 2 parties : un éclairage direct (si pas dans une ombre portée) ainsi qu'un éclairage indirect, généré selon une unique trajectoire aléatoire. On obtient l'image suivante avec 50 rayons générés et 10 rebonds :



Temps d'exécution : ~8 min

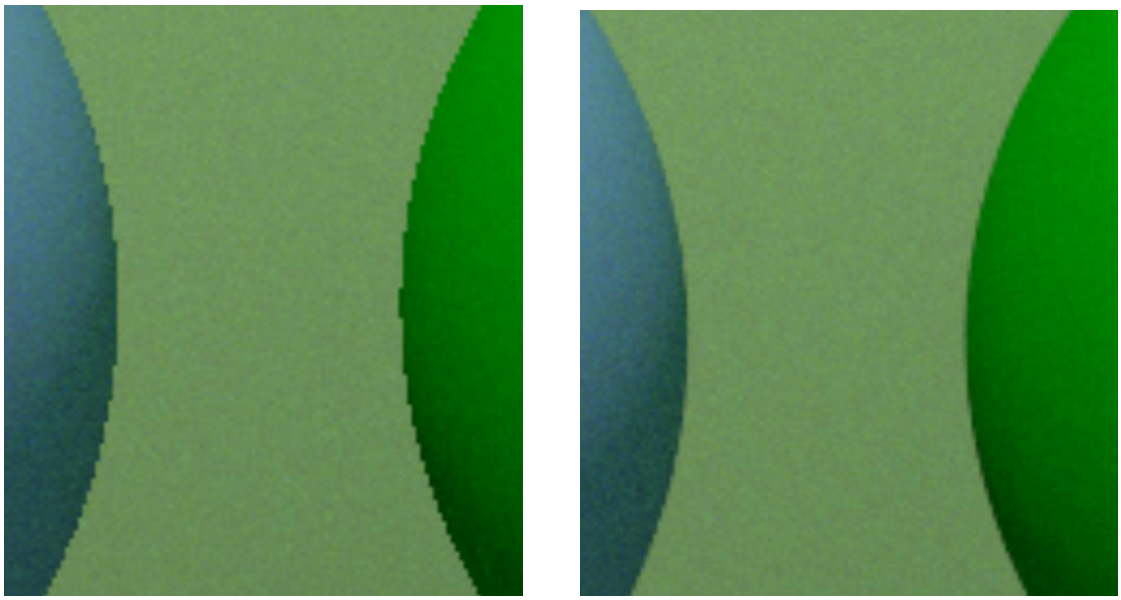
On observe que les zones d'ombres portées ne sont plus totalement noires, car elles reçoivent de l'éclairage indirect.

Anti-aliasing (anti-crénelage)

On va implémenter l'anti-aliasing pour éviter ce genre d'effet digracieux. Ainsi, chaque rayon tiré depuis la caméra ne va pas se diriger exactement vers le centre du pixel, mais plutôt selon une gaussienne centrée sur le centre du pixel.

```
double u1 = u(engine);
double u2 = u(engine);
double rx = sqrt(-2 * log(u1))*cos(2 * pi*u2)*0.25;
double ry = sqrt(-2 * log(u1))*sin(2 * pi*u1)*0.25;
Ray ray = Ray(eye, getNormalized(Vector(i - W / 2 + rx, j - W / 2 + ry, -W / 2 / tan(fov / 2))));
```

On obtient une image avec moins de crénelage :



