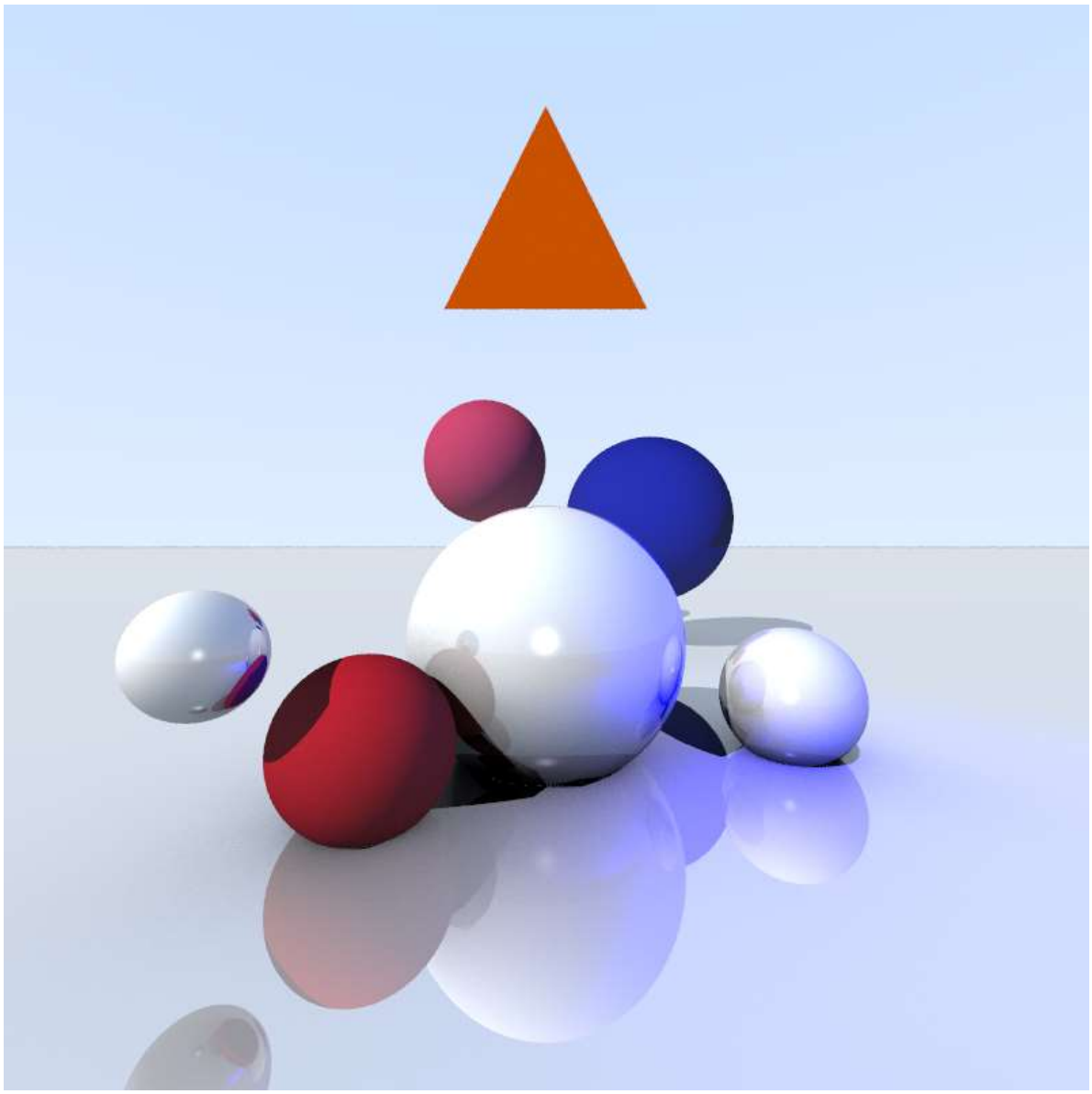
Raytracing

Par Margot Mangenot et Arthur Jacquin



Ce rapport a pour but d'expliquer les différents algorithmes et équations développés ainsi que l'architecture de notre programme.

Sommaire :

I) Lancé de rayons

a. Algorithme et traçage de scène

L'algorithme principale du raytracing se situe dans le fichier Raytracer.cpp.

