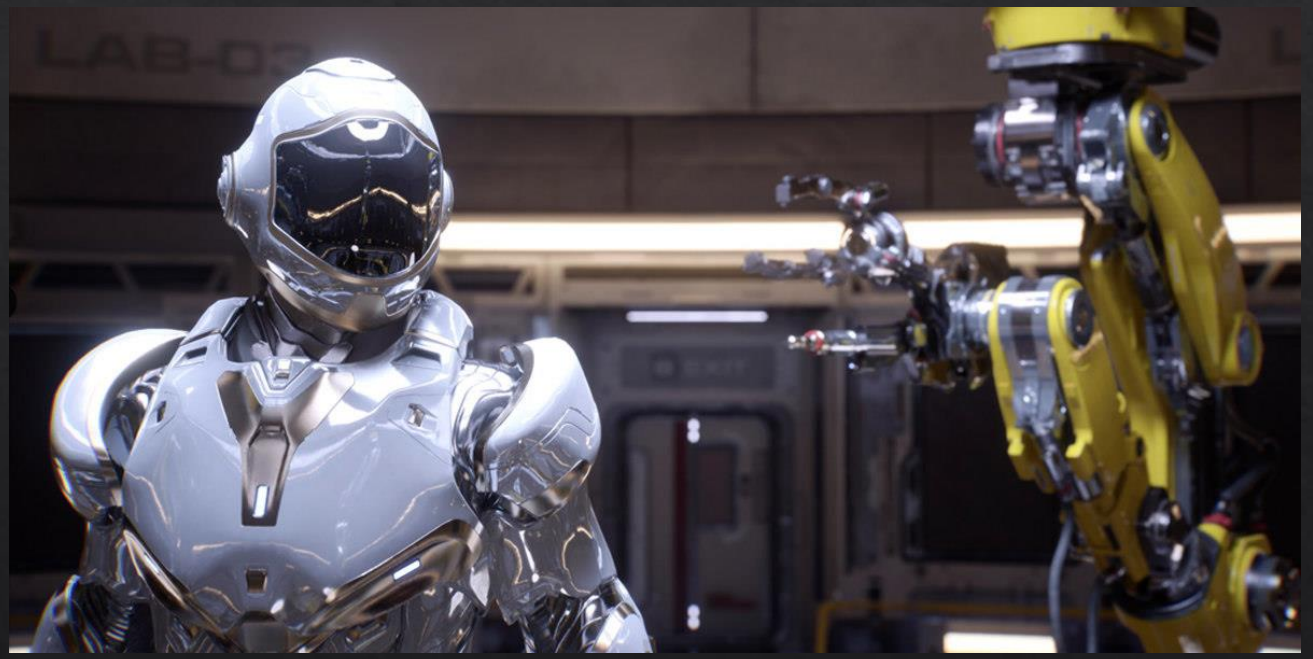