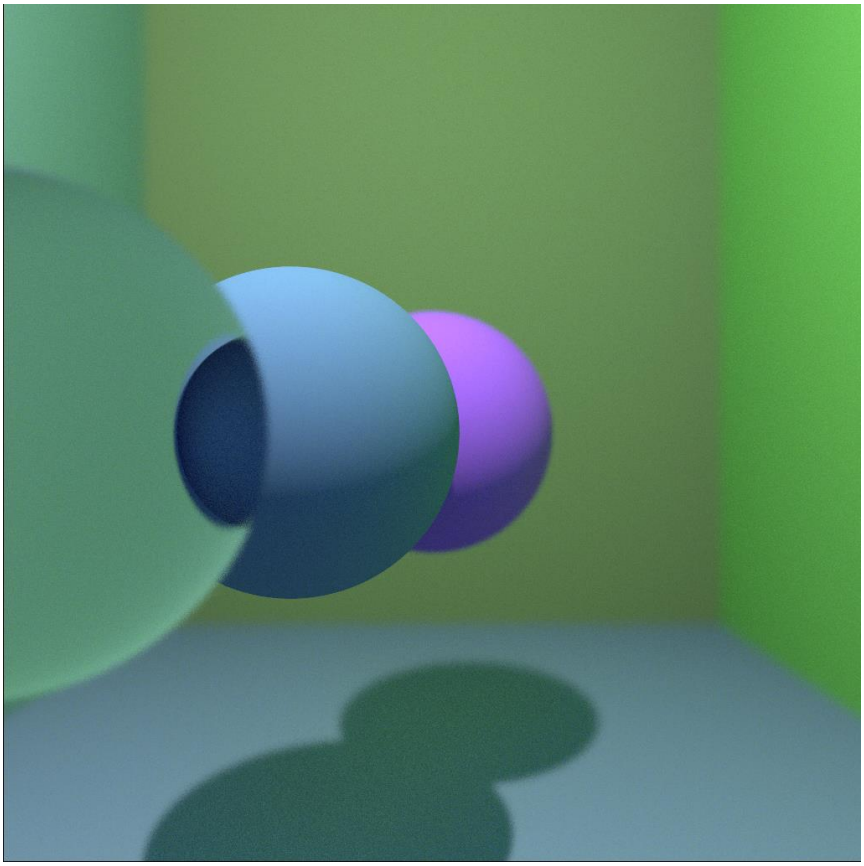
Sans anti-aliasing // Avec anti-aliasing

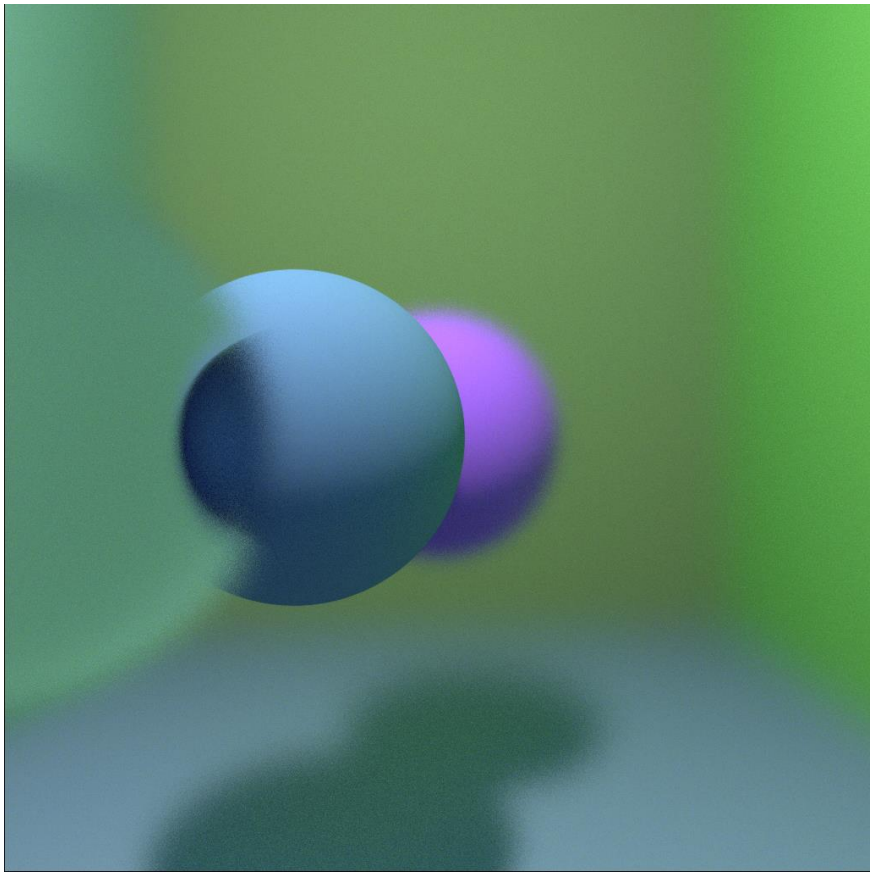
Depth-of-Field (distance de focus)