Pour chaque centre des pixels de l'image, un premier rayon est envoyé depuis la position de la caméra jusqu'à la scène et renvoie la couleur de l'objet qu'il a frappé. Si aucun objet se trouve sur la trajectoire du rayon, il renvoie la couleur du ciel.

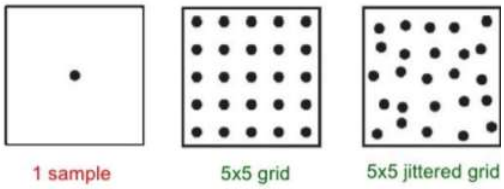
Le rayon est une classe ray.h avec une origine et une direction ainsi qu'une fonction evaluate() qui retourne la distance entre le point d'origine du rayon et le point d'intersection avec un objet de la scène.

b. Antialiasing super-sampling et jittering

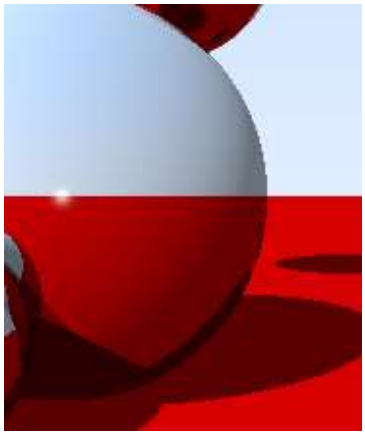
Pour atténuer cet effet de crénelage des pixels dû à l'envoie d'un seul rayon par pixel, nous avons utilisé la technique du super sampling.

Pour chaque pixel de l'image, plusieurs rayons sont uniformément envoyés et la couleur du pixel est calculé par la moyenne des couleurs des rayons retournés.

Pour aller plus loin, nous avons ajouté la technique du jittering. Les rayons par pixel ne sont plus envoyés uniformément mais



Le nombre de rayons par pixels peut être choisis par l'utilisateur.



Rendu sans AA sur une image en 800*800

Rendu avec 4 rayons par pixels

Rendu avec 8 rayons par pixel

```
//Raytracing
for (j = 0; j < height; j++)
{
    pixels.resize(width);

    for (i = 0; i < width; i++)
    {
        vec3 col = vec3{ 0.f, 0.f, 0.f };
        //Generation du rayon primaire
        ray ray;
        ray.origin = { 0.f, 0.f, 0.f };

        for (int sample = 0; sample < numberOfSamples; ++sample)
        {
            //Supersampling with jittering
            float x = aspectRatio * 2.f * (float((i + randomFloat.random_float()) - (width / 2)) + 0.5f) / width;
            float y = -2.f * (float((j + randomFloat.random_float()) - (height / 2)) + 0.5f) / height;

            ray.direction = { x, y, 1.f };
            ray.direction.normalize();

            col += vec3{ tracer.trace(ray) };
        }

        // Get the average
        col = col / numberOfSamples;
```

Boucle principale du raytracing avec anti-aliasing

II) Objets

Les scènes sont composées de plusieurs types d'objets : des primitives et des lights. Tous les objets présents dans la scène héritent de la classe Entity leurs fournissant une position et permettant de les manipuler facilement.

a. Primitives

La classe Primitive définit la structure qui sera appliqué sur chacune des primitives quelle qu'elle soit. Chaque primitive possède donc :

- Un matériau : définissant la couleur et le comportement de l'objet lors du shading
- Un fonction Intersect() : permettent de calculer un point d'intersection entre la primitive et un rayon, et retourne la distance entre le point d'intersection et l'origine du rayon.
- Un fonction calculateNormal() : qui retourne la normal en un point de sa surface.
- Une fonction CalculateUVs() : qui retourne les UVs en un point de sa surface.

