

La place de la radiosit  dans la qu te du r alisme virtuel

B TOUS Marc (n 14959)

avec BA ZA Mika (n 13891)

2023 - 2024

Jeux et sports

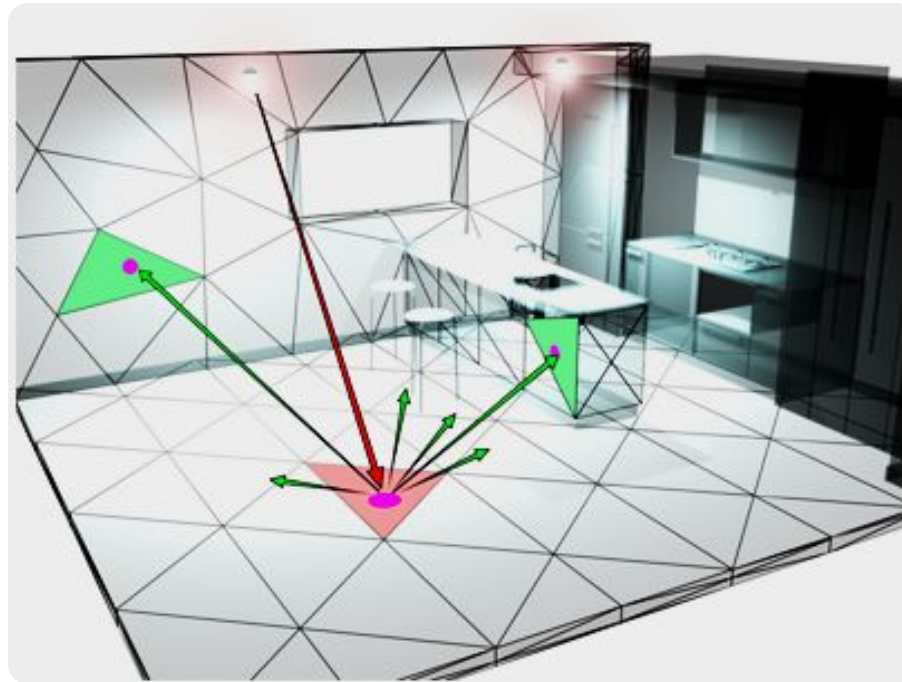


Wikipedia

L'importance de la lumière dans une scène

Sommaire

1. Explication de la radiosité
2. L'accélération du temps de calcul
3. Le post processing



Autodesk


$$D_i = \sum_{j=1}^n (D_j \times C_{ij}) + E_i$$

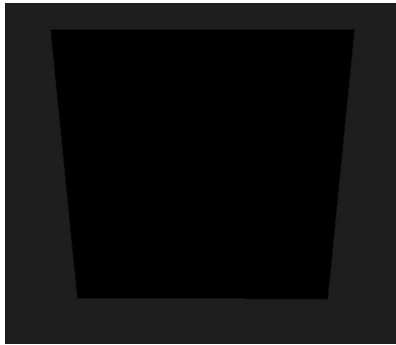
Équation de radiosité :

$$\begin{pmatrix} D_1 \\ \dots \\ D_n \end{pmatrix} = \begin{pmatrix} C_{11} & \dots & C_{1n} \\ \dots & \dots & \dots \\ C_{n1} & \dots & C_{nn} \end{pmatrix} \times \begin{pmatrix} D_1 \\ \dots \\ D_n \end{pmatrix} + \begin{pmatrix} E_1 \\ \dots \\ E_n \end{pmatrix}$$

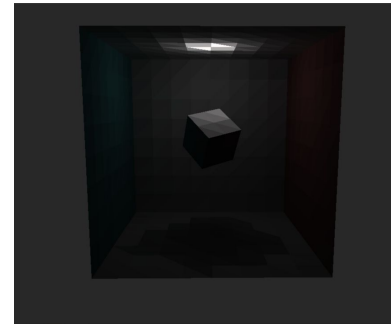
$$\Leftrightarrow D = C \times D + E$$

On converge vers la solution par récurrence : $D^{(n+1)} = C \times D^{(n)} + E$

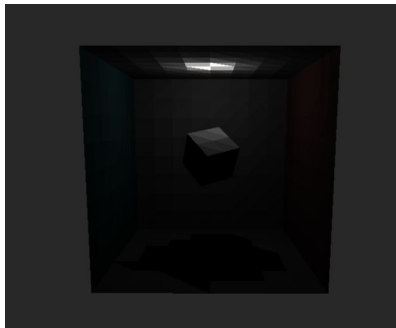
- Subdivision des faces.
- Initialisation des vecteurs E et D_0 .
- Calcul de la matrice C .
- Calcul de la récurrence sur D . 
- Application des couleurs aux faces.
- Lancement du moteur 3d.



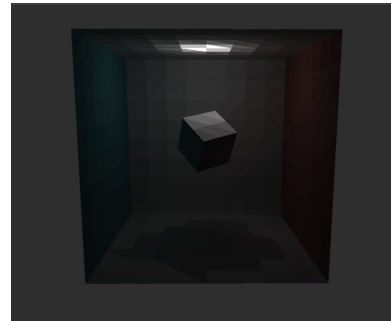
Affichage de $D^{(0)}$



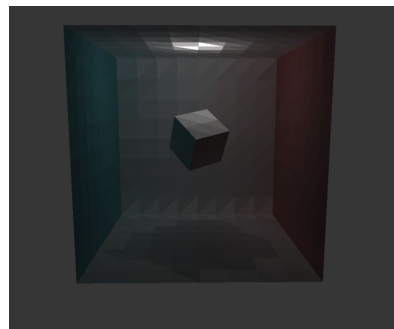
Affichage de $D^{(2)}$



Affichage de $D^{(1)}$



Affichage de $D^{(3)}$



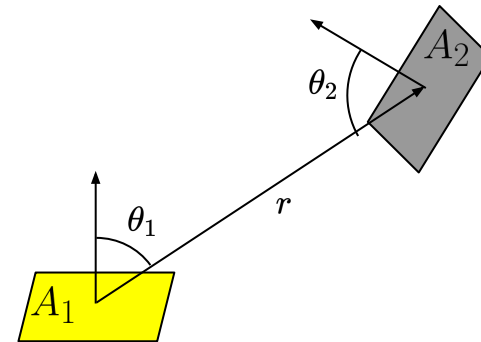
Affichage de $D^{(100)}$

Approximation de la solution D

```
C = calculMatriceC();  
D = vecteurVide();  
E = vecteurLumièreIntrinseque();  
tant que le vecteur D varie :  
    | D = C × D + E;
```


Formule des facteurs de forme :

$$C_{12} = \frac{\cos(\theta_1) \times \cos(\theta_2)}{\pi r^2} \times A_2$$



Matrice C contenant les facteurs de forme

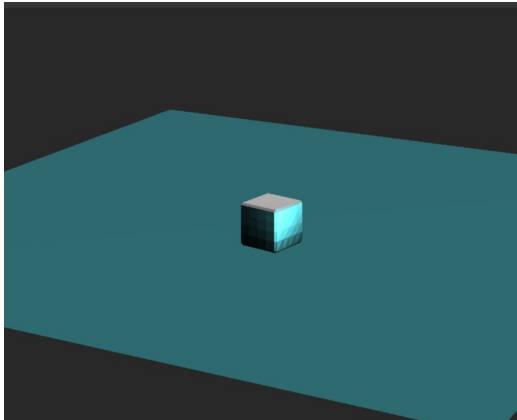
$$\begin{pmatrix} C_{11} & C_{12} & \dots & C_{1n} \\ C_{21} & C_{22} & \dots & C_{2n} \\ \dots & \dots & \dots & \dots \\ C_{n1} & C_{n2} & \dots & C_{nn} \end{pmatrix}$$

Remplissage de la matrice C

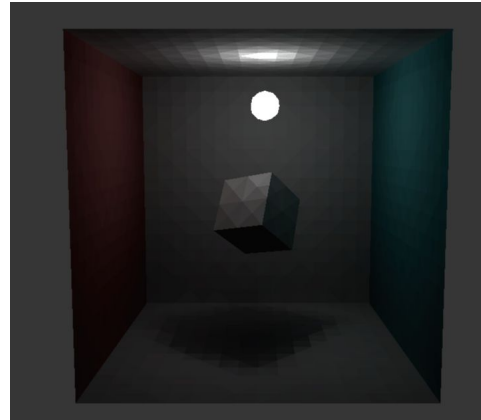
```
Pour chaque face i :  
  | Pour chaque face j :  
  |   | Pour chaque face k:  
  |   |   | si la face k intersecte le rayon entre la face i et j :  
  |   |   |   | C[i, j] = 0;  
  |   |   | sinon:  
  |   |   |   | C[i, j] = CalculFacteurForme(i,j);
```

Complexité en $O(n^3)$

Scène à 304 faces
(Scène 1)



Scène à 2639 faces
(Scène 2)



Scène à 5367 faces
(Scène 3)



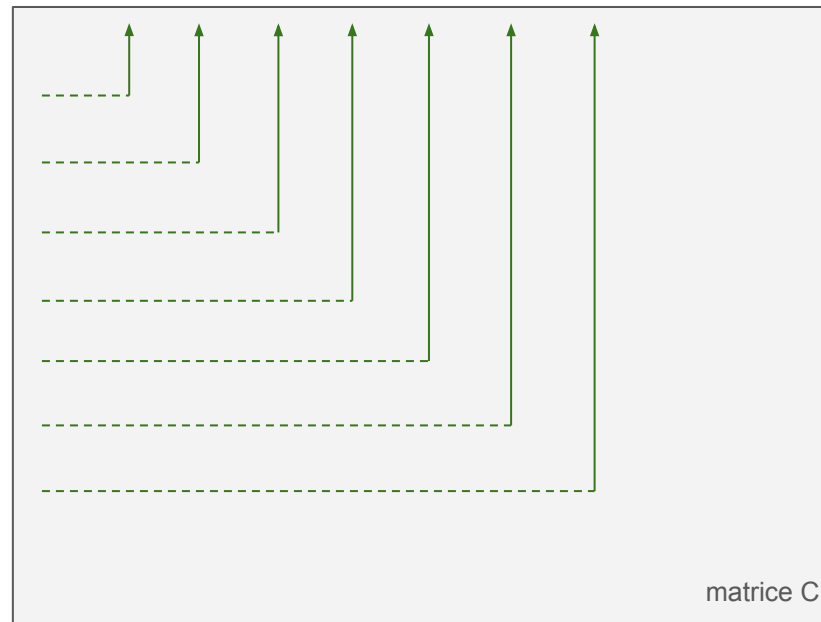
Temps de calcul :