Une caméra va admettre un plan (sphère en réalité) focal telle que les objets dans ce plan focal vont être nets. Les objets plus loin ou plus proches vont apparaître de plus en plus flou qu'ils en sont éloignés. Pour implémenter cela, on créer une première direction, comme précédemment, qui va définir une destination selon la distance de focus choisie pour la caméra. On va ensuite choisir aléatoirement un point légèrement horizontalement ou verticalement décalé par rapport à la caméra (l'écart-type de ce décalage représente l'ouverture de la caméra). Ces deux points vont définir le rayon qui va nous servir à calculer pour un essai la couleur du pixel.

```
double u3 = u(engine);
double u4 = u(engine);
double rxo = sqrt(-2 * log(u3))*cos(2 * pi*u4)*obstru;
double ryo = sqrt(-2 * log(u3))*sin(2 * pi*u4)*obstru;
Vector direction = getNormalized(Vector(j - W / 2 + ry, i - W / 2 + rx,
z)); // c'est à la direction (corrigée par l'A-A)
Vector destination = eye + focus_distance*direction;
Vector eye_decale = eye + Vector(ryo, rxo, 0); // c'est l'oeil de la
caméra modifée pour le depth of field
Ray ray = Ray(eye_decale, getNormalized(destination - eye_decale));
```

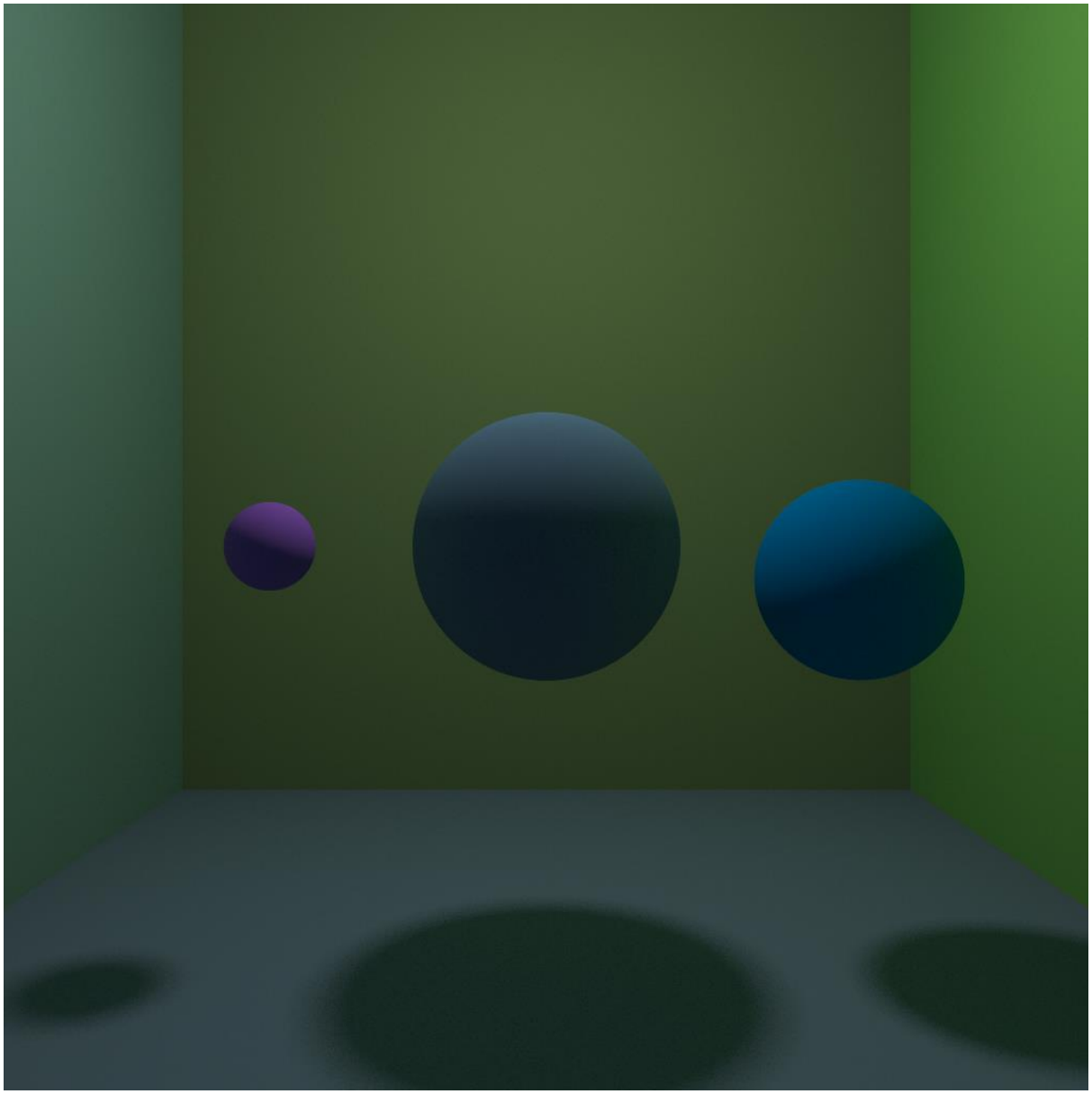
On obtient pour des ouvertures différentes les images suivantes (d'ouverture croissante respective 0.6 et 2.5) :