Ce qui nous donne la classe suivante :

```cpp
class Primitive : public Entity
{
protected:
    const Material* material;

public:
    Primitive(){}
    Primitive(vec3 pos, const Material* mat) : Entity(pos), material(mat){}

    virtual float intersect(const ray& ray) const = 0;
    virtual vec3 calculateNormal(vec3& p) const = 0;
    virtual vec3 calculateUVs(vec3& p) const = 0;
    virtual ~Primitive() {}

    const Material* getMaterial() const { return material; }
};
```

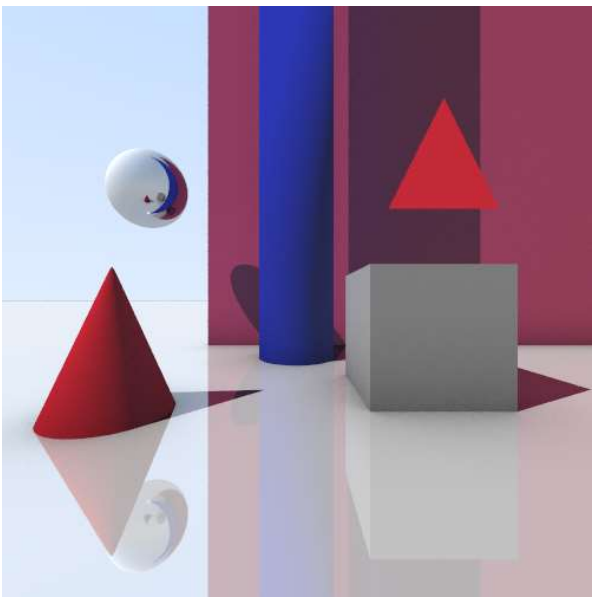
Dans notre raytracer, il est possible de faire 7 types de primitives :

- Plan
- Sphère
- Cube
- Triangle
- Carré
- Cylindre infini
- Cône

Chacune de ces primitives redéfinissent les fonctions de la classe Primitive pour les faire correspondre à ses propres caractéristiques. Voici un exemple avec la classe sphère qui possède une caractéristique supplémentaire, un radius.

```cpp
class Sphere : public Primitive
{
    float radius;

public:
    Sphere();
    Sphere(const vec3& pos, float r, Material* mat);

    float intersect(const ray& ray) const;
    virtual vec3 calculateNormal(vec3& p) const final;
    virtual vec3 calculateUVs(vec3& p) const final;
    virtual ~Sphere() {}
};
```

Voici un rendu montrant toutes les primitives :



b. Lights

La classe Light définit toutes les propriétés et fonctions que possède toutes les lights. Chaque light possède les propriétés suivantes :

- Couleur : couleur RGB
- Intensité : force de l'éclairage
- Portée : Distance jusqu'à laquelle la light à un effet

Elles possède également d'une fonction CalculateLighting() qui permet de calculer la transformation apporté par cette lumière en un point donnée.

Voici donc à quoi ressemble cette classe :

```
class Light : public Entity
{
protected:
    vec3 color;
    float intensity;
    float range;

public:
    Light(vec3 pos, vec3 col, float i, float range): Entity(pos), color(col), intensity(i), range(range) {}

    virtual vec3 CalculateLighting(const vec3& normal, const ray& ray, float _Glossiness, vec3 Color, const vec3& pos = vec3()) const = 0;

    virtual vec3 getDirection(const vec3& pos = vec3()) const = 0;
    vec3 getColor() const { return color; }
    float getIntensity() const { return intensity; }
    float getRange() const { return range; }

};
```

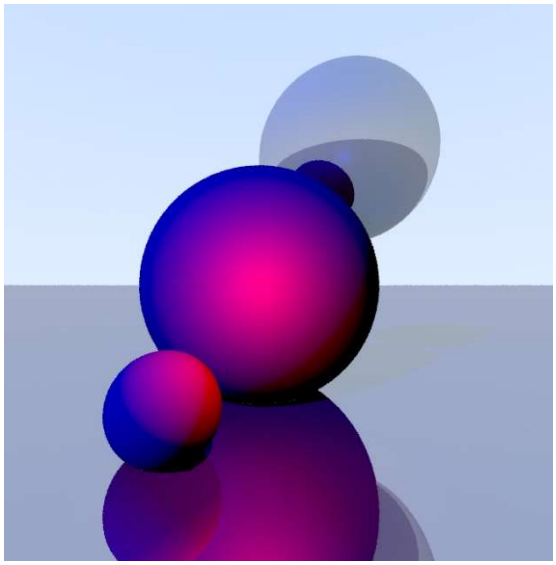
Notre raytracer implémente 2 types de lights :

- Les directional lights
- Les Point lights

Les directional lights, émettent de la lumière dans une direction avec une intensité qui ne diminue pas avec la distance. Leur position n'a donc aucun impact et leur portée est infinie.

