Ray Tracer : Phase Finale

Mateusz Birembaut

January 5, 2025

Contents

1 ImGui

Ajout de "ImGui" pour modifier des paramètres lors de l'exécution.

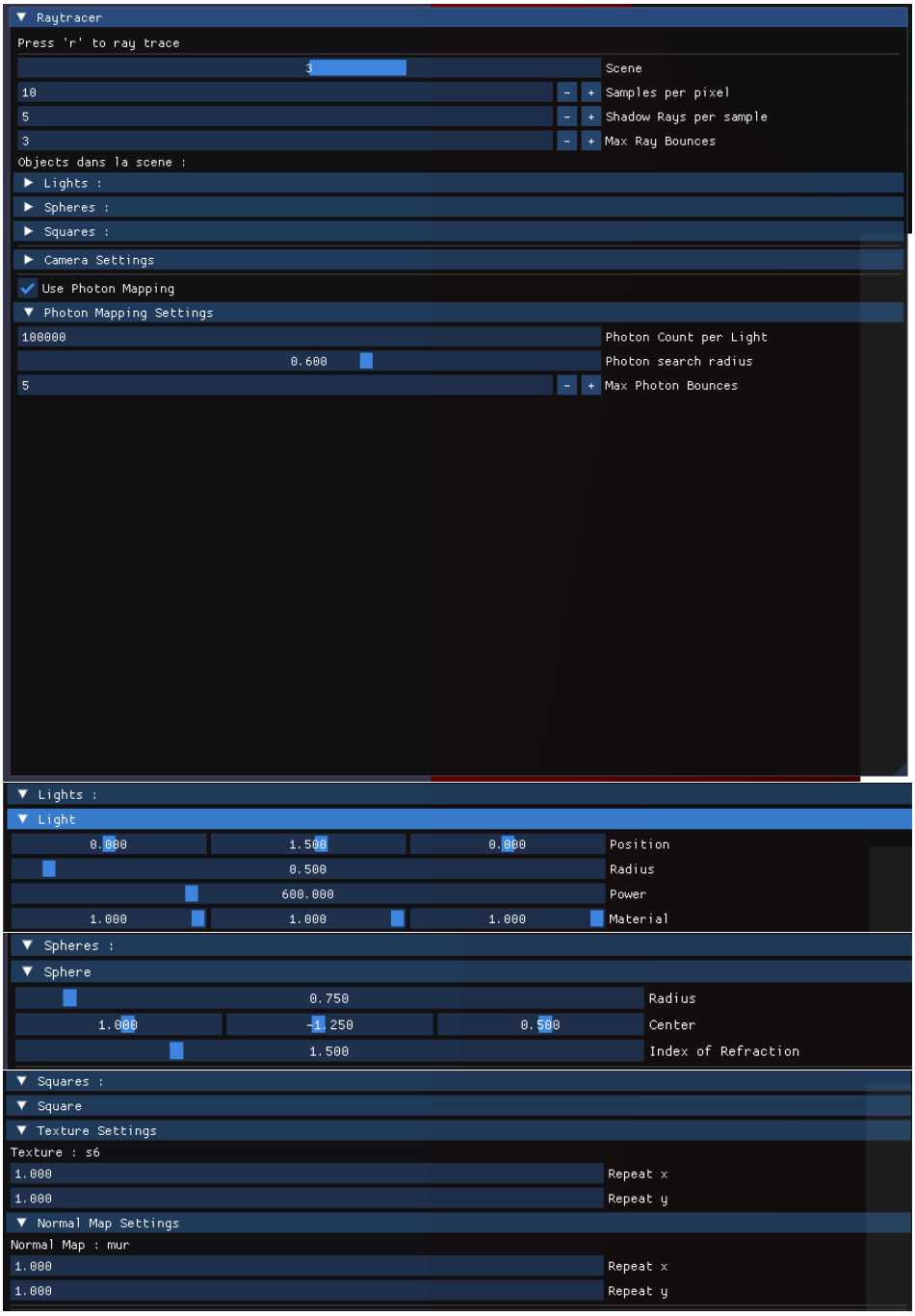


Figure 1: UI

2 Multi-threading

Création de threads pour rendre l'image plus rapidement. Je crée plus de threads que de threads du processeur, c'est ce qui me donnait les meilleures performances. On passe de 1.93 seconde pour la cornellbox (10 samples par pixel, 480x480) à 0.24 seconde pour les mêmes paramètres.

```
int num_threads = std::thread::hardware_concurrency();
int block_size_x = w / num_threads;
int block_size_y = h / num_threads;

unsigned int nsamples = g_samplesPerPixel;
std::vector< Vec3 > image( w*h , Vec3(0,0,0) );
auto start = std::chrono::high_resolution_clock::now();

std::vector<std::thread> threads;
for (int i = 0; i < num_threads; ++i) {
    for (int j = 0; j < num_threads; ++j) {
        int start_x = i * block_size_x;
        int end_x = (i == num_threads - 1) ? w : start_x + block_size_x;
        int start_y = j * block_size_y;
        int end_y = (j == num_threads - 1) ? h : start_y + block_size_y;
        threads.emplace_back(ray_trace_block, start_x, end_x, start_y, end_y, w, h, nsamples, std::ref(image), std::ref(kdTreePhotonMap), pos);
    }
}

for (auto& thread : threads) {
    thread.join();
}
```

Figure 2: Création threads

```
void ray_trace_block(int start_x, int end_x, int start_y, int end_y, float w, float h, unsigned int nsa
    float inv_w = 1.0f / w;
    float inv_h = 1.0f / h;
    float inv_nsamples = 1.0f / nsamples;
    for (int y = start_y; y < end_y; y++) {
        for (int x = start_x; x < end_x; x++) {
            Vec3 sum_color(0, 0, 0);