Source non ponctuelle

On va supprimer la source de lumière et la remplacer par une source lumineuse (qui sera la sphère n°0 de la scène). L'éclairage direct se fera désormais par échantillonnage (un seul échantillon/point) de la surface de l'hémisphère de la sphère lumineuse dirigée vers le point.



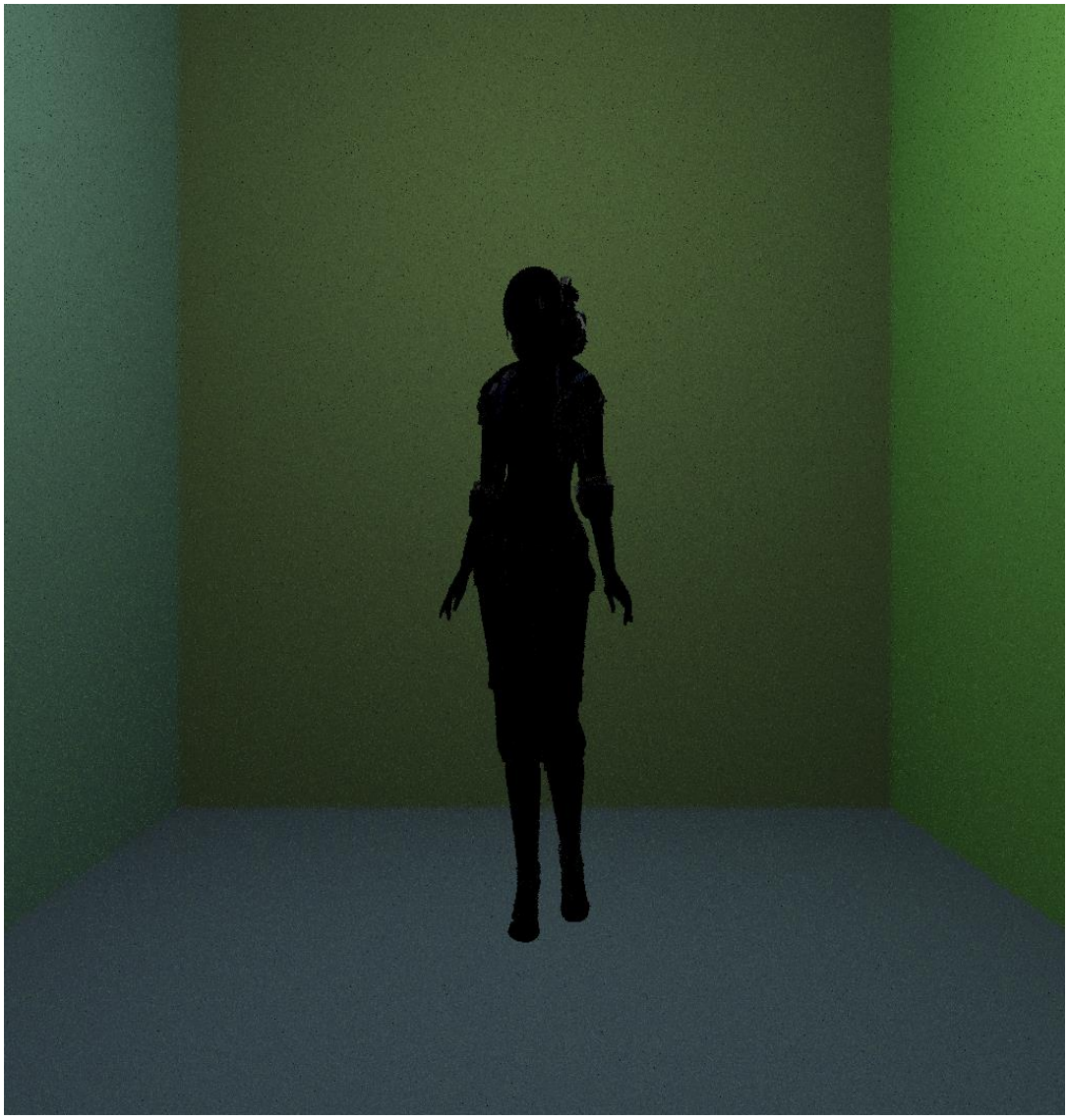
Temps d'exécution : ~8 min (50 rayons, 10 rebonds)