Les point lights sont positionnées dans la scène et éclairent tout autour d'elles. Plus on s'éloigne de leur position, plus leur intensité diminue.

Voici un exemple de rendu avec : une directional light bleue allant de gauche à droite, et une point light rouge positionnée juste devant la plus grosse sphère :



III) Shading

a. Ombres

Les ombres sont calculées en suivant une logique de hard shadows, un fragment est donc soit complétement dans l'ombre (noir), soit en dehors de l'ombre (blanc). Pour déterminer si le fragment en question est dans l'ombre ou non, nous relançons un dans la direction de chacune des lights présentent dans la scène. Si ce rayon touche un autre objet, c'est que cet objet est devant la source de lumière et donc que nous sommes dans l'ombre.

Dans le cas des point lights, si la distance entre la position du fragment et la light est supérieure à la portée de la light le fragment est alors dans l'ombre.

Voici la représentation de tout cela dans le code :

```
float shadow = 1.f;
bool isInShadow = true;

for (int i = 0; i < lights.size(); i++)
{
    ray feeler;
    feeler.origin = position + normal * EPSILON;
    feeler.direction = lights[i]->getDirection(position) * -1.f;

    if (!inShadow(feeler) &&
        position.distance(lights[i]->getPosition()) < lights[i]->getRange())
    {
        isInShadow = false;
        break;
    }
}
```

b. Materiaux

Les matériaux sont calculés grâce à une classe Material.h dans laquelle on trouve un énumérateur permettant de classer les matériaux en quatre types : matte, plastic, metallic et dielectric. Les matériaux matte ne reflète aucune réflexion. Les metallic sont des miroirs (voir plus loin) et plastic reflète avec du fresnel. Les dielectrics sont réfractifs. Les deux derniers matériaux cités ne sont pas totalement implémentés.

Le modèle de shading utilisé est celui de Blinn – Phong et est calculé pour chaque light de la scène.

```
vec3 DirectionLight::CalculateLighting(const vec3& normal, const ray& ray, float _Glossiness, vec3 Color, const vec3& pos) const
{
    BRDFs brdf;

    //Ambiant
    vec3 ambient = vec3{ 0.0f, 0.0f, 0.0f };

    //Diffuse
    float diffuseFactor = brdf.clamp(std::max(0.f, normal.dot(getDirection(pos) * -1)), 0.f, 1.f);
    vec3 diffuseColor = Color * getColor() * getIntensity() * diffuseFactor;

    //Specular
    vec3 R = brdf.reflect(ray.direction, normal);
    float specularFactor = brdf.clamp(std::pow(std::max(0.f, R.dot(ray.direction * -1)), _Glossiness), 0.f, 1.0f);
    vec3 specularColor = vec3{ 1.f, 1.f, 1.f } *specularFactor * getColor() * getIntensity();

    Color = ambient + diffuseColor;

    if (_Glossiness > 0)
        Color = ambient + Color * diffuseColor + Color * specularColor;
    return Color;
}
```

L'utilisateur peut choisir également entre afficher un material avec une couleur unie ou avec deux couleurs sous forme de ligne (la couleur choisie au départ et du noir) voire même afficher le material avec une texture qu'il aura choisie.

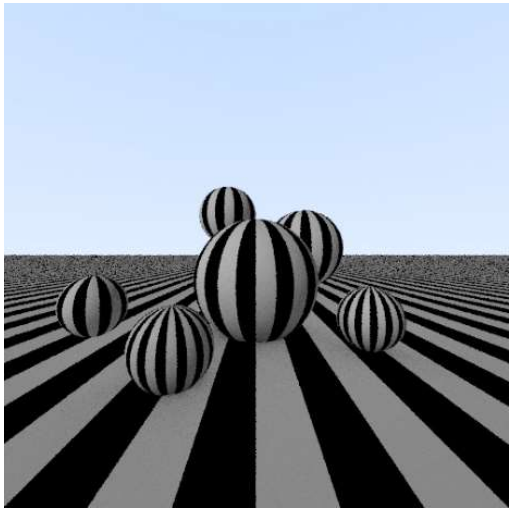
Le calcul des lignes se fait procéduralement, en modifiant les uvs de la primitive.

```
vec3 Material::getColor(vec3 uvS) const
{
    if (tex != nullptr)
    {
        return getColorInTexture(uvS);
    }
    else if (stripe)
    {
        float patternU = fmod(uvS.x * uvS.z, 1) < 0.5;
        return color * patternU;
    }
    else
        return color;
}
```