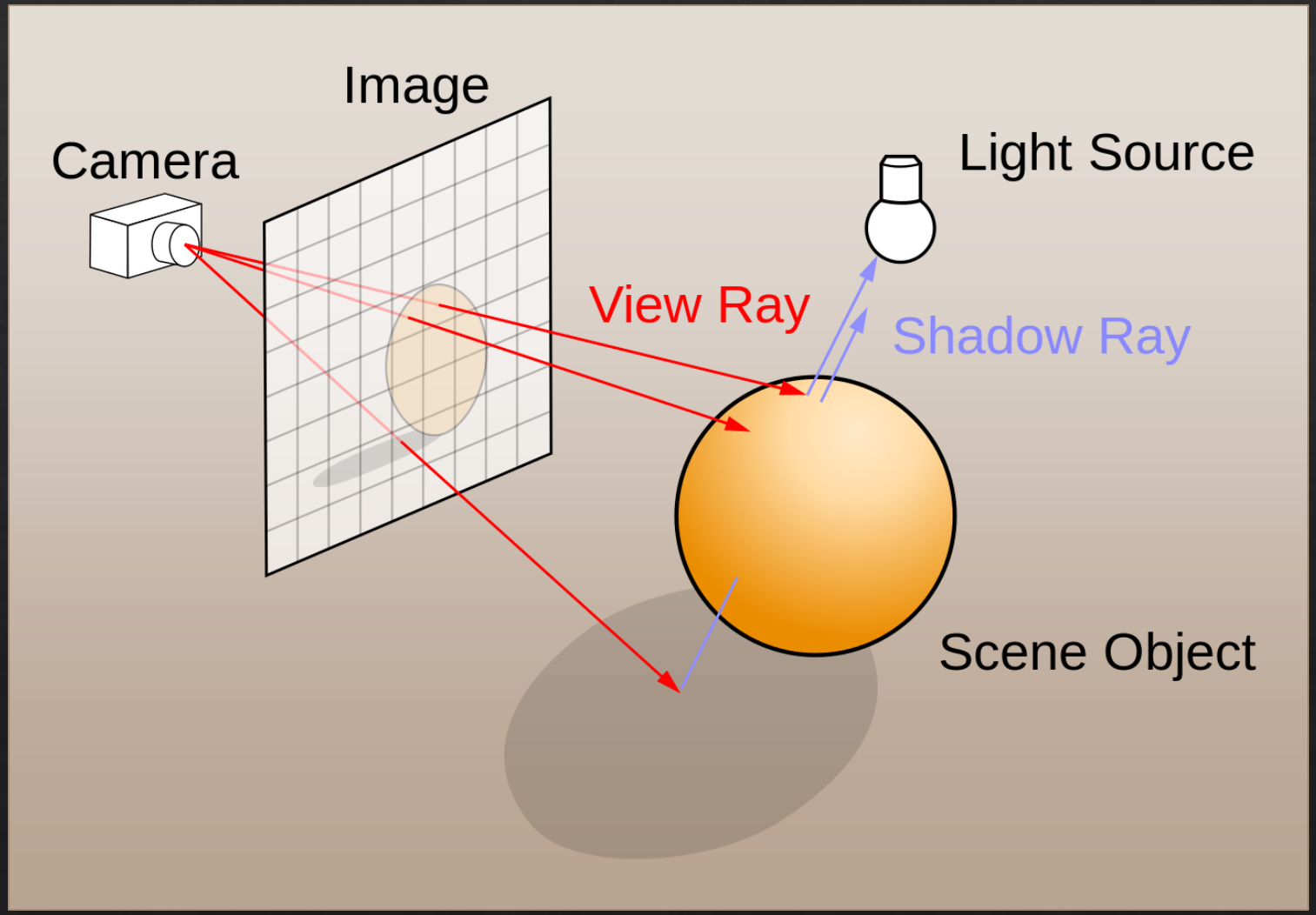
Lancer de rayons