~ 1 minutes

~ 10 heures

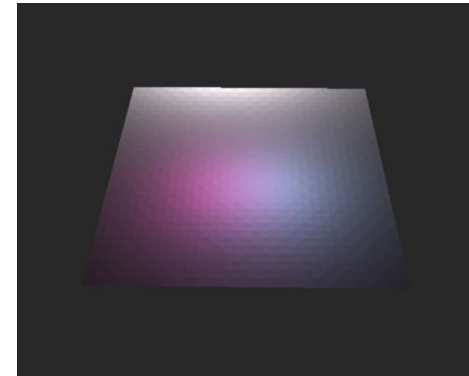
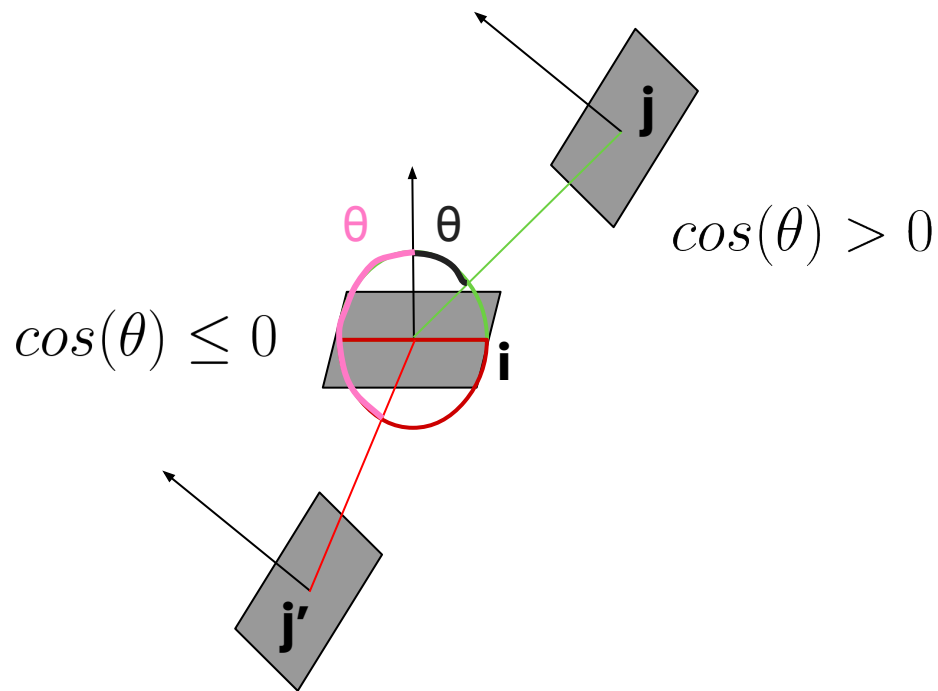
~ ?

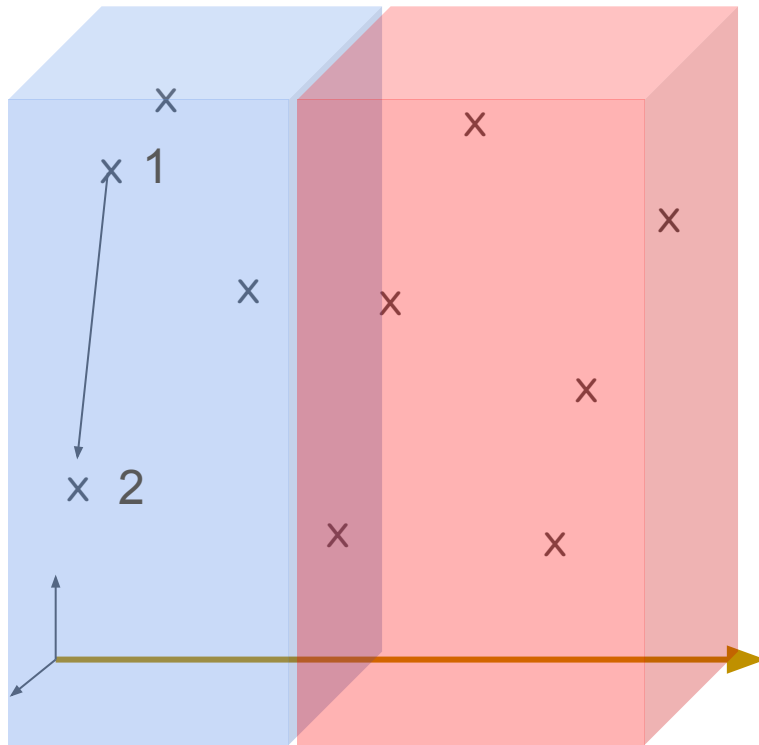
$$C_{12} = \frac{\cos(\theta_1) \times \cos(\theta_2)}{\pi r^2} \times A_2 \quad C_{21} = \frac{\cos(\theta_2) \times \cos(\theta_1)}{\pi r^2} \times A_1$$



Temps de calcul divisé par 2

Ignorer des calculs inutiles





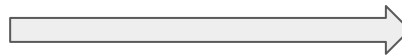
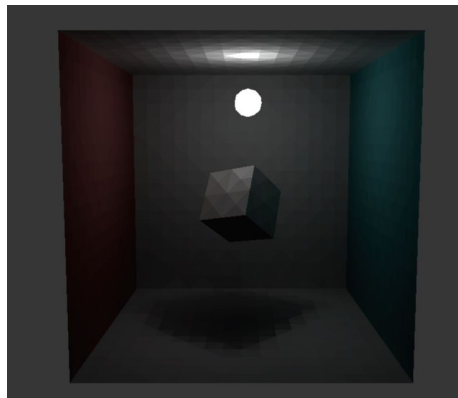
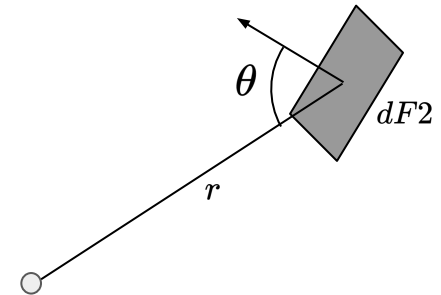
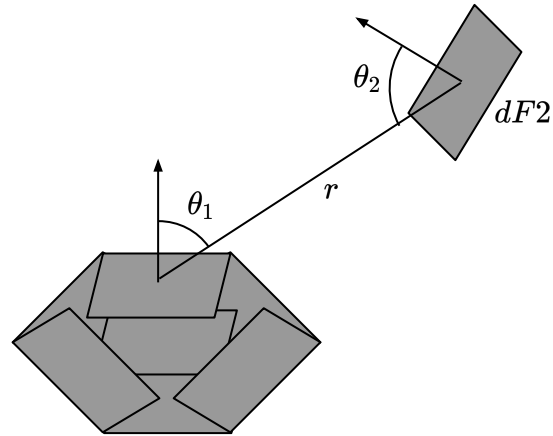
Points à gauche

~~Points à droite~~

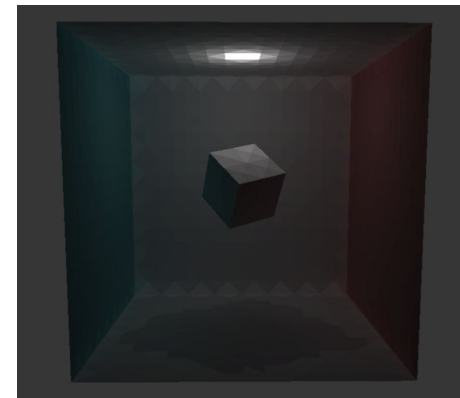
On ignore la moitié des points.

$$O(n) \rightarrow O(\log(n))$$

| | Scène 1 | Scène 2 | Scène 3 |
|--------------------|---------|---------|---------|
| Nombre de faces | 304 | 2600 | 5367 |
| hauteur de l'arbre | 9 | 12 | 13 |



Élimination de 80 faces



Deux possibilités pour le multithreading :

Matrice C

$$\begin{pmatrix} C_{11} & C_{12} & \dots & C_{1n} \\ C_{21} & C_{22} & \dots & C_{2n} \\ \dots & \dots & \dots & \dots \\ C_{n1} & C_{n2} & \dots & C_{nn} \end{pmatrix}$$

Équation de radiosit 

$$\begin{pmatrix} D_1 \\ D_2 \\ \dots \\ D_n \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^n D_i \times C_{1i} + E_1 \\ \sum_{i=1}^n D_i \times C_{2i} + E_2 \\ \dots \\ \sum_{i=1}^n D_i \times C_{ni} + E_n \end{pmatrix}$$

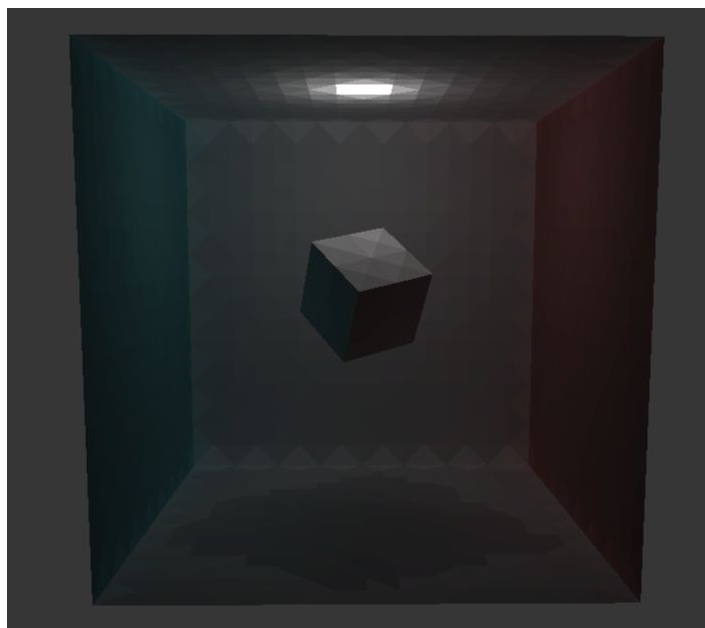
Calcul de la matrice C

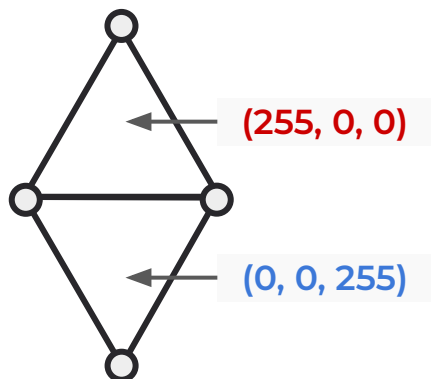
| | Scène 1 | Scène 2 | Scène 3 |
|---|-------------------|----------------------|------------------------|
| Nombre de faces | 304 | 2639 | 5367 |
| Temps de calcul sans parallélisation | 5,3 millisecondes | 3 minutes 6 secondes | 22 minutes 50 secondes |
| Temps de calcul avec parallélisation (12 cœurs) | 2,8 millisecondes | 22 secondes | 4 minutes 23 secondes |

Calcul de l'équation de radiosit 

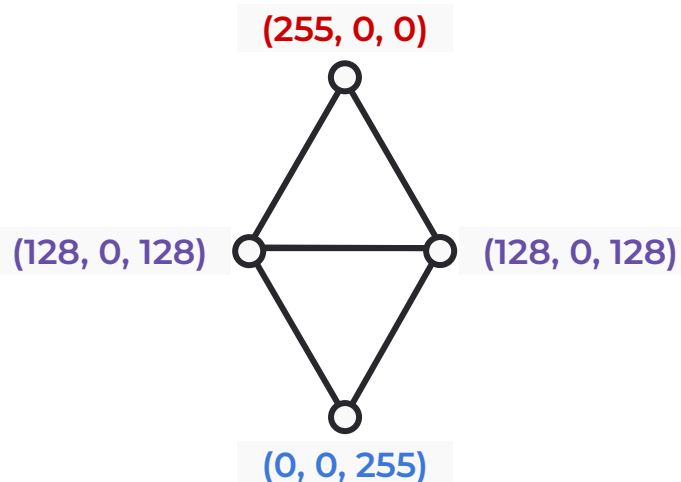
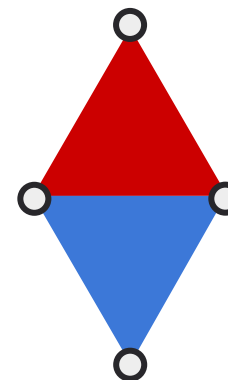
| | Sc ne 1 | Sc ne 2 | Sc ne 3 |
|---|------------------|------------|-------------|
| Nombre de faces | 304 | 2639 | 5367 |
| Temps de calcul sans parall lisation | 61 millisecondes | 6 secondes | 26 secondes |
| Temps de calcul avec parall lisation (12 c urs) | 89 millisecondes | 7 secondes | 24 secondes |