            for (unsigned int s = 0; s < nsamples; ++s) {
                float u = (x + dist05(rng)) * inv_w;
                float v = (y + dist05(rng)) * inv_h;
                Vec3 color = scenes[selected_scene].rayTrace(
                    kdTreePhotonMap,
                    depth_of_field_ray(u, v, camera.focalPlaneDistance, camera.apertureSize, pos)
                );
                sum_color += color;
            }
            image[x + y * w] = sum_color * inv_nsamples;
        }
    }
}
```

Figure 3: Rendu d'un bloc de l'image

3 Miroir et Transparence

Lorsque l'on intersecte un objet avec "material.type" = "Material_Glass" ou "Material_Mirror", on relance un rayon depuis le point d'intersection. La direction sera donné par refract ou reflect. (Pour les objets transparents, si le rayon incident est trop incliné, on utilise reflect et pas refract).

```
Vec3 reflect(const Vec3& N, const Vec3& I) {
    float cosI = -1 * Vec3::dot(N, I);
    Vec3 reflectedDirection = (I + 2 * cosI * N);
    reflectedDirection.normalize();
    return reflectedDirection;
}
```

Figure 4: Réflexion d'un rayon

```
Vec3 refract(const Vec3 &I, const Vec3 &N,  float eta_t,  float eta_i=1.f) {
    float cosi = -std::max(-1.f, std::min(1.f, Vec3::dot(I, N)));
    Vec3 n = N;
    if (cosi<0){
        n = -1 * N;
        std::swap(eta_i, eta_t);
    }
    float eta = eta_i / eta_t;
    float k = 1 - eta*eta*(1 - cosi*cosi);
    if (k < 0){
        return reflect(N,I);
    }
    Vec3 direction_out = I*eta + n*(eta*cosi - sqrtf(k));
    direction_out.normalize();
    return direction_out;
}
```

Figure 5: Réfraction d'un rayon

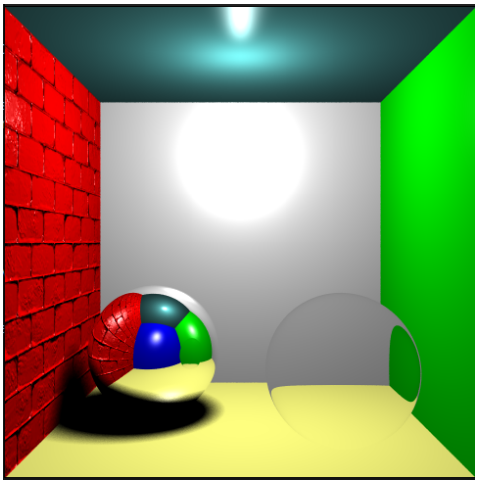


Figure 6: Sphère Réflective et Transparente (index de réfraction à 1.05)