Luc Jiang

Contexte d’utilisation

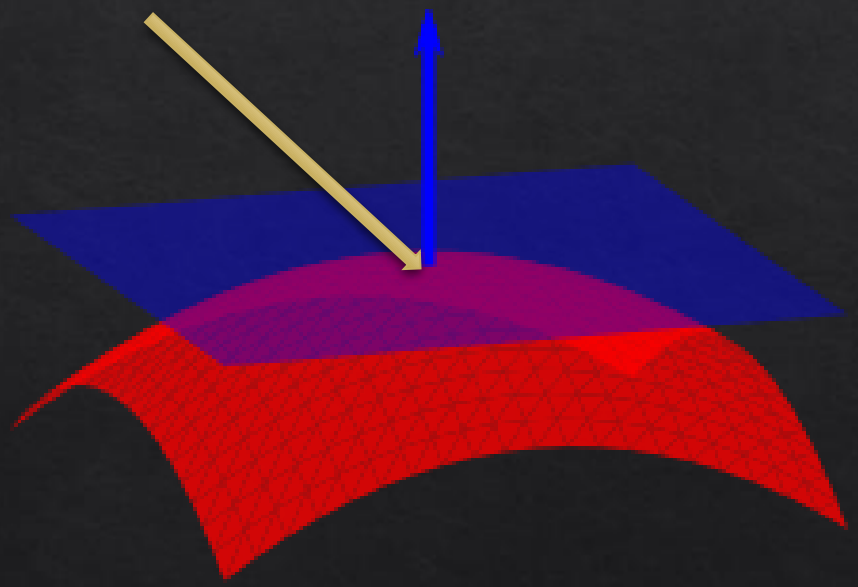


Principe du lancer de rayons

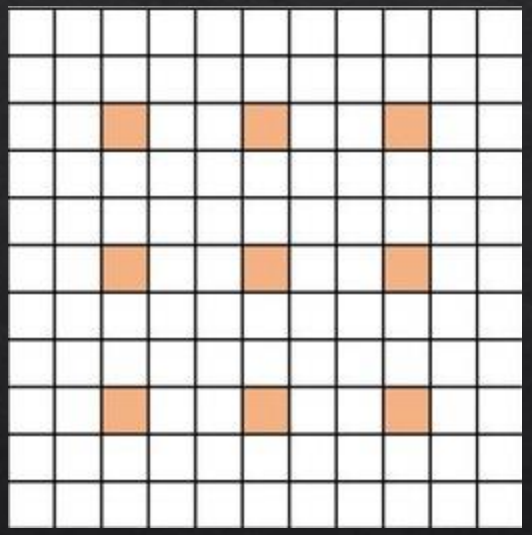


Réflexions et Choix

Intensité de la lumière :



Lumière diffusée:



Structure du code réalisé

- Les Classes :
  - Point
  - Camera
  - Objet (Sphère et Plan)
  - Source de lumière
  - Le Monde

- Les Fonctions et Méthodes importantes:
  - Lancer un rayon
  - Mise à jour des couleurs
  - Diffusion des couleurs
  - Le rendu de l'image

- Utilisation de SFML:
  - Afficher une fenêtre
  - Gestion de l'image et des couleurs (RGBA)

Les Classes

```
class Point
{
private:
    float x, y, z;

public:
    Point(float u, float v, float w) {x=u; y=v; z=w;}

    float getx() const {return x;}
    float gety() const {return y;}
    float getz() const {return z;}

    void setx(float a) {x=a;}
    void sety(float a) {y=a;}
    void setz(float a) {z=a;}
    void setcoord(float a, float b, float c) {x=a; y=b; z=c;}

    Point operator + (const Point& aPoint) const {

    Point operator - (const Point& aPoint) const {

    Point operator ^ (const Point& p) const {

    Point operator * (const float& t) const {

    float operator * (const Point& aPoint) const {

    float norm() const {

    void normalized() {

    void getbase(Point* ux, Point* uz);
```

```
class Camera
{

class Object
{
protected:
    Point position;
    sf::Uint8 R,G,B;
    Object_type type;

public:
    Object(Point position, sf::Uint8 r, sf::Uint8 g, sf::Uint8 b)

    Point getpos() const {return position;}

    sf::Uint8 getR() const {return R;}
    sf::Uint8 getG() const {return G;}
    sf::Uint8 getB() const {return B;}

    Object_type gettype() const {return type;}

};

class Sphere : public Object
{

class Plan : public Object
{

class Light
{

class World
{
```