Nos scènes manquent de réalisme

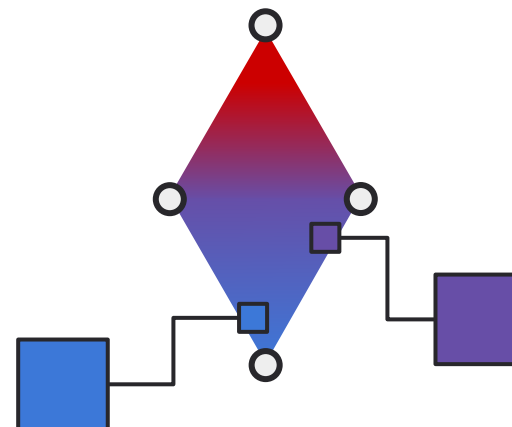


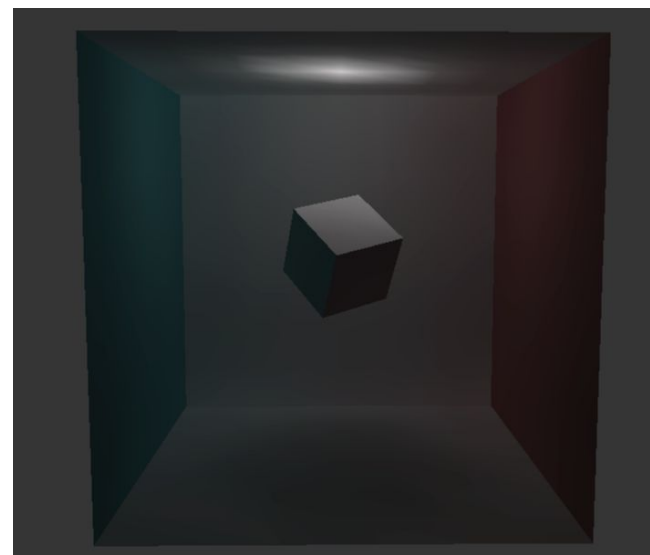
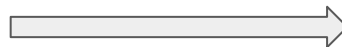
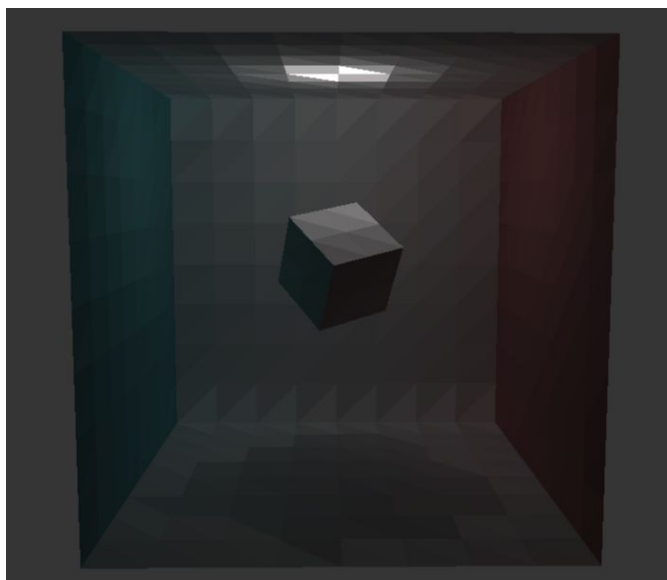


→
Moteur 3d

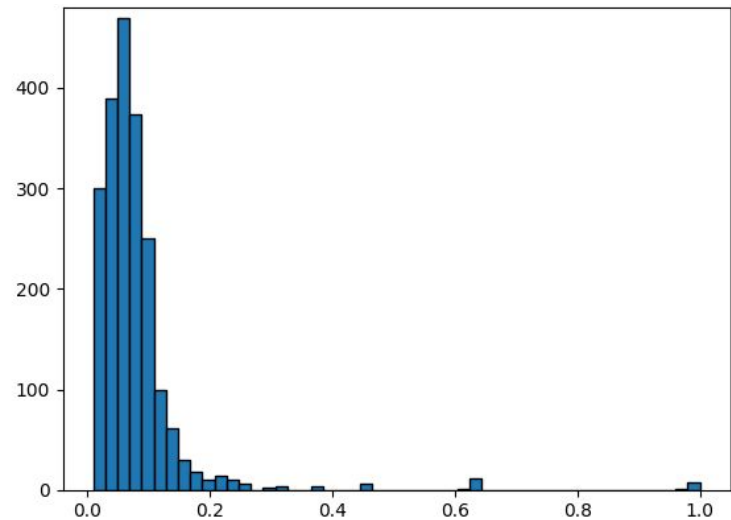
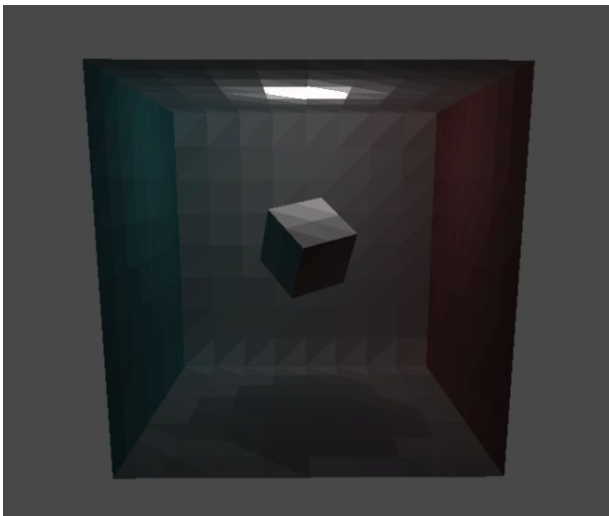


→
Moteur 3d





Un manque de contraste irréaliste

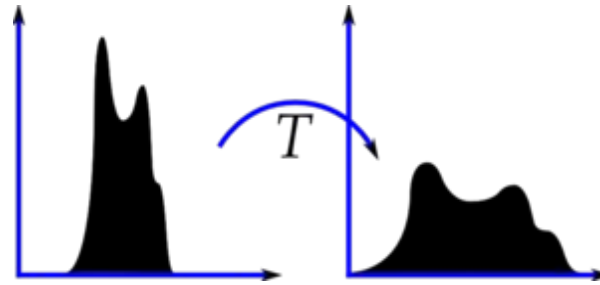


Nombre de faces par tranche de luminosité

Maximum : 478 faces (entre 0,04 et 0,08 de luminosité)

Moyenne : 0,08 de luminosité

Écart-type : 0,088



$$Eq(color) = \alpha \times color + \beta$$

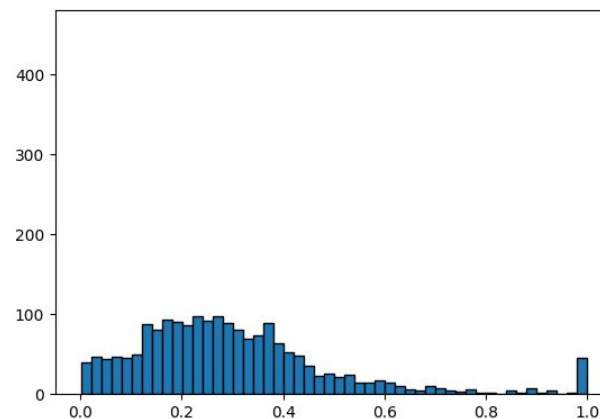
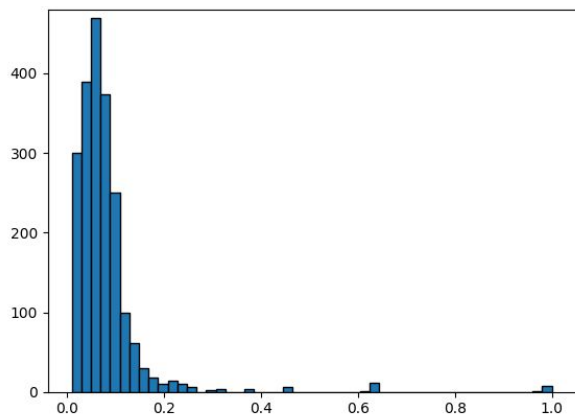
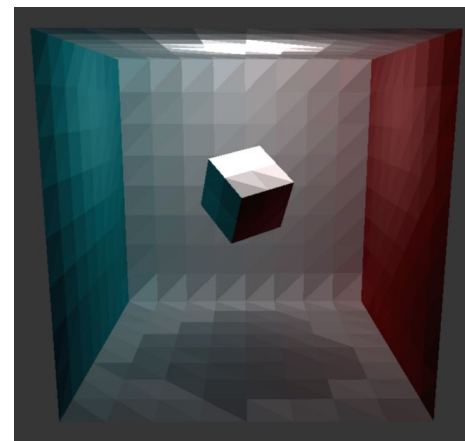
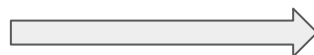
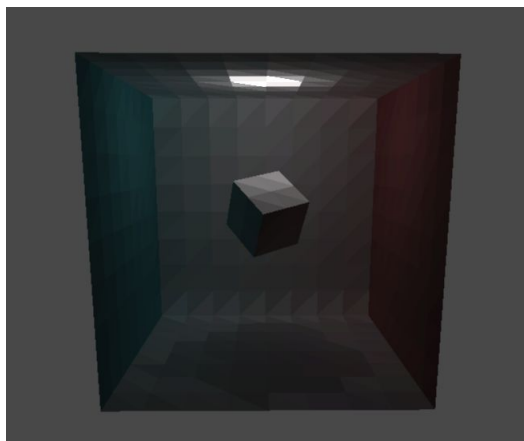


troncature du tableau



$$\alpha = \frac{1}{\max(gray) - \min(gray)}$$

$$\beta = -\min(gray) \times \alpha$$



Maximum : 101 faces (entre 0,22 et 0,24 de luminosité)

Moyenne : 0,30 de luminosité

Écart-type : 0,20

Une technologie inutilisable en temps réel, mais utile dans certaines circonstances.