4 Texture et Normal Map

On utilisera la même fonction pour récupérer la couleur dans les normals maps ou textures.

```
Vec3 samplePPMtoVec3(float u, float v , int repeatU, int repeatV, DATA_TYPE data_type) {
    u *= repeatU;
    v *= repeatV;

    u = u - std::floor(u);
    v = v - std::floor(v);

    if ( data_type == DATA_TYPE::NORMAL_MAP ) {
        v = 1.0f - v; // flip
    }

    u = std::clamp(u, 0.0f, 1.0f);
    v = std::clamp(v, 0.0f, 1.0f);

    int x = u * (w - 1);
    int y = v * (h - 1);

    unsigned int index = y * w + x;

    if (index < 0 || index >= data.size()) {
        std::cerr << "Error: Texture coordinate out of bounds. u: " << u << ", v: " << v
        std::cerr << "Texture size: " << w << "x" << h << std::endl;
        return Vec3(0.0f, 0.0f, 0.0f);
    }

    RGB color = data[index];

    return Vec3(color.r / 255.0f,
                color.g / 255.0f,
                color.b / 255.0f);
}
```

Figure 7: Récupération de la couleur pour un u, v

Pour les normales map, il faut utiliser une matrice TBN (tangente, bitangente et normale) pour faire passer la normale 2D en normale 3D sur l'objet. J'ai seulement implémenté les normals maps pour les carrés et sphères.

```
void updateNormal(Vec3 & N, ppmLoader::ImageRGB * normalMap, float u, float v, Material & mat, const RaySceneIntersec
    Vec3 tangentSpaceNormal = normalMap->samplePPMtoVec3(u, v, mat.n_uRepeat, mat.n_vRepeat, DATA_TYPE::NORMAL_MAP);
    tangentSpaceNormal = 2 * tangentSpaceNormal - Vec3(1, 1, 1);
    tangentSpaceNormal.normalize();

    Vec3 tangent, bitangent;
    if( raySceneIntersection.typeOfIntersectedObject == 0){
        tangent = Vec3(-sin(v), 0, cos(u));
        tangent.normalize();
    }else if (raySceneIntersection.typeOfIntersectedObject == 1){
        tangent = raySceneIntersection.raySquareIntersection.rightVector;
        tangent.normalize();
        tangent *= -1;
    }

    bitangent = Vec3::cross(N, tangent);
    bitangent.normalize();

    Mat3 TBN = computeTBN(tangent, bitangent, N);

    Vec3 worldNormal = transformToWorldSpace(tangentSpaceNormal, TBN);
    N = worldNormal;
}
```

Figure 8: Mise à jour de la normale

Figure 9: 1 square avec texture et normal map

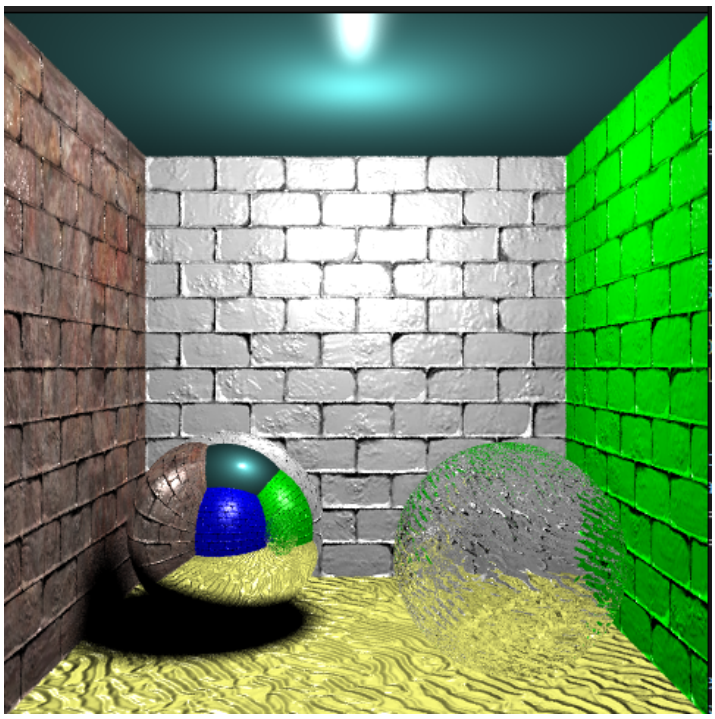


Figure 10: 1 square avec texture et normal map

5 Intersection Mesh / Triangles : KdTree

La bounding box du kdTree permettra de determiner si un rayon va intersecter avec le mesh.

```cpp
class KdTree {


public:

    BoundingBox box;
    KdNode<Triangle>* root;
    int maxDepth;
```

```cpp
template <typename T>
struct KdNode {
    KdNode *left, *right;

    BoundingBox node_box;
    int dimSplit;
    float splitDistance;

    bool isLeaf = false;
    std::vector<T> primitives;
};
```