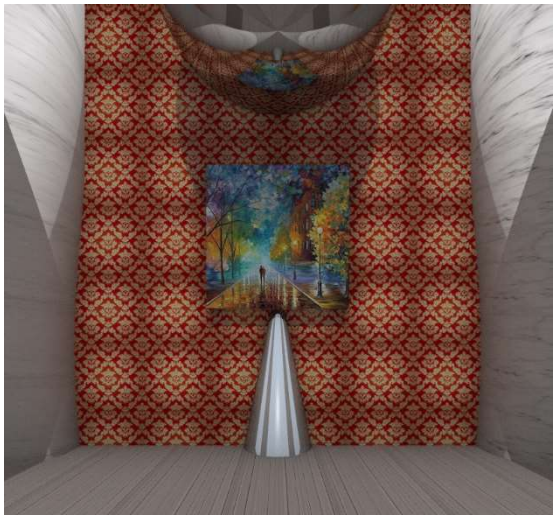
Code du material avec des lignes

```
vec3 Material::getColorInTexture(vec3 uv) const
{
    int u = int(uv.x * (tex->getWidth() - 1));
    int v = int(uv.y * (tex->getHeight() - 1));

    vec3 col = tex->getPixelColor(u, v);
    col.r /= 255.0f;
    col.g /= 255.0f;
    col.b /= 255.0f;

    return  col;
}
```

Code du material avec des textures



Material matte avec des lignes



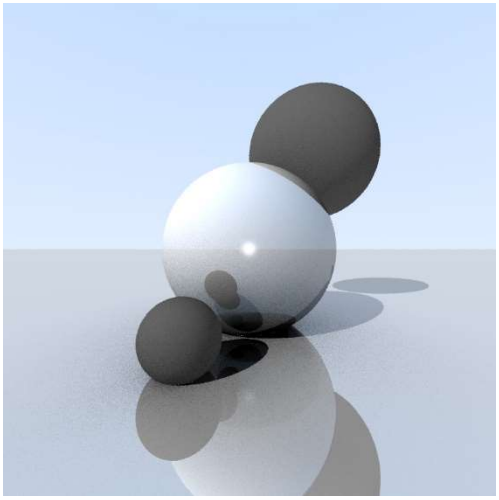
Material avec des textures différentes

c. Reflection

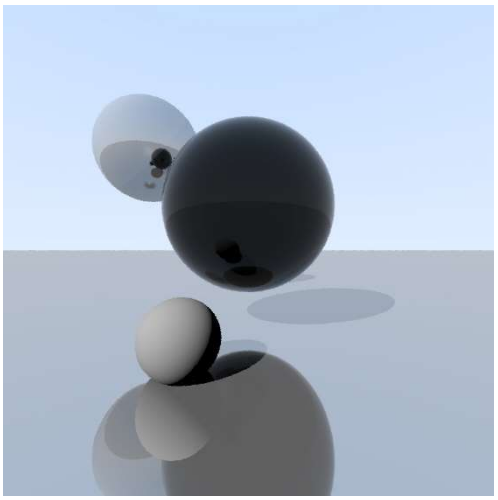
Pour les matériaux de type metallic et plastic, des réflexions sont calculées : un rayon secondaire est renvoyé de ce point d'intersection. Si ce rayon secondaire touche un autre objet de la scène, c'est la couleur renvoyée par ce deuxième rayon qui est la couleur du premier. Ce rayon secondaire est calculé par la fonction reflect().

```
vec3 reflect(const vec3& hitPos, const vec3& normal)
{
    vec3 rayonReflect = hitPos - normal * (2 * (hitPos.dot(normal)));
    return rayonReflect;
}
```

Cette réflexion est de type miroir.