archi-visualisation3d



Utilisation dans des logiciels de rendu d'architecture

League of Legends



Utilisation pour des cinématiques pré-rendues de jeux vidéos

On a, $\forall i \in [1, n], |a_{i,i}| \geq \sum_{\substack{j=1 \\ j \neq i}}^n |a_{i,j}| \Rightarrow$ La suite (D^k) converge vers l'unique solution de $AD = E$

$$D = C \times D + E$$

$$\Leftrightarrow (Id - C) \times D = E$$

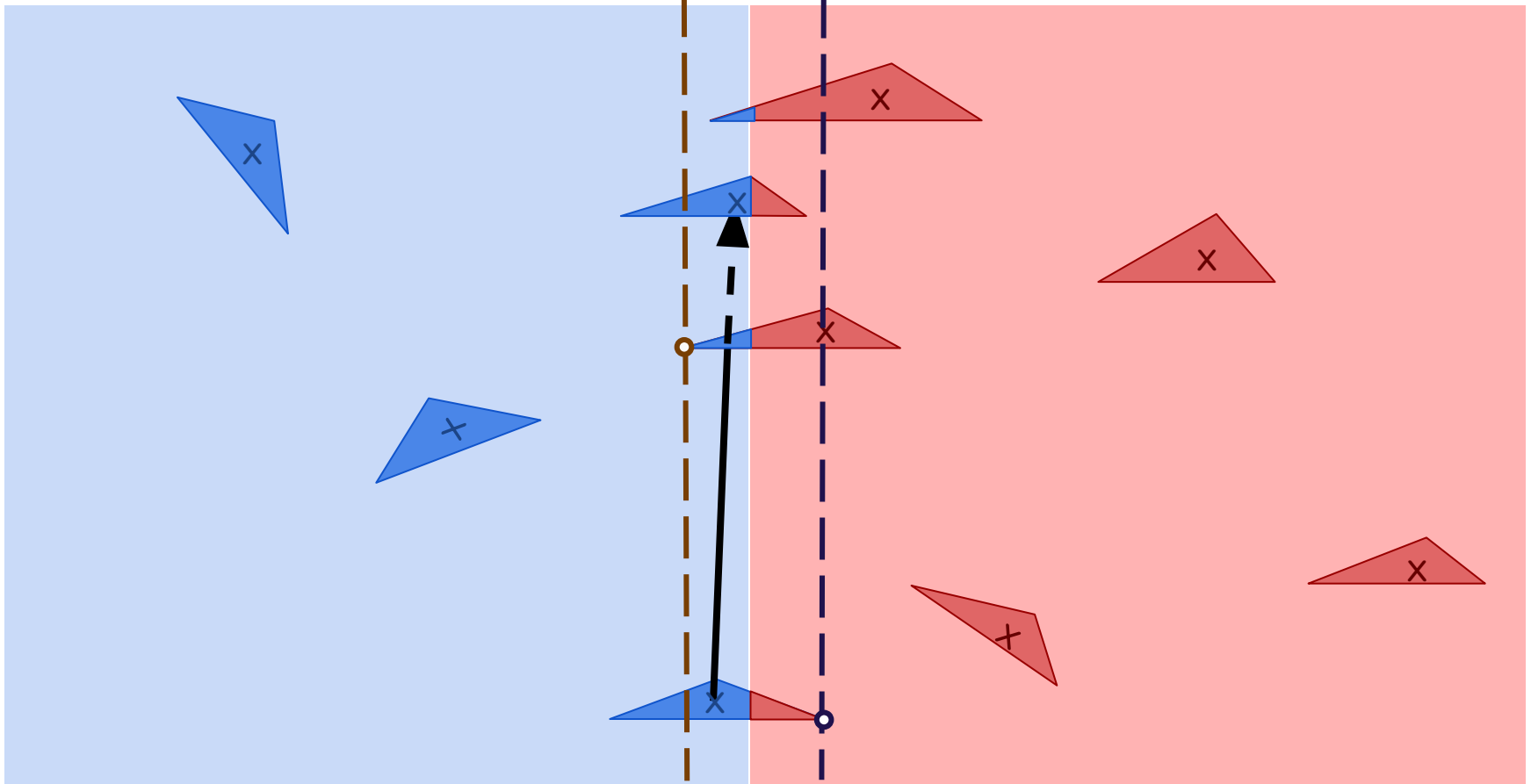
On pose $A = Id - C$

On a, $\forall i \in [1, n], |a_{i,i}| \geq \sum_{\substack{j=1 \\ j \neq i}}^n |a_{i,j}| \Rightarrow$ La suite (D^k) converge vers l'unique solution de $AD = E$

$$\text{Ici, } A = \begin{pmatrix} 1 & -C_{12} & \dots & -C_{1n} \\ -C_{21} & 1 & \dots & -C_{2n} \\ \dots & \dots & \dots & \dots \\ C_{n1} & -C_{n2} & \dots & 1 \end{pmatrix} \text{ et } \sum_{j=1}^n C_{ij} = 1$$

Noeud

marge gauche : 0.2
marge droite : 0.3



BSP.h

```
#pragma once

#include <iostream>
#include <vector>
#include "world.h"

#define FACE_PTR Face*
#define BSP_ERROR 0.2

class BinaryTree
{
public:
    Face* face;
    int axis;
    BinaryTree* left;
    BinaryTree* right;
    float margin_left; // Marge d'erreur à gauche (distance du plan parent à laquelle une des faces à droite peut intervenir).
    float margin_right; // Marge d'erreur à droite (distance du plan parent à laquelle une des faces à gauche peut intervenir).
    BinaryTree(vector<Face*>& tab, int min, int max);
};

// Compare le centre de %facel et de %face2
bool cmp_face(Face* facel, Face* face2, int compo);

// Compare %facel et %face2
bool cmp_face(int facel, int face2, int compo);

// Échange dans %tab les valeurs d'indice %i1 et %i2
void swap(vector<FACE_PTR>& tab, int i1, int i2);

// Affiche dans le terminal les valeurs de %tab
void print_vect(vector<int> tab);

/* Renvoie x, la position à laquelle serait %tab[%pivotIndex] si le tableau était trié en partitionnant le sous tableau entre les bornes
%min et %max tel que:
pour i <= x tab[i] <= tab[x]
pour i > x tab[i] > tab[x]
*/
int partitioning(vector<Face*> & tab, int min, int max, int pivotIndex, int compo);
```

```

/* place l'élément en position %r dans %tab en ne regardant que entre les bornes %min à %max, en comparant selon la composante %compo
*/
void setPositionR(vector<Face*> & tab, int min, int max, int r, int compo);

/* Place dans %tab la median au bon endroit si ils étaient triés selon la coordonnée n°%coord */
void setMedianIndex(vector<FACE_PTR> & tab, int min, int max, int coord);

// Renvoie la composante selon laquelle il est plus avantageux de couper dans le sous-tableau de %tab: [%min, %max]
int chooseCuttingAxis(vector<FACE_PTR> & tab, int min, int max);

/* Créé un arbre de partitionnage binaire des faces a partir de la %scene */
BinaryTree * buildBinaryTree(Scene & scene);

/* Créé un arbre de partitionnage binaire des faces a partir de la %list */
BinaryTree * buildBinaryTreeFromList(vector<Face*> list);

// Fonction auxiliaire de getNeighborInRadius permettant les itérations récursives
void rec_getNeighborInRadius(BinaryTree * binaryTree, FACE_PTR elem, float radius, vector<FACE_PTR> & neighborList);

/* Renvoie un vecteur des éléments présents dans un %radius autour de %elem ne utilisant l'arbre k-dimensionnel %binaryTree */
vector<Face*> getNeighborInRadius(BinaryTree * binaryTree, Face * elem, float radius);

// Renvoie le coefficient du point d'intersection entre le rayon (%origin, %direction) et le plan de séparation représenté par %node.
float intersectionPlan(Vector3f origin, Vector3f direction, BinaryTree * node);

// Renvoie la distance entre le rayon (%origin, %direction) et le plan de séparation représenté par %node
float distanceRayPlan(Vector3f origin, Vector3f direction, BinaryTree * node);

// Fonction auxiliaire de rayIsIntersectedInScene permettant les itérations récursives.
bool rec_rayIsIntersected(BinaryTree * binaryTree, Vector3f origin, Vector3f direction, BinaryTree * parent);

// Renvoie si le rayon d'origine %origin et de direction %direction est intersecté par une des faces dans la %scene.
bool rayIsIntersectedInScene(Vector3f origin, Vector3f direction, Scene * scene);

// Renvoie si le rayon d'origine %origin et de direction %direction est intersecté par une des faces dans %bt.
bool rayIsIntersectedInBinaryTree(BinaryTree * bt, Vector3f origin, Vector3f direction);

```

BSP.cpp

```
/* Bipartition de l'espace */
#include "world.h"
#include <stdlib.h>
#include <stack>

#include "BSP.h"

bool cmp_face(Face * face1, Face * face2, int compo)
{
    if (face1->center()(compo) <= face2->center()(compo))
    {
        return true;
    }
    return false;
}

bool cmp_face(int face1, int face2, int compo)
{
    return face1 <= face2;
}

void swap(vector<FACE_PTR> & tab, int i1, int i2)
{
    FACE_PTR temp = tab[i1];
    tab[i1] = tab[i2];
    tab[i2] = temp;
}

void print_vect(vector< int> tab)
{
    for (int i = 0; i < tab.size(); i++)
    {
        printf("%d ", tab[i]);
    }
    printf("\n");
}
```

```

int partitioning(vector<FACE_PTR> & tab, int min, int max, int pivotIndex, int compo)
{
    assert(min<=max && max < tab.size() && min>= 0);
    assert(min<=pivotIndex && pivotIndex<=max);
    int j = min;
    swap(tab, pivotIndex, max);
    for(int i =min; i<max; i++)
    {
        if (cmp_face(tab[i], tab[max], compo)) //tab[i] > tab[j]
        {
            swap(tab, i, j);
            j++;
        }
    }
    swap(tab, (j), max);
    return (j);
}

```

```

void setPositionR(vector<FACE_PTR> & tab, int min, int max, int r, int compo)
{
    assert(min<=max && max < tab.size() && min>= 0);
    assert(min<=r && r<=max);
    if (min == max) // Cas limite
    {
        tab[min];
    } else {
        srand(time( NULL));
        int pivot = (rand()% (max-min)) + min;
        int candidat = partitioning(tab, min, max, pivot, compo);
        if (r < candidat)
        {
            setPositionR(tab, min, candidat- 1, r, compo);
        }
        else if (r > candidat) {
            setPositionR(tab, candidat+ 1, max, r, compo);
        }
    }
}

```

```

void setMedianIndex(vector<FACE_PTR> & tab, int min, int max, int coord)
{
    assert(min<=max && max < tab.size() && min>= 0);
    return setPositionR(tab, min, max, (min+max)/ 2, coord);
}

int chooseCuttingAxis(vector<FACE_PTR> & tab, int min, int max)
{
    float spr_x[2] = {INFINITY,-INFINITY};
    float spr_y[2] = {INFINITY,-INFINITY};
    float spr_z[2] = {INFINITY,-INFINITY};
    for (int i = min; i <= max; i++)
    {
        spr_x[0] = std::min(spr_x[0], tab[i]->center()(0));
        spr_x[1] = std::max(spr_x[1], tab[i]->center()(0));
        spr_y[0] = std::min(spr_y[0], tab[i]->center()(1));
        spr_y[1] = std::max(spr_y[1], tab[i]->center()(1));
        spr_z[0] = std::min(spr_z[0], tab[i]->center()(2));
        spr_z[1] = std::max(spr_z[1], tab[i]->center()(2));
    }
    if (spr_x[1]-spr_x[0] >= spr_y[1]-spr_y[0] && spr_x[1]-spr_x[0] >= spr_z[1]-spr_z[0]){
        return 0;
    }
    if (spr_y[1]-spr_y[0] >= spr_x[1]-spr_x[0] && spr_y[1]-spr_y[0] >= spr_z[1]-spr_z[0]){
        return 1;
    }
    if (spr_z[1]-spr_z[0] >= spr_y[1]-spr_y[0] && spr_z[1]-spr_z[0] >= spr_x[1]-spr_x[0]){
        return 2;
    }
    assert(false);
    return -1;
}