Figure 11: Classe KdTree

On commence par prendre la dimension la plus grande, ce sera la dimension à couper. Ensuite, on cherche la meilleure coupe, pour faire cela :

- Je coupe la dimension à split en 2 à la distance $parentBox.min[dimSplit]+i*(diff[dimSplit]/splitsToTest)$.
- On place les triangles à gauche ou à droite ou dans les deux s'ils chevauchent.
- On calcule le cout, si c'est le cout le plus petit jusqu'à maintenant, on le garde.

Une fois qu'on a la distance optimale, on place nos éléments dans les nœuds de droite, de gauche ou dans les deux et on rappelle cette fonction sur les deux nœuds tant qu'on n'a pas atteint la profondeur voulue.

```cpp
void KdTreeBuild(BoundingBox _parentBox, KdNode<Triangle>* _node, const std::vector<Triangle>& _triangles,int splitsToTest, int depth = 0)
    if (depth == maxDepth || _triangles.size() <= 1) {
        _node->isLeaf = true;
        _node->primitives = _triangles;
        _node->left = nullptr;
        _node->right = nullptr;
        return;
    }

    Vec3 diff = _parentBox.max - _parentBox.min;
    _node->dimSplit = diff.getMaxAbsoluteComponent();
    //_node->splitDistance = _parentBox.min[_node->dimSplit] + diff[_node->dimSplit] / 2;
    _node->splitDistance = findBestSplit(_node, _parentBox, _triangles, _node->dimSplit, splitsToTest);

    std::vector<Triangle> leftTriangles;
    std::vector<Triangle> rightTriangles;

    for (const Triangle& triangle : _triangles){
        BoundingBox triangleBox = BoundingBox::triangleBoundingBox(triangle);
        if (triangleBox.min[_node->dimSplit] <= _node->splitDistance) leftTriangles.push_back(triangle);
        if (triangleBox.max[_node->dimSplit] > _node->splitDistance) rightTriangles.push_back(triangle);

    }
    _node->left = new KdNode<Triangle>();
    _node->right = new KdNode<Triangle>();

    BoundingBox leftBox = _parentBox;
    leftBox.max[_node->dimSplit] = _node->splitDistance;
    BoundingBox rightBox = _parentBox;
    rightBox.min[_node->dimSplit] = _node->splitDistance;

    _node->left->node_box = leftBox;
    _node->right->node_box = rightBox;

    if (depth != maxDepth){
        KdTreeBuild(leftBox, _node->left, leftTriangles, splitsToTest ,depth + 1);
        KdTreeBuild(rightBox, _node->right, rightTriangles, splitsToTest ,depth + 1);
    }
}
```

Figure 12: Build KdTree

Si le rayon intersection bien la box, on parcourt le kdtree jusqu'à une feuille. On test si le rayon intersecte un des triangles de la feuille.

```
RayTriangleIntersection traverse(Ray const & ray, KdNode<Triangle>* _node, float t_start, float t_end) const {
    if (_node->isLeaf){
        RayTriangleIntersection intersection;
        intersection.t = FLT_MAX;
        for (const Triangle& triangle : _node->primitives) {
            RayTriangleIntersection tempIntersection = triangle.getIntersection(ray);
            if (tempIntersection.intersectionExists && tempIntersection.t < t_end && tempIntersection.t > t_start && tempIntersection.t < intersection.t)
                intersection = tempIntersection;
            }
        }
        return intersection;
    }

    std::pair<float, float> interval = _node->node_box.intersect(ray);
    if (interval.first == INFINITY && interval.second == INFINITY) {
        return RayTriangleIntersection();
    }

    float t = (_node->splitDistance - ray.origin()[_node->dimSplit]) / ray.direction()[_node->dimSplit];

    KdNode<Triangle>* firstNode;
    KdNode<Triangle>* secondNode;

    if (ray.direction()[_node->dimSplit] >= 0) {
        firstNode = _node->left;
        secondNode = _node->right;
    } else {
        firstNode = _node->right;
        secondNode = _node->left;
    }

    if (t <= t_start) {
        return traverse(ray, secondNode, t_start, t_end);
    } else if (t >= t_end) {
        return traverse(ray, firstNode, t_start, t_end);
    } else {
        RayTriangleIntersection hit = traverse(ray, firstNode, t_start, t);
        if (hit.intersectionExists && hit.t <= t && hit.t >= 0.0001) {
            return hit;
        }
        return traverse(ray, secondNode, t, t_end);
    }
}
```