Reflexion de type miroir

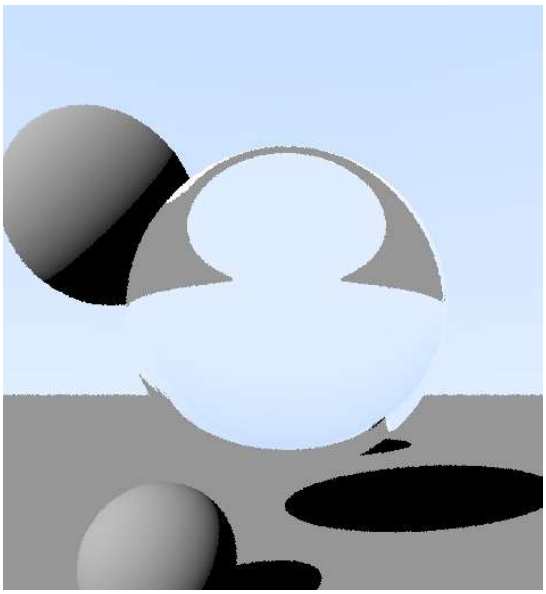


Reflexion avec un fresnel

d. Refraction

Lorsque le rayon primaire intersecte une primitive de type dielectric, un rayon secondaire est réfracté. On ajoute un bias pour tester si le rayon primaire a intersecté à l'intérieur ou à l'extérieur de la primitive. Si c'est à l'extérieur, le rayon secondaire est une réflexion (passe dans la fonction reflect()) mais s'il est à l'intérieur, il est réfracté (passe dans la fonction refract()).

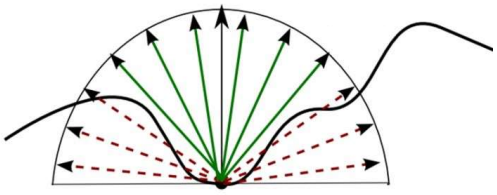
```
vec3 tracer::refract(const ray& hitPos, const vec3& normal, const float& ior)
{
    float cosi = hitPos.direction.dot(normal);
    float etai = 1, etat = ior;
    vec3 n = normal;
    if (cosi > 0)
    {
        etai = ior;
        etat = 1.0f;
        n = normal * -1;
    }
    else
    {
        etai = 1.0f;
        etat = ior;
        cosi = -cosi;
    }
    float eta = etai / etat;
    float k = 1 - eta * eta * (1 - cosi * cosi);
    return k < 0 ? vec3{ 1, 0, 0 } : (hitPos.direction * n) + n * (eta * cosi - sqrtf(k));
}
```



Primitive sphère avec de la refraction sans fresnel

IV) Feature de rendus supplémentaires

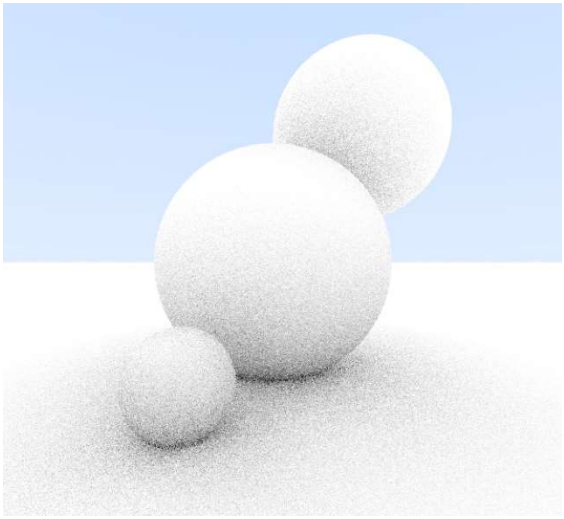
a. Ambient occlusion

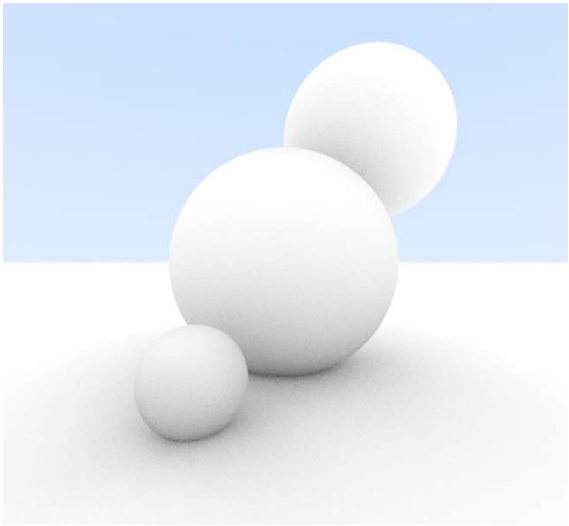


L'occlusion ambiante permet d'ajouter de l'ombre indirecte soft et les ombres ne sont plus noires pures.