```



```

BinaryTree::BinaryTree(vector<FACE_PTR>& tab,int min, int max)
{
    assert(max < tab.size() && min>=0 && min<=max);
    axis = chooseCuttingAxis(tab, min, max);
    left = nullptr;
    right = nullptr;
    if(min == max)
    {
        // On est dans une feuille
        face = tab[min];
    }
    else // On est sûr que min < max
    {
        int median = (max+min)/2;
        setMedianIndex(tab, min, max, axis);
        face = tab[median];

        if(min<median){
            left = new BinaryTree(tab, min, median-1);
        }
        if(max>median){
            right = new BinaryTree(tab, median+1, max);
        }
        // Calcul des marges d'erreur
        float error = -1;
        float separator = face->center()(axis);
        for (int i = min; i <= median; i++)
        {
            error = std::max({error, tab[i]->vertex0()(axis)-separator, tab[i]->vertex1()(axis)-separator, tab[i]->vertex2()(axis)-separator});
        }
        // assert(error >=0);
        margin_right = error +0.001; // L'inversion "for" et "margin_right" est normal: on regarde à gauche qui est-ce qui déplace le plus sur la partie droite.
        error = -1;
        for (int i = median; i <= max; i++)
        {
            error = std::max({error, separator - tab[i]->vertex0()(axis), separator-tab[i]->vertex1()(axis), separator-tab[i]->vertex2()(axis)});
        }
        // assert(error >=0);
        margin_left = error +0.001; // Ditto
    }
}
}

```

```

BinaryTree* buildBinaryTree(Scene & scene)
{
    vector<FACE_PTR> facePtrList = scene.getFacesPointer();
    BinaryTree* binaryTree = new BinaryTree(facePtrList, 0, facePtrList.size()-1);
    return binaryTree;
};

BinaryTree* buildBinaryTreeFromList(vector<FACE_PTR> list)
{
    BinaryTree* binaryTree = new BinaryTree(list, 0, list.size()-1);
    return binaryTree;
};

oid rec_getNeighborInRadius(BinaryTree * binaryTree, FACE_PTR elem, float radius, vector<FACE_PTR> & neighborList)
{
    if (binaryTree == nullptr)
    {
        return;
    }
    FACE_PTR m = binaryTree->face;
    int axis = binaryTree->axis;
    if(elem->center()(axis) <= m->center()(axis)) // On plonge à gauche
    {
        rec_getNeighborInRadius(binaryTree->left, elem, radius, neighborList);
        if(abs(elem->center()(axis) - m->center()(axis)) < radius) // Si besoins aussi à droite
        {
            rec_getNeighborInRadius(binaryTree->right, elem, radius, neighborList);
        }
    }
    else // On plonge à droite
    {
        rec_getNeighborInRadius(binaryTree->right, elem, radius, neighborList);
        if(abs(elem->center()(axis) - m->center()(axis)) < radius) // Si besoins aussi à gauche
        {
            rec_getNeighborInRadius(binaryTree->left, elem, radius, neighborList);
        }
    }
    if(distance(elem->center(), m->center()) <= radius)
    {
        neighborList.push_back(m);
    }
}

```

```

vector<FACE_PTR> getNeighborInRadius(BinaryTree * binaryTree, FACE_PTR elem, float radius)
{
    vector<FACE_PTR> facePtrList;
    rec_getNeighborInRadius(binaryTree, elem, radius, facePtrList);
    return facePtrList;
}

float intersectionPlan(Vector3f origin, Vector3f direction, BinaryTree * node)
{
    float t = direction[node->axis]; // Produit scalaire normale du plan et direction du rayon
    if (abs(t) < 0.00001) return INFINITY; // Si il est parallèle on renvoie l'infini.
    return (node->face->center()[node->axis] - origin(node->axis))/t;
}

float distanceRayPlan(Vector3f origin, Vector3f direction, BinaryTree * node){
    float dist1 = origin(node->axis) - node->face->center()(node->axis);
    float dist2 = (origin(node->axis)+direction(node->axis)) - node->face->center()(node->axis);
    if ((dist1 > 0 && dist2 < 0) || (dist1 < 0 && dist2 > 0)) return 0; // Le rayon traverse la face
    return min(abs(dist1), abs(dist2));
}

```

```

bool rec_rayIsIntersected(BinaryTree * binaryTree, Vector3f origin, Vector3f direction, BinaryTree * parent)
{
    if(binaryTree == nullptr) return false;

    BinaryTree* mainChild = nullptr;
    BinaryTree* secondaryChild = nullptr;
    float margin;
    if(origin[binaryTree->axis] < binaryTree->face->center()[binaryTree->axis]) // L'origine du rayon est à gauche.
    {
        mainChild = binaryTree->left;
        margin = binaryTree->margin_left;
        secondaryChild = binaryTree->right;
    }
    else
    {
        mainChild = binaryTree->right;
        margin = binaryTree->margin_right;
        secondaryChild = binaryTree->left;
    }
    if(rec_rayIsIntersected(mainChild, origin, direction, binaryTree)) // On cherche une intersection du coté principal
    {
        return true;
    }
    else
    {
        if (parent == nullptr) // On est en haut de l'arbre, on doit forcément aller de l'autre coté
        {
            if (binaryTree->face->isIntersectedByRay(origin, direction))
            {
                return true;
            }
            else
            {
                return rec_rayIsIntersected(secondaryChild, origin, direction, binaryTree);
            }
        }
        else // Il faut faire un choix: remonter au parent ou aller de l'autre coté ?
        {
            float coefParent = intersectionPlan(origin, direction, parent);
            float coefSelf = intersectionPlan(origin, direction, binaryTree);
            float minDistance = distanceRayPlan(origin, direction, binaryTree);
            if (minDistance <= margin || coefSelf < coefParent) // Si le rayon est trop près de l'autre coté ou il va vraiment de l'autre coté
            {
                if(binaryTree->face->isIntersectedByRay(origin, direction)) // d'abord on regarde notre face séparatrice
                {
                    return true;
                }
                else
                {
                    if((coefSelf >= 1 || coefSelf <= 0) && minDistance > margin) return false; // Le rayon n'atteint en réalité pas l'autre partie et il est trop loin
                    return rec_rayIsIntersected(secondaryChild, origin, direction, binaryTree); // Bon... bah on va aller regarder de l'autre coté
                }
            }
            else // Le rayon atteint le plan parent avant notre plan et il est trop loin de l'autre coté: on remonte
            {
                return binaryTree->face->isIntersectedByRay(origin, direction);
            }
        }
    }
}
}

```

```
bool rayIsIntersectedInScene(Vector3f origin, Vector3f direction, Scene * scene)
{
    for (Object& object : scene->getObjects())
    {
        if (object.boundingBox->isInterectedByRay(origin, direction)) {
            if(rec_rayIsIntersected(object.binaryTree, origin, direction, nullptr)){
                return true;
            }
        }
    }
    return false;
}

bool rayIsIntersectedInBinaryTree(BinaryTree * bt, Vector3f origin, Vector3f direction)
{
    if(rec_rayIsIntersected(bt, origin, direction, nullptr)){
        return true;
    }
    return false;
}
```

chiffre.py

```
import matplotlib.pyplot as plt
f = open("values2.txt", "r")
valR = []
valB = []
valG = []
val = f.readline().strip()
print(val)
while val != "":
    valR.append(float(val))
    val = f.readline().strip()
val = f.readline().strip()
while val != "":
    valB.append(float(val))
    val = f.readline().strip()
val = f.readline().strip()
while val != "":
    valG.append(float(val))
    val = f.readline().strip()

plt.title("V2")
plt.hist(valB, bins = 50, edgecolor = "black")
plt.show()
```

lighting.h

```
#pragma once
#include "world.h"

typedef Eigen::Matrix<bool, Eigen::Dynamic, Eigen::Dynamic> MatrixXb;

// Classe des matrices D, C et E (3 canaux de lumières: rouge, vert, bleu)
class LightMatrix
{
public:
    MatrixXf red;
    MatrixXf green;
    MatrixXf blue;
    LightMatrix(MatrixXf matrix);
};

// Structure des données nécessaires au multithreading pour le calcul de la matrice C
struct arg_s
{
    int* id_todo;
    int thread_id;
    BinaryTree* binaryTree;
    vector<Face*>* faceList;
    LightMatrix* C;
    MatrixXb* hasToCalcul;
    Scene* scene;
};

//Overload de l'opérateur * pour les matrices
LightMatrix operator* (const LightMatrix & x, const LightMatrix & y);

//Overload de l'opérateur + pour les matrices
LightMatrix operator+ ( const LightMatrix & x, const LightMatrix & y);

//Overload de l'opérateur / pour les matrices
LightMatrix operator/ ( const LightMatrix & x, const float y);

//Getter de la composante bleue d'une matrice
vector<float> getBlue(LightMatrix * m);
```

```

//Getter de la composante rouge d'une matrice
vector<float> getRed(LightMatrix* m);

//Getter de la composante verte d'une matrice
vector<float> getGreen(LightMatrix* m);

//Normalisation des matrices D et E par le maximum des coefficients de ces matrices
void normalisationV1(LightMatrix* D, LightMatrix* E);

//Renvoie l'indice du premier élément non nul de vec
int indexFirstPos(vector<float>* vec);

//Calcule le coefficient d'un élément de la matrice C correspondant à l'interaction entre deux faces
void fillFaceToFaceCell(Face* face1, Face* face2, LightMatrix* C, Scene* scene);

//Calcule le coefficient d'un élément de la matrice C correspondant à l'interaction entre une face et une source lumineuse
void fillLightToFaceCell(LightSource* light, Face* face, LightMatrix* C, Scene* scene);

//Gère le multithreading du calcul des coefficients de la matrice C
void* process_calculCoeffCMatrix(void* ptr_arg);

//Renvoie la matrice C associée à la scène
LightMatrix createCmatrixFromScene_multithreading(Scene&scene);

//Renvoie le vecteur E associée à la scène
LightMatrix createVectorFromScene(Scene&scene);

//Applique aux faces de la scène la couleur déterminée par D
void lightFaces(Scene&scene, LightMatrix* D);

// Calcul le rendu de lumière pour une scène, et effectue le nombre d'itérations lors de la résolution.
LightMatrix light(Scene&scene, float iteration);

//Mêmes calculs, avec effet de contraste.
LightMatrix lightV2(Scene&scene, float iteration);

// Calcul le rendu de lumière pour une scène, et effectue le nombre d'itérations lors de la résolution.
LightMatrix fake_light(Scene&scene, float iteration);

//Permet de passer au rendu de lumière selon light sans recalcul de C
void switch_to_lightV1(LightMatrix C, Scene&scene, float iteration);

//Permet de passer au rendu de lumière selon lightV2 sans recalcul de C
void switch_to_lightV2(LightMatrix C, Scene&scene, float iteration, float low, float up);