Figure 13: Traverse KdTree

```cpp
RayTriangleIntersection getIntersection( Ray const & ray ) const {
    RayTriangleIntersection result;

    // 1) check that the ray is not parallel to the triangle:
    if(isParallelTo(ray)){
        //std::cout << "le triangle est parallel to the triangle" << std::endl;
        return result;
    }

    Vec3 e1 = m_c[1] - m_c[0];
    Vec3 e2 = m_c[2] - m_c[0];

    // 2) check that the triangle is "in front of" the ray:
    float t = - (Vec3::dot(m_normal, ray.origin()) + m_D) / Vec3::dot(m_normal, ray.direction());
    if (t <= 0.001){
        return result;
    }

    // 3) check that the intersection point is inside the triangle:
    // CONVENTION: compute u,v such that p = w0*c0 + w1*c1 + w2*c2, check that 0 <= w0,w1,w2 <= 1

    Vec3 intersection_point = ray.origin() + t * ray.direction();
    Vec3 local_point = intersection_point - m_c[0];

    float d00 = Vec3::dot(e1, e1);
    float d01 = Vec3::dot(e1, e2);
    float d11 = Vec3::dot(e2, e2);
    float d20 = Vec3::dot(local_point, e1);
    float d21 = Vec3::dot(local_point, e2);

    float denom = 1 / (d00 * d11 - d01 * d01);
    float v = (d11 * d20 - d01 * d21) * denom;
    float w = (d00 * d21 - d01 * d20) * denom;
    float u = 1.0f - v - w;

    // 4) Finally, if all conditions were met, then there is an intersection! :
    if (u >= 0 && v >= 0 && w >= 0 && u+v+w <= 1) {
        result.intersectionExists = true;
        result.t = t;
        result.intersection = intersection_point;
        result.normal = m_normal;
    }

    return result;
}
```

Figure 14: Intersection Triangle

Avec le fichier : https://seafile.lirmm.fr/d/63e408a83eea4e21b60a/files/?p=%2Fraptor_1M.off et un kdTree avec une profondeur maximale de 14 et 10 testes de splits a chaque fois, la construction du Kdtree prennait 3s.