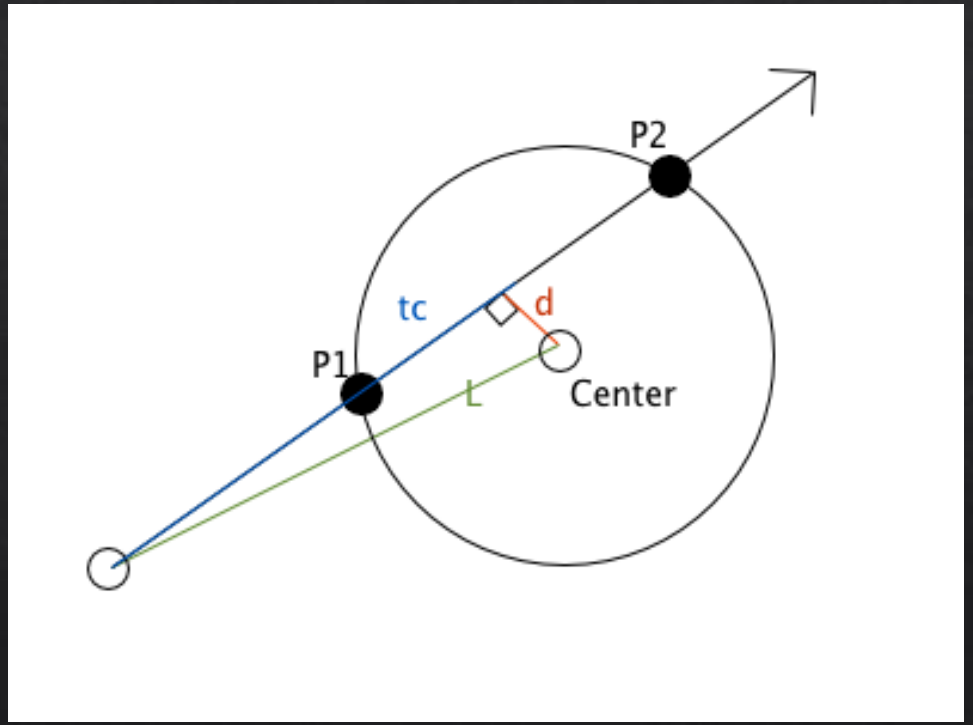
Les Classes

```cpp
class Sphere : public Object
{
private:
    float radius;

public:
    Sphere(Point position, sf::Uint8 r, sf::Uint8 g, sf::Uint8 b, float radius) :
        Object(position, r, g, b), radius(radius) {type = sphere;}

    float getradius() const {return radius;}

};

class Plan : public Object
{
private:
    Point normale;

public:
    Plan(Point position, sf::Uint8 r, sf::Uint8 g, sf::Uint8 b, Point Normale)
        : Object(position, r, g, b), normale(Normale) {type = plan;}

    Point getnormale() const {return normale;}

};
```

```cpp
class World
{
private:
    Camera camera;
    std::vector<Object*> objects;
    std::vector<Light> lights;
    sf::Image* image;

public:
    World(Camera Cam, std::vector<Object*> obj, std::vector<Light> lit, sf::Image* img) :
        camera(Cam), objects(obj), lights(lit), image(img) {}

    void raytracing();
    bool ray(Point& position, Point& direction, float* tfinal, int* objseen, int ignore);
    void diffusion(Point& touch, int objseen, float* Dintensity);

};
```

Les Fonctions

```
bool World::ray(Point& position, Point& direction, float* tfinal, int* objseen, int ignore)
{
    *tfinal = -1;
    *objseen = -1;
    int itobj = 0;

    if (itobj == ignore)
        itobj++;

    //on parcourt les objects du monde
    while ((unsigned int)itobj<objects.size())
    {
        Object_type type = (objects[itobj])->gettype();
        //std::cout << objects.size() << std::endl;

        //On parcourt la liste des sphères
        if (type == sphere)
        {
            Sphere* obj = (Sphere*)objects[itobj];
            //std::cout << "inside" << std::endl;
            Point objpos = obj->getpos();
            float radius = obj->getradius();
            float t = -1;

            float tc = (objpos - position)*direction;
            float D = (objpos - position).norm();
            //std::cout << objpos.getz() << std::endl;

            if (D < radius) //Dans la sphère
            {
                t = sqrt(radius*radius - D*D + tc*tc) + tc;
            } else
            {
                float h = sqrt(D*D - tc*tc);
                if (tc >= 0 && h <= radius)
                {
                    t = tc - sqrt(radius*radius - h*h);
                }
            }

            if (*tfinal == -1)
            {
                if (t>0)
                {*tfinal = t; *objseen = itobj;}
            }
            else
            {
                if (0<t && t<*tfinal)
                {*tfinal = t; *objseen = itobj;}
            }
        }
```