```


Lighting.cpp

```
#include "world.h"
#include "BSP.h"
#include <math.h>
#include <iostream>
#include <math.h>
#include <stack>
#include <pthread.h>
#include <stdexcept>
#include <vector>
#include <iostream>
#include <fstream>

#include "lighting.h"
#define NB_MULTITHREAD 12

using namespace std;

typedef Eigen::Matrix<bool, Eigen::Dynamic, Eigen::Dynamic> MatrixXb;

typedef struct arg_s arg_mt;

pthread_mutex_t verrou;

LightMatrix::LightMatrix(MatrixXf matrix) // Lors de la construction, matrix donne les dimensions et valeurs par default des matrices
rouge, vert et bleu.
{
    red = matrix;
    green = matrix;
    blue = matrix;
}

LightMatrix operator * ( const LightMatrix& x, const LightMatrix& y)
{
    LightMatrix m(x.red);
    m.red = x.red * y.red;
    m.blue = x.blue * y.blue;
    m.green = x.green * y.green;
    return m;
}
```

```
LightMatrix operator+ ( const LightMatrix& x, const LightMatrix& y)
{
    LightMatrix m(x.red);
    m.red = x.red + y.red;
    m.blue = x.blue + y.blue;
    m.green = x.green + y.green;
    return m;
}
```

```
LightMatrix operator- ( const LightMatrix& x, const LightMatrix& y)
{
    LightMatrix m(x.red);
    m.red = x.red - y.red;
    m.blue = x.blue - y.blue;
    m.green = x.green - y.green;
    return m;
}
```

```
LightMatrix operator/ ( const LightMatrix& x, const float y)
{
    LightMatrix m(x.red);
    m.red = x.red/y;
    m.green = x.green/y;
    m.blue = x.blue/y;
    return x;
}
```

```
float maxLightMat(LightMatrix * m)
{
    float maxR = abs(m->red( 0, 0));
    float maxG = abs(m->green( 0, 0));
    float maxB = abs(m->blue( 0, 0));
    for (int i = 0; i < m->red.rows(); i++)
    {
        maxR = max(maxR, abs(m->red(i, 0)));
        maxG = max(maxG, abs(m->green(i, 0)));
        maxB = max(maxB, abs(m->blue(i, 0)));
    }
    return max({maxR, maxG, maxB});
}
```

```
vector<float> getBlue(LightMatrix * m)
{
    vector<float> color(m->blue.rows(), 0);
    for (int i = 0; i < m->blue.rows(); i++)
    {
        color[i] = (m->blue(i, 0));
    }
    return color;
}
```

```
vector<float> getRed(LightMatrix * m)
{
    vector<float> color(m->red.rows(), 0);
    for (int i = 0; i < m->red.rows(); i++)
    {
        color[i] = (m->red(i, 0));
    }
    return color;
}
```

```
vector<float> getGreen(LightMatrix * m)
{
    vector<float> color(m->green.rows(), 0);
    for (int i = 0; i < m->green.rows(); i++)
    {
        color[i] = (m->green(i, 0));
    }
    return color;
}
```

```
void normalisationVl(LightMatrix * D, LightMatrix * E)
{
    float maxi = -INFINITY;
    for (int i = 0; i < D->red.rows(); i++)
    {
        maxi = max({D->red(i, 0), D->green(i, 0), D->blue(i, 0), maxi});
    }
    *D = *D/maxi;
    *E = *E/maxi;
}
```

```

int indexFirstPos(vector< float>* vec)
{
    for (int i = 0; i < vec->size(); i++)
    {
        if ((*vec)[i] != 0)
        {
            return i;
        }
    }
    return -1;
}

void fillFaceToFaceCell(Face * facel, Face * face2, LightMatrix * C, Scene * scene)
{
    int i = facel->id;
    int j = face2->id;
    Vector3f distanceVector = distanceVect(facel->center(), face2->center());
    float constCoef = max(0.f, cosAngle(face1->normalVector(), distanceVector))*max(0.f, cosAngle(face2->normalVector(),
-1*distanceVector))/(3.141592*max(0.001F, normeSquare(distanceVector)));
    if(constCoef > 0.000001)
    {
        if(!rayIsIntersectedInScene(facel->center(), distanceVector, scene))
        {
            if(!face2->material.emissive)
            {
                C->red(j,i) = ( static_cast<float>(face2->material.color.r) / 255.f)*facel->area()*constCoef;
                C->green(j,i) = ( static_cast<float>(face2->material.color.g) / 255.f)*facel->area()*constCoef;
                C->blue(j,i) = ( static_cast<float>(face2->material.color.b) / 255.f)*facel->area()*constCoef;
            }
            if(!facel->material.emissive)
            {
                C->red(i,j) = ( static_cast<float>(facel->material.color.r) / 255.f)*face2->area()*constCoef;
                C->green(i,j) = ( static_cast<float>(facel->material.color.g) / 255.f)*face2->area()*constCoef;
                C->blue(i,j) = ( static_cast<float>(facel->material.color.b) / 255.f)*face2->area()*constCoef;
            }
        }
    }
}

```

```
void fillLightToFaceCell(LightSource * light, Face * face, LightMatrix * C, Scene * scene)
{
    Vector3f distanceVector = distanceVect(light->origin.pos, face->center());
    float constCoef = max(0.f, cosAngle(face->normalVector(), -distanceVector))*light->strength/( 3.141592*max(0.001f,
normeSquare(distanceVector)));
    if(constCoef > 0.)
    {
        if(!rayIsIntersectedInScene(light->origin.pos, distanceVector, scene))
        {
            C->red(face->id, light->global_id) = ( static_cast<float>(face->material.color.r)/ 255.f)*constCoef;
            C->green(face->id, light->global_id) = ( static_cast<float>(face->material.color.g)/ 255.f)*constCoef;
            C->blue(face->id, light->global_id) = ( static_cast<float>(face->material.color.b)/ 255.f)*constCoef;
        }
    }
}
```

```

void* process_calculCoeffCMatrix( void* ptr_arg)
{

    arg_mt* a = (arg_mt*) ptr_arg;
    MatrixXb& hasToCalcul = *(a->hasToCalcul);
    LightMatrix* C = a->C;
    vector<Face*>& faceList = *(a->faceList);
    BinaryTree* binaryTree = a->binaryTree;
    Scene* scene = a->scene;
    int id_actuel;
    pthread_mutex_lock(&verrou);
    id_actuel = *(a->id_todo);
    *(a->id_todo) = *(a->id_todo)+ 1;
    pthread_mutex_unlock(&verrou);
    while (id_actuel < faceList.size())
    {
        Face* face = faceList[id_actuel];
        int i = face->id;
        for(Face* neighbor : faceList)
        {
            int j = neighbor->id;
            if (hasToCalcul(i,j))
            {
                hasToCalcul(i,j) = false;
                hasToCalcul(j,i) = false;
                fillFaceToFaceCell(face, neighbor, C, scene);
            }
        }
        if (id_actuel%100 == 0)
        {
            cout << id_actuel << "/" << faceList.size() << "(thread n°" << a->thread_id << ")" << endl;
        }
        pthread_mutex_lock(&verrou);
        id_actuel = *(a->id_todo);
        *(a->id_todo) = *(a->id_todo)+ 1;
        pthread_mutex_unlock(&verrou);
    }

    return nullptr;
}

```

```

LightMatrix createCmatrixFromScene_multithreading(Scene &scene)
{
    BinaryTree* binaryTree = buildBinaryTree(scene); // Arbre binaire de la scène
    vector<Face*> faceList = scene.getFacesPointer(); // Liste des pointeurs de chaque face dans la scène.
    vector<LightSource*> lightList = scene.getLightSourcesPointer(); // Liste des pointeurs de chaque source lumineuse dans la scène.
    LightMatrix C(MatrixXf::Constant(faceList.size() + lightList.size(), faceList.size() + lightList.size(), 0.f)); // Matrice C
    MatrixXb hasToCalcul = MatrixXb::Constant(faceList.size(), faceList.size(), true); // Matrice donnant les coefficients à calculer.
    arg_mt* args = (arg_mt*) malloc(NB_MULTITHREAD* sizeof(arg_mt));
    int* todo = (int*) malloc(sizeof(int));
    *todo = 0;
    for (int k = 0; k < NB_MULTITHREAD; k++)
    {
        args[k].id_todo = *todo;
        args[k].thread_id = k;
        args[k].binaryTree = binaryTree;
        args[k].faceList = &faceList;
        args[k].C = &C;
        args[k].hasToCalcul = &hasToCalcul;
        args[k].scene = &scene;
    }
    pthread_t* threads = (pthread_t*) malloc(NB_MULTITHREAD* sizeof(pthread_t));
    for (int k = 0; k < NB_MULTITHREAD; k++)
    {
        pthread_create(threads+k, NULL, process_calculCoeffCMatrix, args+k);
    }
    for (int k = 0; k < NB_MULTITHREAD; k++)
    {
        pthread_join(threads[k], NULL);
    }
    for (LightSource* light : lightList)
    {
        for (Face* face : faceList)
        {
            fillLightToFaceCell(light, face, &C, &scene);
        }
    }

    return C;
}

```

```

LightMatrix createVectorFromScene(Scene &scene)
{
    vector<Face> faceList = scene.getFaces();
    vector<LightSource> lightList = scene.getLightSources();
    LightMatrix E(MatrixXf::Constant(faceList.size() + lightList.size(), 1,0.f));
    for (size_t i = 0; i < faceList.size(); i++)
    {
        if(faceList[i].material.emissive)
        {
            E.red(i, 0) = (static_cast<float>(faceList[i].material.color.r)/ 255.f);
            E.green(i, 0) = (static_cast<float>(faceList[i].material.color.g)/ 255.f);
            E.blue(i, 0) = (static_cast<float>(faceList[i].material.color.b)/ 255.f);
        }
    }

    for(int i = 0; i < lightList.size(); i++)
    {
        int coef = faceList.size() + i;
        E.red(coef, 0) = (static_cast<float>(lightList[i].color.r) / 255.f);
        E.green(coef, 0) = (static_cast<float>(lightList[i].color.g) / 255.f);
        E.blue(coef, 0) = (static_cast<float>(lightList[i].color.b) / 255.f);
    }
    return E;
}

void lightFaces(Scene &scene, LightMatrix & D)
{
    int k = 0;
    for (Object& object : scene.getObjects())
    {
        for (Face& face : object.faces)
        {
            face.lightAmount = sf::Color(
                static_cast<std::uint8_t>((D.red(k, 0))*255),
                static_cast<std::uint8_t>((D.green(k, 0))*255),
                static_cast<std::uint8_t>((D.blue(k, 0))*255));
            k++;
        }
    }
}

```



```

LightMatrix light(Scene &scene, float iteration = 1)
{
    LightMatrix D = createVectorFromScene(scene);
    LightMatrix E = createVectorFromScene(scene);
    LightMatrix C = createCmatrixFromScene_multithreading(scene);

    // Iteration.
    for (int i = 0; i < iteration; i++)
    {
        D = (C*D) + E;
        // Normalisation (valeurs entre 0 et 1)
        normalisationV1(&D, &E);
    }

    float maxi = -INFINITY;
    vector<Face> faceList = scene.getFaces();
    for (int i = 0; i < faceList.size(); i++)
    {
        maxi = max({D.red(i,0), D.blue(i,0), D.green(i,0), maxi});
    }
    for (int i = 0; i < faceList.size(); i++)
    {
        D.red(i,0) = min(1.f, D.red(i,0)/maxi);
        D.green(i,0) = min(1.f, D.green(i,0)/maxi);
        D.blue(i,0) = min(1.f, D.blue(i,0)/maxi);
    }
    for (int i = 0; i < D.red.rows(); i++)
    {
        D.red(i, 0) = pow(D.red(i, 0), 0.82);
        D.green(i, 0) = pow(D.green(i, 0), 0.82);
        D.blue(i, 0) = pow(D.blue(i, 0), 0.82);
    }
    ofstream myfile;
    myfile.open("values1.txt");
    for(int i = 0; i < D.red.rows(); i++)
    {
        myfile << D.red(i, 0) << "\n";
    }
    myfile << "\n";
    for(int i = 0; i < D.red.rows(); i++)
    {
        myfile << D.blue(i, 0) << "\n";
    }
    myfile << "\n";
    for(int i = 0; i < D.red.rows(); i++)
    {
        myfile << D.green(i, 0) << "\n";
    }
    myfile.close();
    lightFaces(scene, D);
    std::cout << "Eclairage de la scene termin .\n";
    scene.previewMode = false;
    return C;
}