On voit que les ombres propres et les ombres portées n'ont pas une frontière aussi nette qu'avant, et il y désormais une zone de pénombre (zone éclairée uniquement par une partie de la source étendue)

Objets géométriques

On va rendre possible l'insertion d'objets (Geometry) qui sont des ensembles de sommets (vertices) formant des triangles.

Après avoir implémenté les routines d'intersection rayon-plan puis rayon-triangle, on peut rassembler triangle et cercle dans une même classe mère, et définir une routine d'intersection pour elle. On peut ensuite également définir une rayon-objet, qui consiste simplement à parcourir l'ensemble des triangles de l'objet et regarder si cette intersection existe. Ceci n'est pas optimisé, et même le programme avec uniquement le test d'intersection prend beaucoup de temps vu le nombre de triangle à tester. L'image ci dessous a donc été obtenu pour la routine avec BVH.



Temps d'exécution : ~6 min (5 rayons, 3 rebonds)

Textures

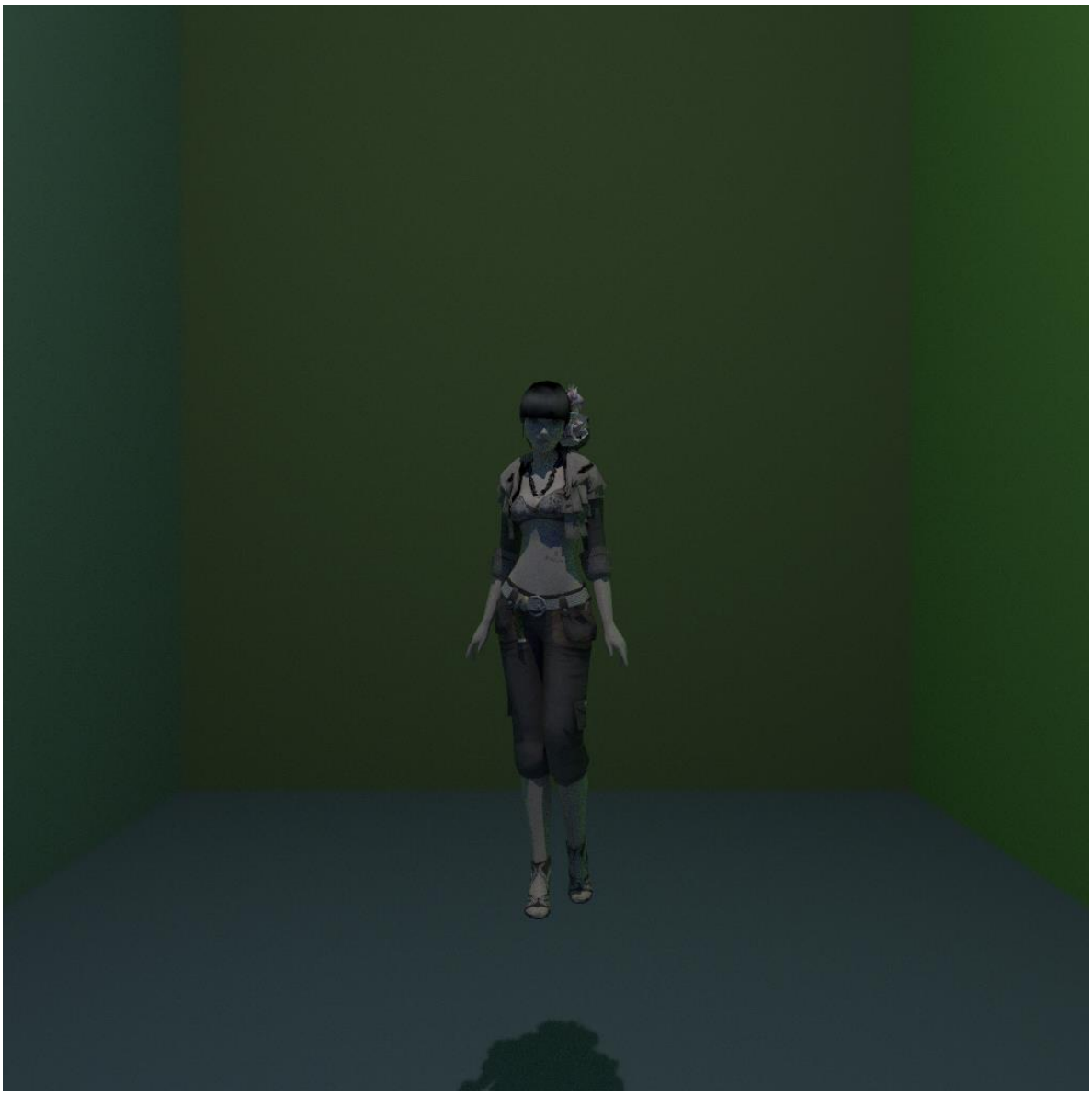