Pour chaque rayon primaire qui a touché un objet de la scène, plusieurs rayons secondaires sont envoyés dans toutes les directions et testés s'ils touchent un autre objet ou la skysphere. Si un rayon touche la skysphere, il ajoute de l'intensité lumineuse à son point d'origine, multipliée ensuite par le reste du rendu.

```
float tracer::AmbientOcclusion(const vec3& pos, const vec3& n)
{
    randomNumbers random;

    float aoIntensity = 0.25f;
    int nbSamples = Properties::get()->getSampleAO();;
    float occlusionDistance = 2.0f;

    vec3 hitPos = pos + n * EPSILON;

    double percent = 0.0;
    for (int i = 0; i < nbSamples; i++)
    {
        vec3 hemisphereDir = random.randomDirectionInHemisphere(n);
        ray hemisphereRay = ray(hitPos, hemisphereDir);

        Intersection currentTry;
        GetIntersection(currentTry, hemisphereRay, vec3());

        percent += (currentTry.distance > occlusionDistance) ?
            1.0 : (currentTry.distance / occlusionDistance);
    }

    return percent / nbSamples;
}
```





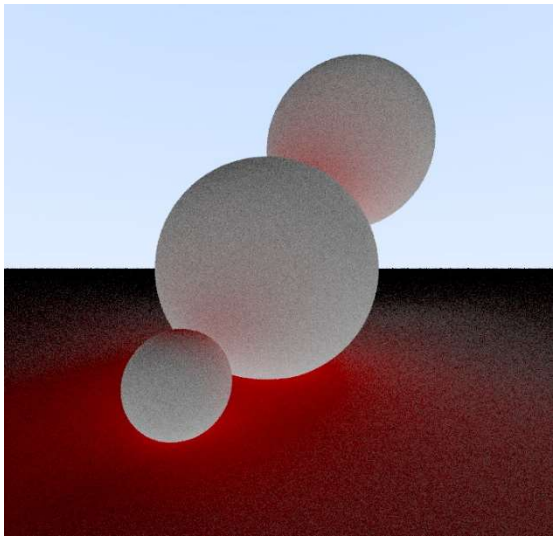
Rendu AO seule 10 samples sur une image en 800*800
Rendu AO seule 50 samples sur une image en 800*800

b. Global Illumination

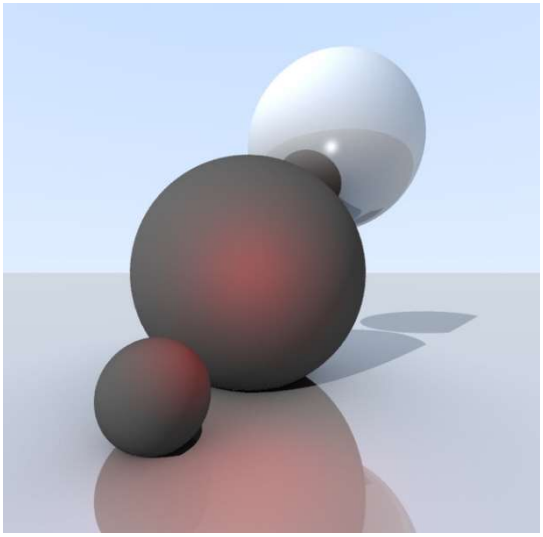
Pour aller encore plus loin, nous avons décidé de mettre en place un algorithme d'illumination globale qui simule des rayons de photons émanent d'une source lumineuse dans toutes les directions. C'est, dans l'idée, le même principe que pour l'occlusion ambiante.

```
vec3 tracer::GlobalIllumination(const vec3& pos, const vec3& n)
{
    randomNumbers random;
    int nbSamples = Properties::get()->getSampleGI();

    vec3 hitPos = pos + n * EPSILON;
    color indirectLigthing = vec3();

    for (int i = 0; i < nbSamples; i++)
    {
        vec3 hemisphereDir = random.randomDirectionInHemisphere(n);
        ray hemisphereRay = ray(hitPos, hemisphereDir);

        Intersection currentTry;
        color col;

        if (GetIntersection(currentTry, hemisphereRay, col))
            indirectLigthing += col;
    }

    return indirectLigthing / nbSamples;
}
```

Pour chaque intersection touchée par le rayon primaire, plusieurs rayons sont renvoyés (nombres modifiables par la variable nbSamples) de ce point d'impact à une hémisphère autour d'eux. La couleur retournée est une accumulation des couleurs des différents rayons renvoyés.