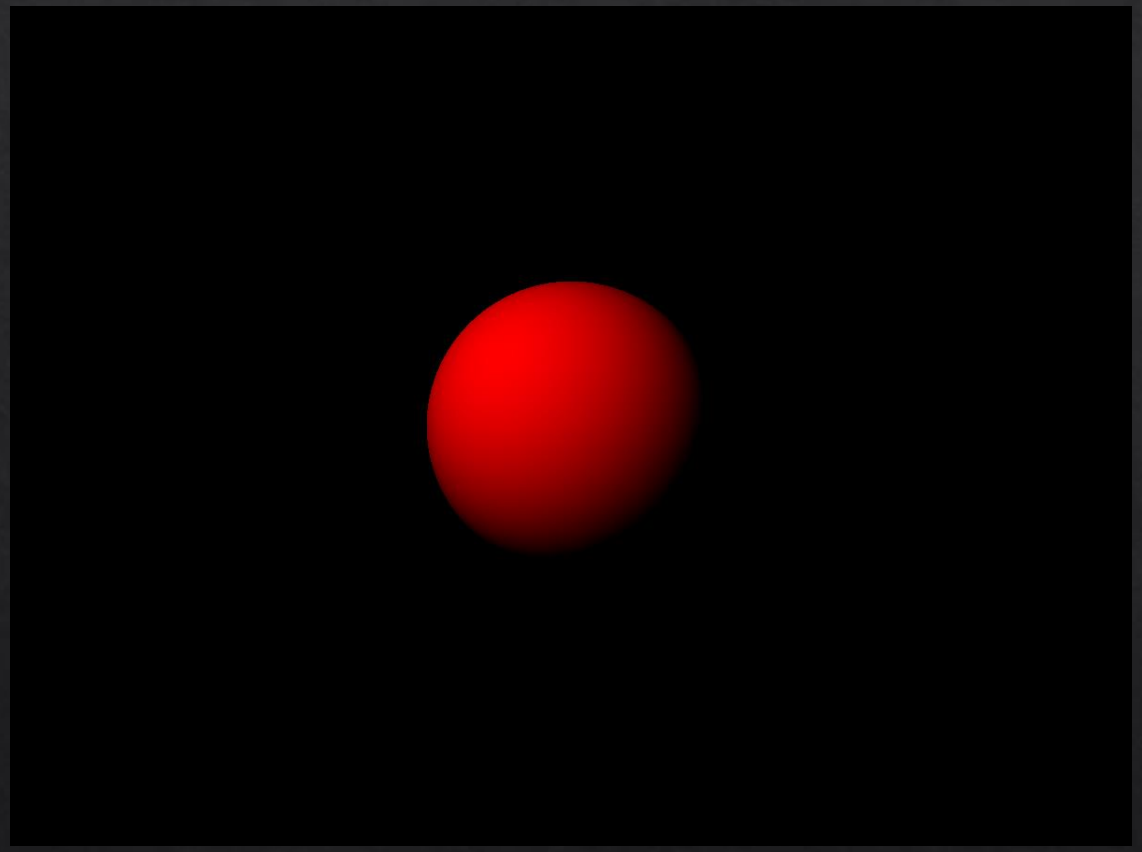


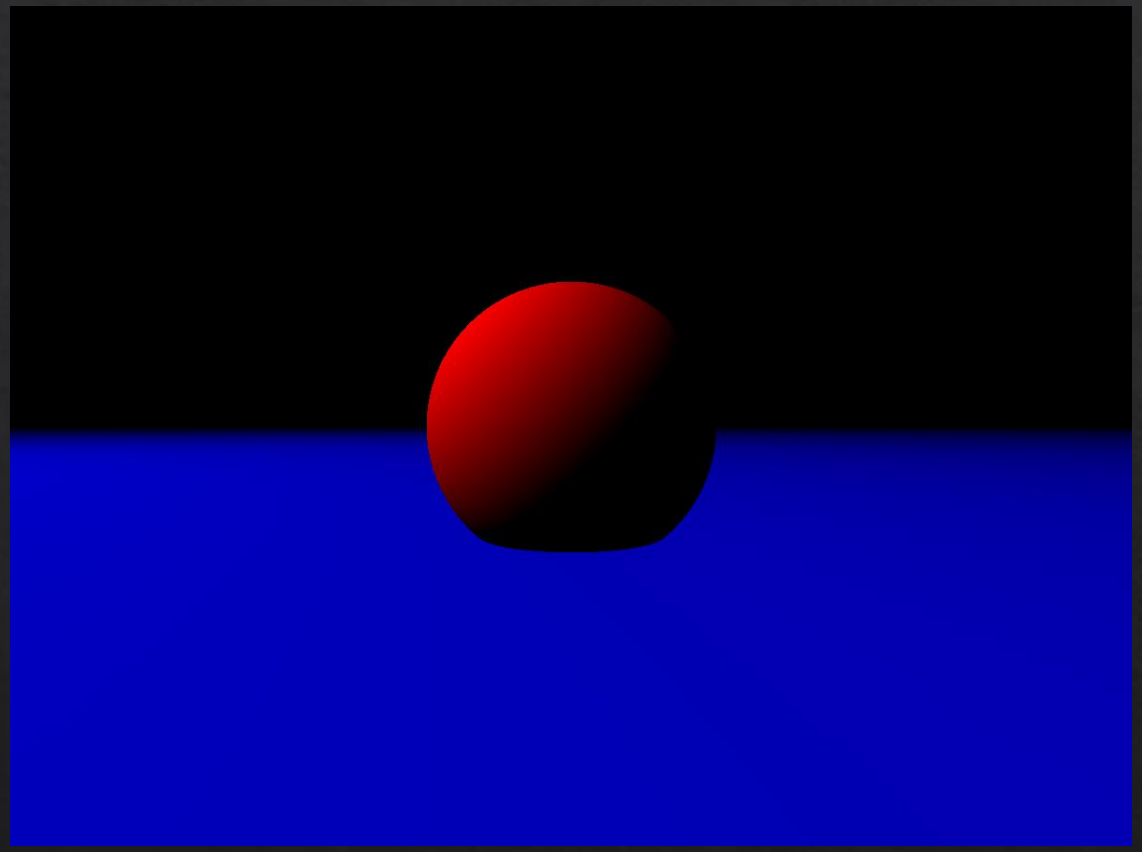
Les Fonctions

```
void World::raytracing()
{
    //Init
    Point camloc = camera.getloc();
    //Point camori = camera.getori(); plus complexe

    Point pixori(0, dcam, 0);

    int objseen;
    float tfinal;

    for (int i=0; i<height; ++i)
    {
        for (int j=0; j<width; ++j)
        {
            //std::cout << i << " ," << j << std::endl;
            //Calcul de la direction du rayon passant par le pixel (j,i);
            pixori.setx(-width/2 + j);
            pixori.setz(height/2 - i);
            pixori.sety(dcam);

            pixori.normalized();

            tfinal = -1;
            objseen = -1;
            if (this->ray(camloc, pixori, &tfinal, &objseen, INT_MAX))
            {
                Object* obj = objects[objseen];
                Point touch = camloc + pixori*tfinal;
                Point normale(0,0,0);
                //std::cout << obj->gettype() << std::endl;
                if (obj->gettype() == sphere) {
                    normale = (touch - obj->getpos());}
                else if (obj->gettype() == plan) {
                    normale = ((Plan*)obj)->getnormale();}
                normale.normalized();

                //On parcourt les sources de lumières pour pouvoir afficher les couleurs des objets
                for (unsigned int itlight = 0; itlight < lights.size(); ++itlight)
                {
                    Light* lit = &lights[itlight];
```