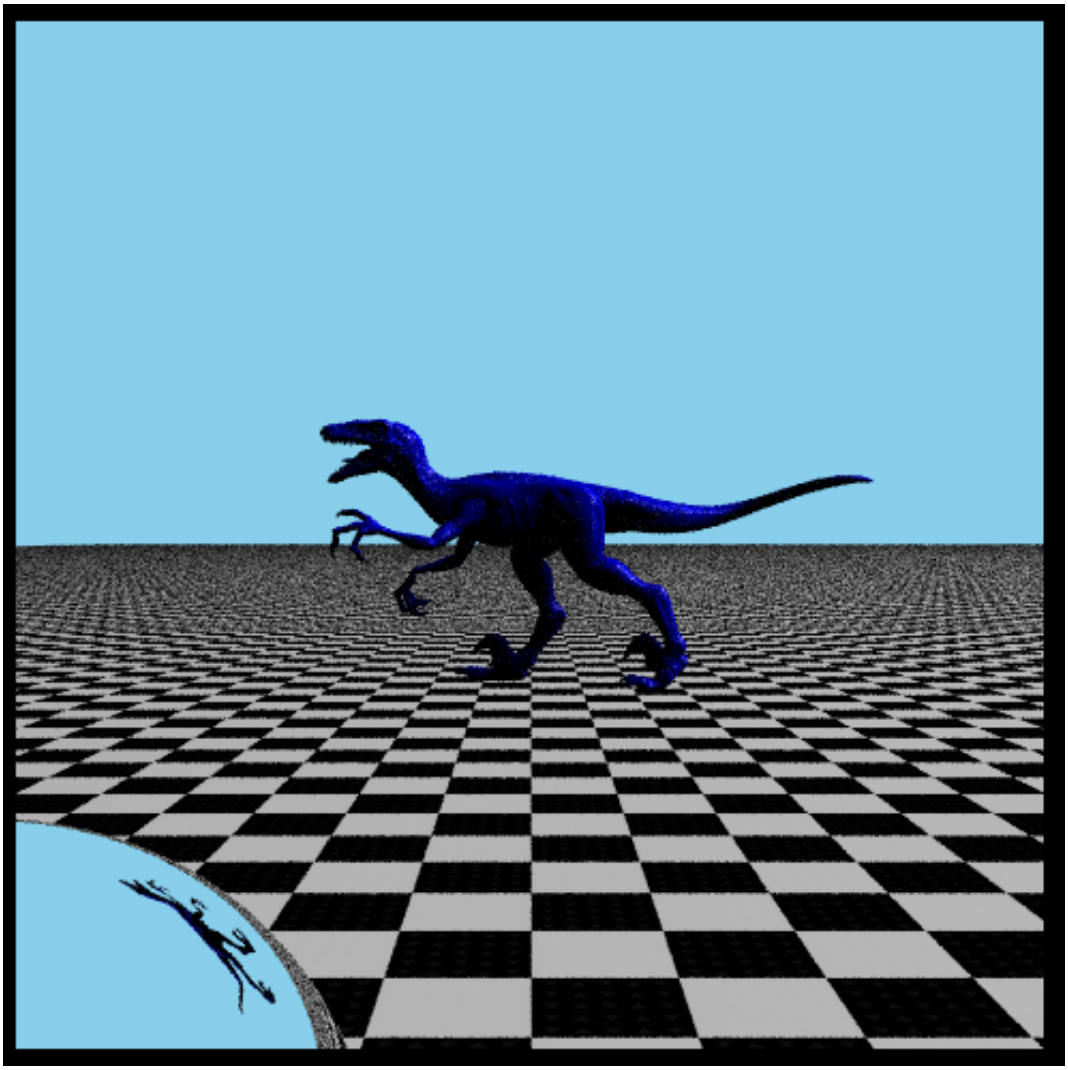


Figure 15: Rendu Mesh 1.7 M de triangles en 1.06 seconde (avec multi-threading)

6 Caméra : Depth of field

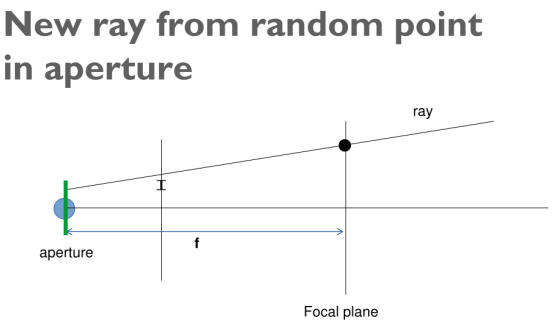
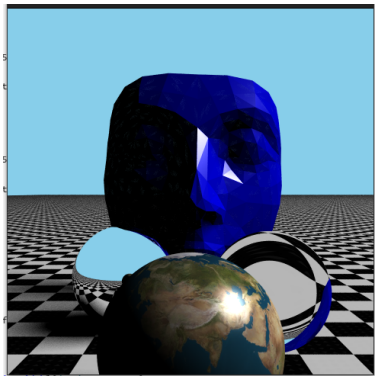


Figure 16: Ajout d'un offset pour créer un effet de profondeur de champ

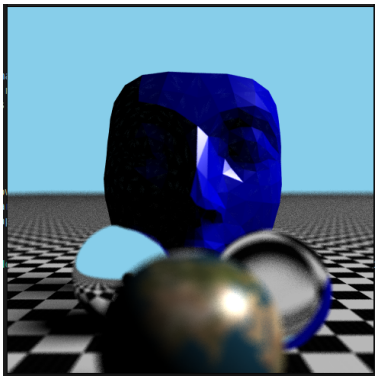
On calcule le point cible avec la position, la direction de base du rayon et la distance au plan focal. Ensuite, on ajoute un offset à la position et on recalcule la direction.

```
Ray depth_of_field_ray(float u, float v, float focal_plane_distance, float aperture_size, Vec3& position) {
    Vec3 direction;
    screen_space_to_world_space_ray_2(u, v, position, direction);

    Vec3 focal_point = position + direction * focal_plane_distance;
    Vec3 aperture_offset = random_in_disk(aperture_size);
    Vec3 new_origin = position + aperture_offset;
    Vec3 new_direction = focal_point - new_origin;
    new_direction.normalize();

    return Ray(new_origin, new_direction);
}
```