```

```

LightMatrix lightV2(Scene &scene, float iteration = 1){
    LightMatrix D = createVectorFromScene(scene);
    LightMatrix E = createVectorFromScene(scene);
    LightMatrix C = createCmatrixFromScene_multithreading(scene);
    vector<float> vectBlue;
    vector<float> vectRed;
    vector<float> vectGreen;
    int fifth_perc = static_cast<int>((1./100.)*D.green.rows());
    int ninety_fifth_perc = static_cast<int>((40./100.)*D.green.rows());

    // Iteration.
    for (int i = 0; i < iteration; i++)
    {
        D = (C*D) + E;
        // Normalisation (valeurs entre 0 et 1)
        float maxi = 1.;
        for (int i = 0; i < D.red.rows(); i++)
        {
            maxi = max({D.red(i, 0), D.green(i, 0), D.blue(i, 0), maxi});
        }
        vectBlue = getBlue(&D);
        vectRed = getRed(&D);
        vectGreen = getGreen(&D);
        sort(vectBlue.begin(), vectBlue.end());
        sort(vectRed.begin(), vectRed.end());
        sort(vectGreen.begin(), vectGreen.end());
        float maxc = max({vectRed[ninety_fifth_perc], vectBlue[ninety_fifth_perc], vectGreen[ninety_fifth_perc]});
        float minc = min({vectRed[fifth_perc], vectBlue[fifth_perc], vectGreen[fifth_perc]});

        for (int i = 0; i < D.red.rows(); i++)
        {
            if ((D.red(i, 0)) > minc && D.red(i, 0) < maxc)
            {
                D.red(i, 0) = (D.red(i, 0)-minc)/(maxc - minc);
            }
            if ((D.blue(i, 0)) > minc && D.blue(i, 0) < maxc)
            {
                D.blue(i, 0) = (D.blue(i, 0)-minc)/(maxc - minc);
            }
            if ((D.green(i, 0)) > minc && D.green(i, 0) < maxc)

```

```

    {
        D.green(i,0) = (D.green(i,0)-minc)/(maxc - minc);
    }
}

float maxi = -INFINITY;
vector<Face> faceList = scene.getFaces();
for (int i = 0; i < D.red.rows(); i++)
{
    if(!faceList[i].material.emissive) maxi = max(D.red(i,0), max(D.blue(i,0), max(D.green(i,0), maxi)));
}

for (int i = 0; i < D.red.rows(); i++)
{
    D.red(i,0) = min(1.f, D.red(i,0)/maxi);
    D.green(i,0) = min(1.f, D.green(i,0)/maxi);
    D.blue(i,0) = min(1.f, D.blue(i,0)/maxi);
}
for (int i = 0; i < D.red.rows(); i++)
{
    D.red(i, 0) = pow(D.red(i, 0), 0.4545);
    D.green(i, 0) = pow(D.green(i, 0), 0.4545);
    D.blue(i, 0) = pow(D.blue(i, 0), 0.4545);
}

ofstream myfile;
myfile.open("values2.txt");
for(int i = 0; i < D.red.rows(); i++)
{
    myfile << D.red(i,0) << "\n";
}
myfile << "\n";
for(int i = 0; i < D.red.rows(); i++)
{
    myfile << D.blue(i,0) << "\n";
}
myfile << "\n";
for(int i = 0; i < D.red.rows(); i++)
{
    myfile << D.green(i,0) << "\n";
}
myfile.close();
lightFaces(scene, D);
std::cout << "Eclairage de la scene terminé.\n";
scene.previewMode = false;
return C;}

```

```

void switch_to_lightV1(LightMatrix C, Scene&scene, float iteration = 1)
{
    LightMatrix D = createVectorFromScene(scene);
    LightMatrix E = createVectorFromScene(scene);
    // Iteration.
    for (int i = 0; i < iteration; i++)
    {
        D = (C*D) + E;
        // Normalisation (valeurs entre 0 et 1)
        normalisationV1(&D, &E);
    }

    float maxi = -INFINITY;
    vector<Face> faceList = scene.getFaces();
    for (int i = 0; i < faceList.size(); i++)
    {
        maxi = max({D.red(i,0), D.blue(i,0), D.green(i,0), maxi});
    }
    for (int i = 0; i < faceList.size(); i++)
    {
        D.red(i,0) = min(1.f, D.red(i,0)/maxi);
        D.green(i,0) = min(1.f, D.green(i,0)/maxi);
        D.blue(i,0) = min(1.f, D.blue(i,0)/maxi);
    }
    ofstream myfile;
    myfile.open("values1.txt");
    for(int i = 0; i < D.red.rows(); i++)
    {
        myfile << D.red(i,0) << "\n";
    }
    myfile << "\n";
    for(int i = 0; i < D.red.rows(); i++)
    {
        myfile << D.blue(i,0) << "\n";
    }
    myfile << "\n";
    for(int i = 0; i < D.red.rows(); i++)
    {
        myfile << D.green(i,0) << "\n";
    }
    myfile.close();
    lightFaces(scene, D);
    std::cout << "Eclairage de la scene terminé.\n";
    scene.previewMode = false;
}

```

```

LightMatrix fake_light(Scene &scene, float iteration = 1)
{
    LightMatrix D = createVectorFromScene(scene);
    LightMatrix E = createVectorFromScene(scene);
    LightMatrix C(MatrixXf::Constant(scene.getFaces().size() + scene.getLightSources().size(), scene.getFaces().size() +
scene.getLightSources().size(), 0.f)); // Matrice C
    return C;
}

void switch_to_lightV2(LightMatrix C, Scene &scene, float iteration = 1, float low = 0.05, float up = 0.9)
{
    LightMatrix D = createVectorFromScene(scene);
    LightMatrix E = createVectorFromScene(scene);
    vector<float> vectBlue;
    vector<float> vectRed;
    vector<float> vectGreen;
    vector<float> vectGrey(D.green.rows(), 0);
    int fifth_perc = static_cast<int>((low)*D.green.rows());
    int ninety_fifth_perc = static_cast<int>((up)*D.green.rows());

    // Iteration.
    float max_diff = 1;
    LightMatrix newD(MatrixXf::Constant(D.red.rows(), 1, 0.f));
    for (int i = 0; i < iteration; i++)
    {
        newD = (C*D) + E;
        LightMatrix diff = D-newD;
        if (maxLightMat(&diff) < 0.05)
        {
            cout << i << endl;
            D = newD;
            break;
        }
        D = newD;
        // Normalisation (valeurs entre 0 et 1)
        float maxi = 1.;
        for (int i = 0; i < D.red.rows(); i++)
        {
            maxi = max({D.red(i, 0), D.green(i, 0), D.blue(i, 0), maxi});
        }
    }
}

```

```

vectBlue = getBlue(&D);
vectRed = getRed(&D);
vectGreen = getGreen(&D);
for (int i = 0; i < vectGreen.size(); i++)
{
    vectGrey[i] = (vectBlue[i] + vectRed[i] + vectGreen[i])/ 3;
}
sort(vectBlue.begin(), vectBlue.end());
sort(vectRed.begin(), vectRed.end());
sort(vectGreen.begin(), vectGreen.end());
sort(vectGrey.begin(), vectGrey.end());

float minc = vectGrey[fifth_perc];
float maxc = vectGrey[ninety_fifth_perc];
float alpha = 1 / (maxc - minc);
float beta = -minc * alpha;
for (int i = 0; i < D.red.rows(); i++)
{
    {
        D.red(i, 0) = D.red(i, 0) * alpha + beta;
        D.green(i, 0) = D.green(i, 0) * alpha + beta;
        D.blue(i, 0) = D.blue(i, 0) * alpha + beta;
    }
}

float maxi = -INFINITY;
vector<Face> faceList = scene.getFaces();
for (int i = 0; i < D.red.rows(); i++)
{
    if(!faceList[i].material.emissive) maxi = max(D.red(i, 0), max(D.blue(i, 0), max(D.green(i, 0), maxi)));
}
maxi = min(1.f, maxi);
for (int i = 0; i < D.red.rows(); i++)
{
    D.red(i, 0) = min(1.f, D.red(i, 0)/maxi);
    D.green(i, 0) = min(1.f, D.green(i, 0)/maxi);
    D.blue(i, 0) = min(1.f, D.blue(i, 0)/maxi);
}

```

```

for (int i = 0; i < D.red.rows(); i++)
{
    D.red(i, 0) = pow(D.red(i, 0), 0.82);
    D.green(i, 0) = pow(D.green(i, 0), 0.82);
    D.blue(i, 0) = pow(D.blue(i, 0), 0.82);
}

ofstream myfile;
myfile.open("values2.txt");
for (int i = 0; i < D.red.rows(); i++)
{
    myfile << D.red(i, 0) << "\n";
}
myfile << "\n";
for (int i = 0; i < D.red.rows(); i++)
{
    myfile << D.blue(i, 0) << "\n";
}
myfile << "\n";
for (int i = 0; i < D.red.rows(); i++)
{
    myfile << D.green(i, 0) << "\n";
}
myfile.close();
lightFaces(scene, D);
std::cout << "Eclairage de la scene V2 terminé.\n" ;
scene.previewMode = false;
}

```

main.cpp

```
#include "view.h"
#include "world.h"
#include "lighting.h"
#include "BSP.h"
#include <math.h>
#include <iostream>
#include <assert.h>

using namespace std;
using namespace sf;

sf::Vector2i LastMousePosition = sf::Mouse::getPosition();

bool ROTATING = false;

int main(){
    Scene scene3d;

    Material wood(sf::Color( 70,60,30), false);
    Material concrete(sf::Color( 250,250,250));
    Material foam (sf::Color( 200,170,150));

    // /* Objets */
    Object floor( "./models/diapo/abs_ombre/floor.obj" , Origin(), concrete);
    scene3d.addObject(floor);
    Object ceiling( "./models/diapo/abs_ombre/ceiling.obj" , Origin(), sf::Color( 255,255,255));
    scene3d.addObject(ceiling);
    Object cube( "./models/diapo/abs_ombre/cube.obj" , Origin(), concrete);
    scene3d.addObject(cube);
    Object wall_b( "./models/diapo/abs_ombre/wall_back.obj" , Origin(), sf::Color( 255,255,255));
    scene3d.addObject(wall_b);
    Object wall_l( "./models/diapo/abs_ombre/wall_left.obj" , Origin(), sf::Color( 60,170,180));
    scene3d.addObject(wall_l);
    Object wall_r( "./models/diapo/abs_ombre/wall_right.obj" , Origin(), sf::Color( 180,80,80));
    scene3d.addObject(wall_r);

    LightSource light_src ( 0, Origin(), Color( 255,250,250), 10000.);
    light_src.origin.pos = Eigen::Vector3f { 0,9,0};
    scene3d.addLightSource(light_src);
```



```

scene3d.subdivideFaces( 01);

LightMatrix C = fake_light(scene3d, 100);

ContextSettings settings;
settings.antialiasingLevel = 8;
sf::RenderWindow window = sf::RenderWindow(sf::VideoMode(SCREEN_WIDTH, SCREEN_HEIGHT), "Rendu 3D - " +
to_string(scene3d.getFaces().size()) + " faces." , sf::Style::Close, settings);
float low = 0.02;
float up = 0.98;
float step = 0.002;
int id = 0;
while (window.isOpen())
{
    sf::Event event;
    while (window.pollEvent(event))
    {
        switch (event.type)
        {
            case sf::Event::Closed:
                window.close();
                break;

            case sf::Event::MouseWheelScrolled:
                if(sf::Keyboard::isKeyPressed(sf::Keyboard::LShift))
                {
                    scene3d.camera.elevate(event.mouseWheelScroll.delta);
                }
                else
                {
                    scene3d.camera.fov = max( 0.1, scene3d.camera.fov + event.mouseWheelScroll.delta* 0.1);
                    scene3d.camera.fov = min( 3.14f, scene3d.camera.fov);
                }
                break;
        }
    }
}