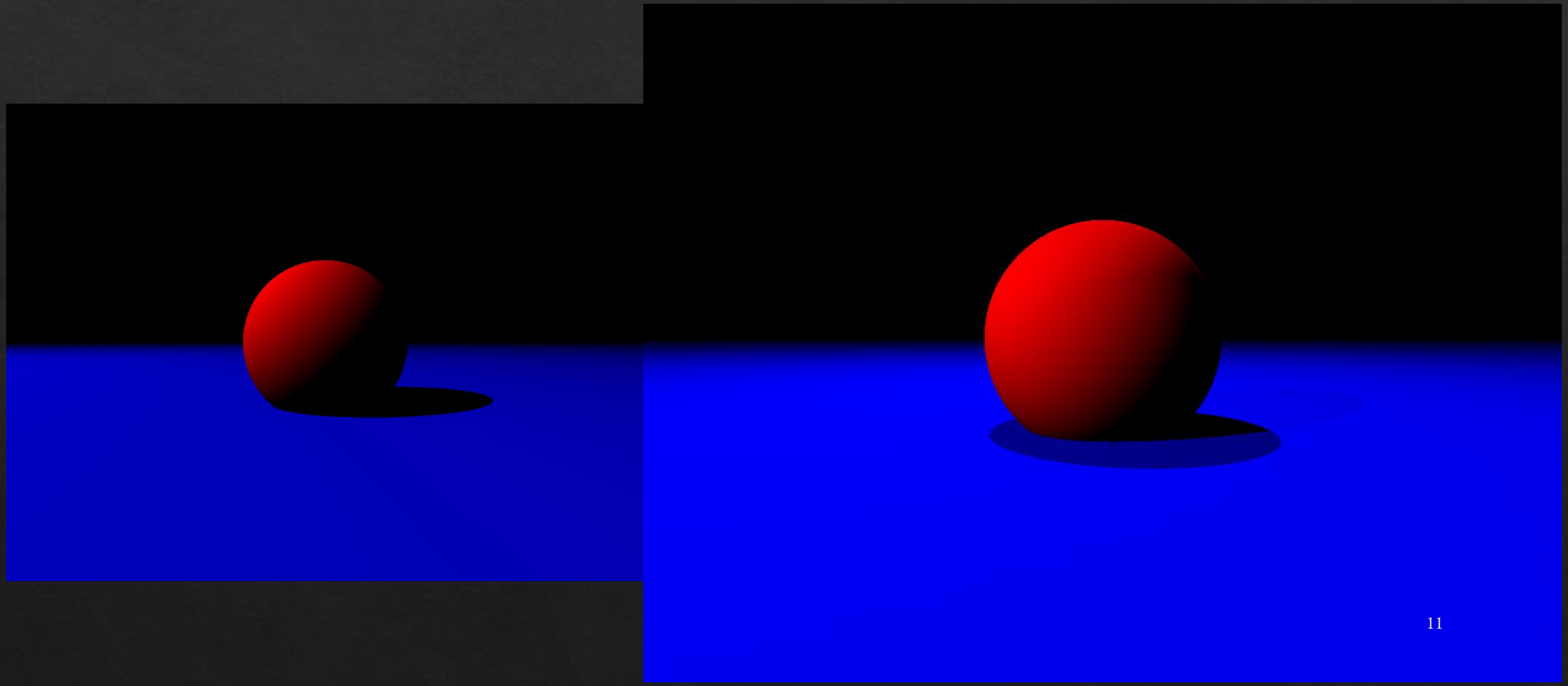
```
//On parcourt les sources de lumières pour pouvoir afficher les couleurs des objets
for (unsigned int itlight = 0; itlight < lights.size(); ++itlight)
{
    Light* lit = &lights[itlight];

    Point lightray = (lit->getloc() - touch);
    lightray.normalized();
    float intensity = lightray*normale;

    float tlight = -1;
    int objobstruct = -1;
    bool obstruct = this->ray(touch, lightray, &tlight, &objobstruct, objseen);

    //obstruct = false; //A (dé)commenter pour afficher ou non les ombres engendrées par les objets entre eux.

    if (intensity > 0 && !obstruct)
    {
        sf::Color colormax(obj->getR(),obj->getG(),obj->getB());
        float intensity3D[3] = {intensity, intensity, intensity};

        updateColor(image, j, i, colormax, intensity3D);
        //std::cout << i << " ," << j << std::endl;

    }
}

float Dintensity[3] = {0,0,0};
this->diffusion(touch, objseen, Dintensity);

sf::Color colormax(obj->getR(),obj->getG(),obj->getB());
updateColor(image, j, i, colormax, Dintensity);
```