Figure 17: Ajout d'un offset pour créer un effet de profondeur de champ



(a) Image de base



(b) Distance focale = 10.0 et aperture size = 0.08

Figure 18: Classe KdTree

7 Photon Mapping

On lance des photons depuis les lumières de la scène. Ils ont comme couleur de base la couleur de la lumière d'où ils viennent. Quand ils intersectent un objet, leur couleur est multipliée à celle de l'objet touché, ensuite, il y a trois comportements possibles qu'on choisit avec la "roulette russe" :

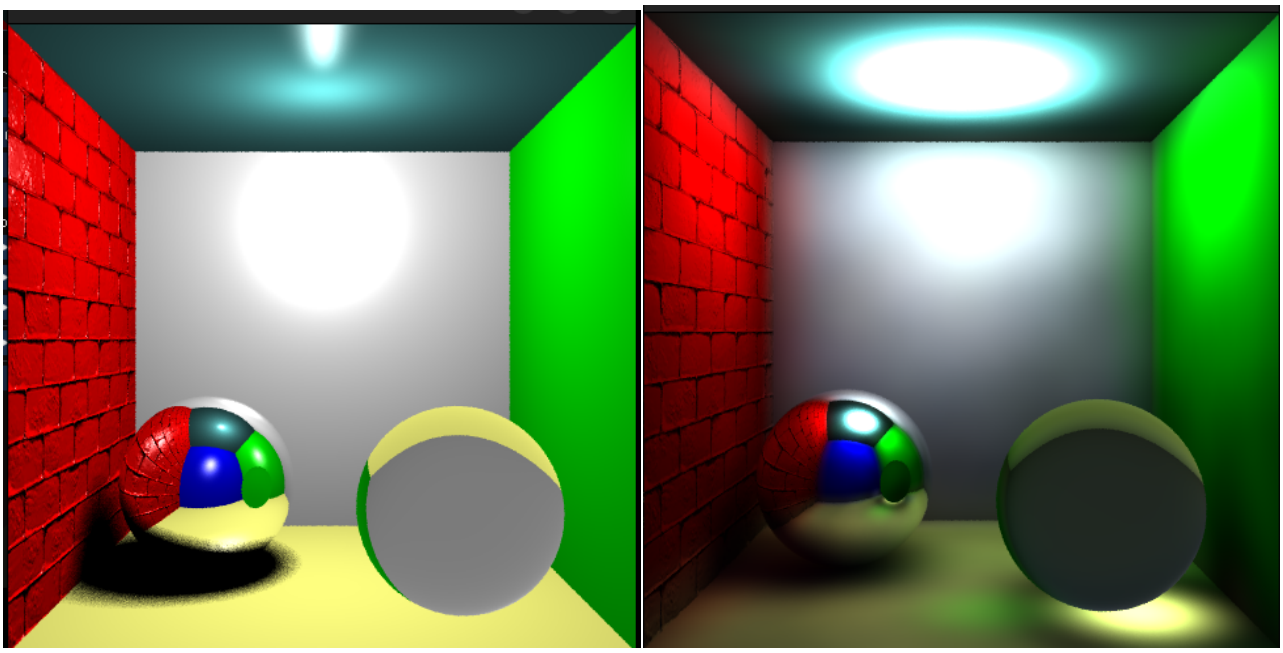
- Réfléchi si $x < mat.reflectivite$
- Réfracté si $x < mat.reflectivite + mat.transparence$
- Absorbé (On stocke notre photon) si $x >= mat.reflectivite + mat.transparence$

J'utilise un KdTree pour stocker les photons. J'ai utilisé la médiane de la position des photons sur la dimension à split (la plus grande).

```
struct Photon {
    Vec3 position;
    Vec3 direction;
    Vec3 color;
    int bounceCount;

    Photon(const Vec3& pos, const Vec3& dir, const Vec3& pow, int bounce = 0, float f = 0.0f): position(pos), direction(dir), color(pow), bounceCount(bounce) {}

    Photon() : position(Vec3(0,0,0)), direction(Vec3(0,0,0)), color(Vec3(0,0,0)), bounceCount(0) {}
};
```

Figure 19: Struct Photon

Quand on intersecte un objet, au lieu de récupérer sa couleur, on fait une moyenne des photons autour du point d'intersection. Le rayon est de base à 0.6 mais peut être modifié avec l'interface. La contribution d'un photon à la couleur finale est plus ou moins importante selon la distance avec le point d'intersection.