Pour la routine d'intersection rayon-objet, on peut optimiser un peu en ne calculant la normale, la position UV et la couleur qu'à la fin de la boucle, en stockant dans la boucle uniquement i_min, alpha_min, beta_min et gamma_min.

```
if (local_has_inter) {
	has_inter = true;
	if (t < tmin) {
		tmin = t;
		imin = i;
		P = localP;
		alphamin = alpha;
		betamin = beta;
		gammamin = gamma;
	}
```

puis

```
if (has_inter) {
	N = alphamin*normals[indices[imin].ni] +
betamin*normals[indices[imin].nj] + gammamin*normals[indices[imin].nk];
	int texID = indices[imin].faceGroup;
	Vector UV = alphamin*uvs[indices[imin].uvi] +
betamin*uvs[indices[imin].uvj] + gammamin*uvs[indices[imin].uvk];
	int x = UV[0] * (w[texID] - 1);
	int y = UV[1] * (h[texID] - 1);
	double rr = textures[texID][(y*w[texID] + x) * 3] / 255.;
	double bb = textures[texID][(y*w[texID] + x) * 3 + 1] / 255.;
	double gg = textures[texID][(y*w[texID] + x) * 3 + 2] / 255.;
	color = Vector(rr, bb, gg);
	tt = tmin;
}
return has_inter;
```

Voici le résultat avec textures :



Temps d'exécution : ~15 min (20 rayons, 5 rebonds)