Résultats : premiers objets

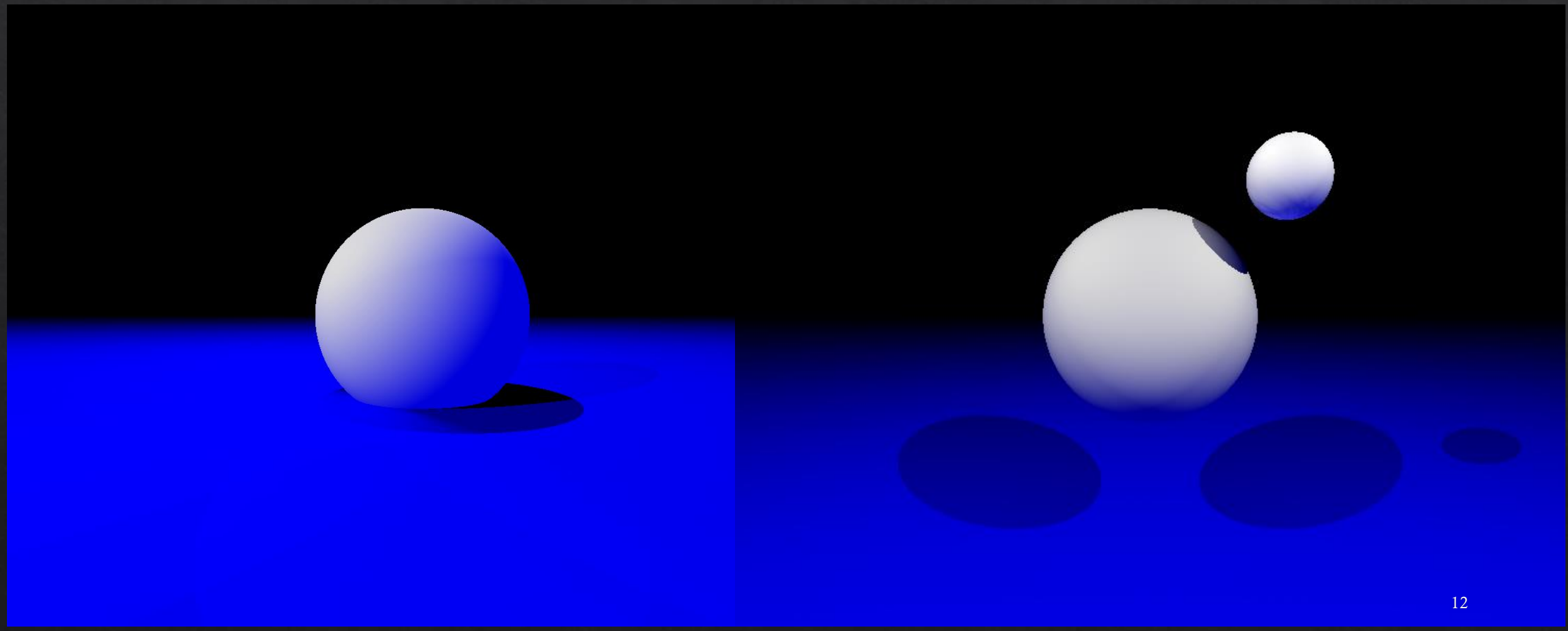




Résultats : les ombres



Résultats : diffusion de la lumière



Amélioration possible

- Fichier de scènes
- Formes plus complexes (cubique, voire maillage)
- Différentes textures (transparentes, absorbantes et réfléchissantes)
- Sources de lumière de couleurs et de formes différentes
- Parallélisation du programme
- Augmenter le nombre de réflexion