(a) Image de base

(b) En utilisant les photons

La lumière est plus réaliste, on observe qu'elle est réfractée en passant dans la sphère et cela créer des caustiques.

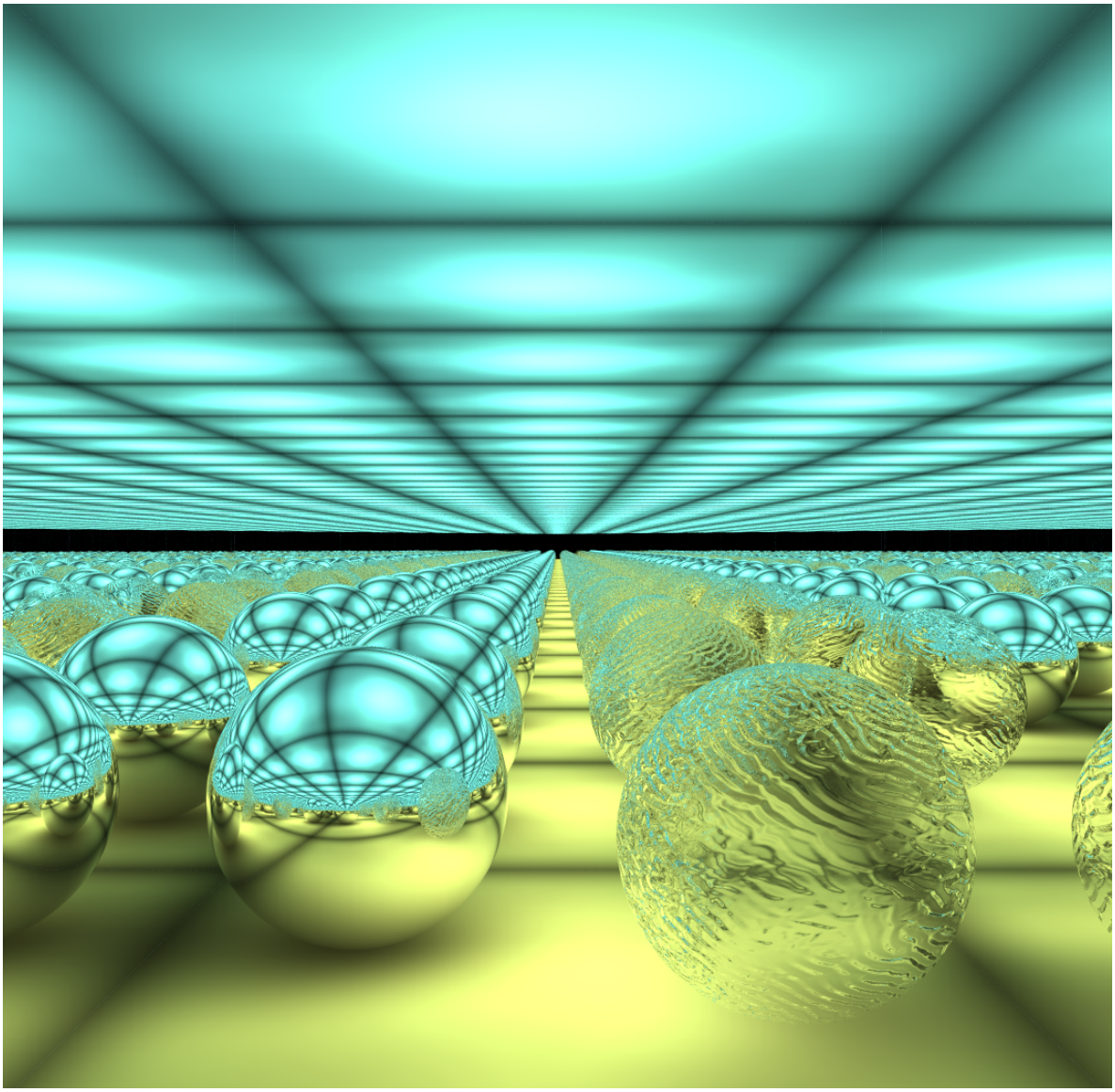


Figure 21: photon mapping avec murs réfléchissants, 30 rebonds, 1074x1051, rayon à 0.6 = 10 minutes de rendu