Rendu GI seule 10 samples sur une image en 800*800



Rendu GI 50 samples sur une image en 800*800

La global illumination n'est pas ajoutée aux matériaux réflectifs et réfractifs.

V) Propriétés du rendu

Lors du lancement du programme, l'utilisateur peut choisir la scène qu'il désire rendre et ses paramètres de rendu.

a. Singleton

Les propriétés du rendu sont toutes stockées dans une classe singleton, à instanciation unique : Properties.h. Lorsque l'utilisateur choisit de modifier des options du rendu, les paramètres par défaut sont corrigés grâce à des setters dans la classe Properties.h.

```
class Properties
{
    static Properties* singleton;
    Properties(): width(500), height(500),
        name("Rendu"), shadow(true), AO(true),
        numberOfSampleAO(10), GI(true),
        numberOfSampleAA(4), numberOfSampleGI(10),
        readyToRender(false){}

    //basic properties
    int width;
    int height;
    string name;
    bool shadow;

    //more properties
    bool AO;
    int numberOfSampleAO;
    bool GI;
    int numberOfSampleGI;
    int numberOfSampleAA;

    bool readyToRender;
```

b. Lecture de scènes

L'utilisateur peut choisir entre quatre scènes déjà définis dans des fichiers texte loadés et décryptés par la classe SceneReader.h.

Le fichier contient les matériaux utilisés dans la scène, les primitives et les lights.

Le fichier est lu ligne par ligne et chaque ligne est stockée dans un vector de string.

La première string est testée. Si c'est un « m », un material est créé avec les paramètres contenu dans la suite du vector de string. Si la première string est un « p », une primitive est

créée et enfin si c'est un « l », une light est créée. La deuxième string du vector permet de connaitre quel objet instancié : plane, sphere, tri…., directional light ou point light.

```
m wall MATTE 1.f 1.f 1.f 0 0.f 0.f Wallpaper.jpg
m carrrelage MATTE 1.f 1.f 1.f 0 0.f 0.f carrelage.jpg
m painting MATTE 1.f 1.f 1.f 0 1.f 50.f painting.jpg
m marble MATTE 1.f 1.f 1.f 0 1.f 50.f marble.jpg
m metallic METALLIC 0.5f 0.5f 0.5f 0 1.0f 100.0f N
p Square 0.f -1.5f 2.f 0.f 1.f 0.f 50.f carrrelage
p Square 0.f 0.f 5.f 0.f 0.f -1.f 50.f wall
p Cube 0.f 1.5f 5.75f 1.f painting
p Cylinder -4.f, 0.f, 5.f 1.0f marble
p Cylinder 4.f, 0.f, 5.f 1.0f marble
p Cylinder -4.f, 0.f, 1.f 1.0f marble
p Cylinder 4.f, 0.f, 1.f 1.0f marble
p Cone 0.f -1.5f 5.f 0.5f 2.5f metallic
p Sphere 0.f 4.5f 4.f 1.5f metallic
l DirectionLight 0.f -1.f 1.f 1.f 1.f 1.f 1.f
```

ANNEXE

Répartition des tâches :

- Algorithme principal du lancé de rayons : Arthur et Margot
- Antialiasing : Margot
- Primitives : (calcul des UVs et Normales compris)
  - Sphere/Plan infini/Triangle : Margot
  - Cylindre infini/Cube/Square/cône infini : Arthur
- Blinn-Phong : Arthur et Margot
- Lighting : directional Light et Point Light : Arthur
- Ombres : Arthur
- Choix matériaux homogènes ou duo : Margot
- Texturing : Arthur
- Reflection : Arthur et Margot
- Fresnel et refraction : Margot
- AO et GI : Arthur et Margot
- Choix de scènes, lecture de scènes et singleton : Margot
- Enregistrement en jpg : Arthur et Margot

Lien github :

https://github.com/Paillette/Raytracer