```

```
case sf::Event::KeyPressed:
    switch (event.key.code)
    {
        case sf::Keyboard::P:
            scene3d.previewMode = !scene3d.previewMode;
            break;

        case sf::Keyboard::S: // Activation/Désactivation de l'ombrage doux
            scene3d.smoothShading = !scene3d.smoothShading;
            break;

        case sf::Keyboard::X:
            switch_to_lightV1(C, scene3d, 100);
            break;

        case sf::Keyboard::Y:
            switch_to_lightV2(C, scene3d, 100, low, up);
            break;

        case sf::Keyboard::Right:
            low += step;
            break;

        case sf::Keyboard::Left:
            low -= step;
            break;

        case sf::Keyboard::Up:
            up += step;
            break;

        case sf::Keyboard::Down:
            up -= step;
            break;

        case sf::Keyboard::Enter:
            switch_to_lightV2(C, scene3d, 100, low, up);
            break;
        default:
```

```

        break;

        case sf::Keyboard::T:
            scene3d.test(id);
            id++;
            break;
    }

    default:
        break;
}
}
if (sf::Mouse::isButtonPressed(sf::Mouse::Left)) // Contrôle de la caméra
{
    if (!ROTATING) LastMousePosition = sf::Mouse::getPosition();
    ROTATING = true;
    sf::Vector2i relMousePos = sf::Mouse::getPosition();
    relMousePos = relMousePos - LastMousePosition;
    LastMousePosition = sf::Mouse::getPosition();
    scene3d.camera.rotate(relMousePos.x, relMousePos.y);
} else ROTATING = false;
renderScene(scene3d, window);
ostringstream myString;
myString << "Range: " << low << " - " << up;
window.setTitle(myString.str());
}
return 0;
}

```

view.h

```
#pragma once
#include "world.h"

typedef Eigen::Matrix<sf::Color, Eigen::Dynamic, Eigen::Dynamic> MatrixXcolor;

// Bijection linéaire de [-1, 1] dans [0, SCREEN_WIDTH]
float Xscreen(float val);

// Bijection linéaire de [-1, 1] dans [0, SCREEN_HEIGHT]
float Yscreen(float val);

float sign (Eigen::Vector2f& p1, Eigen::Vector2f& p2, Eigen::Vector2f& p3);

// Retourne la couleur du vertex %n_vertex (dans[0,2])
sf::Color getColorOfVertex(Face& face, int n_vertex);

// Retourne si la point de coordonnées (%x,%y) sur l'écran est dans le triangle dessiné par la %face selon sa %projection
bool pointIsInTriangle(Face& face, int x, int y, MatrixXf& projection);

// Retourne la coordonnée z du point (%x,%y) se situant sur la %face, et selon la %projection
float getZofPoint(int x, int y, Face& face, MatrixXf& projectedVertices);

// Retourne la coordonnée z du point (%x,%y) se situant sur la %face, et selon la %projection
sf::Color getColorOfPoint(int x, int y, Face& face, MatrixXf& projectedVertices);

/* Réalise une projection matricielle de la %matrix selon la %camera */
MatrixXf matrixProjectionFromCamera(MatrixXf matrix, Camera& camera);

// À compléter
void projectObject(Object &object, Scene &scene, MatrixXf& zBuffer, MatrixXcolor& colorMap);

// À compléter
void drawColorMap(MatrixXcolor colorMap, sf::RenderWindow &window);

// À compléter
void drawLightSources(Scene& scene, sf::RenderWindow &window, MatrixXf& zBuffer);

// À compléter
void drawBoundingBoxes(Scene& scene, sf::RenderWindow &window, MatrixXf& zBuffer);

// Rend la scene sur la fenetre window.
void renderScene(Scene &scene, sf::RenderWindow &window);
```

view.cpp

```
#include <vector>
#include <iostream>

#include "view.h"

using Eigen::MatrixXf;
using Eigen::Matrix3Xf;
using Eigen::Vector2f;
using Eigen::VectorXf;

using namespace Eigen;
using namespace std;
using namespace sf;

// Bijection linéaire de [-1, 1] dans [0, SCREEN_WIDTH]
float Xscreen(float val){ return (1+val)*SCREEN_WIDTH/2; }

// Bijection linéaire de [-1, 1] dans [0, SCREEN_HEIGHT]
float Yscreen(float val){ return (1-val)*SCREEN_HEIGHT/2; }

float sign (Eigen::Vector2f& p1, Eigen::Vector2f& p2, Eigen::Vector2f& p3) {
    return (p1(0) - p3(0)) * (p2(1) - p3(1)) - (p2(0) - p3(0)) * (p1(1) - p3(1));
}

// Retourner la couleur du vertex %n_vertex (dans[0,2])
sf::Color getColorOfVertex(Face& face, int n_vertex){
    float red = 0;
    float blue = 0;
    float green = 0;
    float tot = 0.;
    vector<int> facesIndex = face.parentObject().facesAttachedToVertex[face.vertexIndex(n_vertex)];
    for (int i = 0; i < facesIndex.size(); i++)
    {
        Face otherFace = face.parentObject().faces[facesIndex[i]];
        float coef_mult = max(0.F, cosAngle(face.normalVector(), otherFace.normalVector()));
        red += otherFace.lightAmount.r * coef_mult;
        blue += otherFace.lightAmount.b * coef_mult;
        green += otherFace.lightAmount.g * coef_mult;
        tot += coef_mult;
    }
    return sf::Color(static_cast<uint8_t>(red/tot),static_cast<uint8_t>(green/tot),static_cast<uint8_t>(blue/tot));
}
```

```

// Retourne si la point de coordonnée (%x,%y) sur l'écran est dans le triangle dessiné par la %face selon sa %projection
bool pointIsInTriangle(Face & face, int x, int y, MatrixXf & projection){
    Eigen::Vector2f pt { static_cast<float>(x), static_cast<float>(y)};
    Eigen::Vector2f v1 {projection( 0, face.vertexIndex( 0)), projection( 1, face.vertexIndex( 0))};
    Eigen::Vector2f v2 {projection( 0, face.vertexIndex( 1)), projection( 1, face.vertexIndex( 1))};
    Eigen::Vector2f v3 {projection( 0, face.vertexIndex( 2)), projection( 1, face.vertexIndex( 2))};

    float d1, d2, d3;
    bool has_neg, has_pos;

    d1 = sign(pt, v1, v2);
    d2 = sign(pt, v2, v3);
    d3 = sign(pt, v3, v1);

    has_neg = (d1 < 0) || (d2 < 0) || (d3 < 0);
    has_pos = (d1 > 0) || (d2 > 0) || (d3 > 0);

    return !(has_neg && has_pos);
}

// Retourne la coordonnée z du point (%x,%y) se situant sur la %face, et selon la %projection
float getZofPoint(int x, int y, Face& face, MatrixXf & projectedVertices)
{
    Eigen::Vector2f point {x, y};
    Eigen::Vector2f pa {projectedVertices( 0, face.vertexIndex( 0)) - x, projectedVertices( 1, face.vertexIndex( 0)) - y};
    Eigen::Vector2f pb {projectedVertices( 0, face.vertexIndex( 1)) - x, projectedVertices( 1, face.vertexIndex( 1)) - y};
    Eigen::Vector2f pc {projectedVertices( 0, face.vertexIndex( 2)) - x, projectedVertices( 1, face.vertexIndex( 2)) - y};
    float z1 = projectedVertices( 2, face.vertexIndex( 0));
    float z2 = projectedVertices( 2, face.vertexIndex( 1));
    float z3 = projectedVertices( 2, face.vertexIndex( 2));
    float alpha = pb(0)*pc(1) - pb(1)*pc(0); // Coefficient sommet 0
    float beta = pc(0)*pa(1) - pc(1)*pa(0); // Coefficient sommet 1
    float gamma = pa(0)*pb(1) - pa(1)*pb(0); // Coefficient sommet 2
    return (alpha*z1 + beta*z2 + gamma*z3)/(alpha+beta+gamma);
}

```

```

// Retourne la coordonnée z du point (%x,%y) se situant sur la %face, et selon la %projection
sf::Color getColorOfPoint(int x, int y, Face& face, MatrixXf& projectedVertices)
{
    Eigen::Vector2f point {x, y};
    Eigen::Vector2f pa {projectedVertices(0, face.vertexIndex(0)) - x, projectedVertices(1, face.vertexIndex(0)) - y};
    Eigen::Vector2f pb {projectedVertices(0, face.vertexIndex(1)) - x, projectedVertices(1, face.vertexIndex(1)) - y};
    Eigen::Vector2f pc {projectedVertices(0, face.vertexIndex(2)) - x, projectedVertices(1, face.vertexIndex(2)) - y};
    sf::Color color1 = getColorOfVertex(face, 0);
    sf::Color color2 = getColorOfVertex(face, 1);
    sf::Color color3 = getColorOfVertex(face, 2);
    sf::Color finalColor(0,0,0);
    float alpha = pb(0)*pc(1) - pb(1)*pc(0); // Coefficient sommet 0
    float beta = pc(0)*pa(1) - pc(1)*pa(0); // Coefficient sommet 1
    float gamma = pa(0)*pb(1) - pa(1)*pb(0); // Coefficient sommet 2
    finalColor.r = static_cast<uint8_t>((alpha*static_cast<float>(color1.r) + beta*static_cast<float>(color2.r) + gamma*static_cast<float>(color3.r))/(alpha+beta+gamma));
    finalColor.g = static_cast<uint8_t>((alpha*static_cast<float>(color1.g) + beta*static_cast<float>(color2.g) + gamma*static_cast<float>(color3.g))/(alpha+beta+gamma));
    finalColor.b = static_cast<uint8_t>((alpha*static_cast<float>(color1.b) + beta*static_cast<float>(color2.b) + gamma*static_cast<float>(color3.b))/(alpha+beta+gamma));

    return finalColor;
}

/* Réalise une projection matricielle de la %matrix selon la %camera */
MatrixXf matrixProjectionFromCamera(MatrixXf matrix, Camera& camera)
{
    float tanFov = tanf(camera.fov/2);
    Matrix4f projectionMatrix {
        {(SCREEN_HEIGHT_FLOAT/SCREEN_WIDTH_FLOAT)/tanFov, 0.f, 0.f, 0.f},
        {0.f, 1.f/tanFov, 0.f, 0.f},
        {0.f, 0.f, Z_FAR/(Z_FAR-Z_NEAR), -Z_FAR*Z_NEAR/(Z_FAR-Z_NEAR)},
        {0.f, 0.f, 1.f, 0.f}
    };

    matrix.conservativeResize(4, NoChange); // On ajoute la coordonnée homogène
    for (int i = 0; i < matrix.cols(); i++)
    {
        matrix(3,i) = 1;
    }

    matrix = projectionMatrix * matrix; // On projette

    for (int i = 0; i < matrix.cols(); i++) // On normalise chaque projection de vecteur
    {
        matrix.col(i) = matrix.col(i)/matrix(3,i);
        matrix(0,i) = Xscreen(matrix(0,i));
        matrix(1,i) = Yscreen(matrix(1,i));
    }
    matrix.conservativeResize(3, NoChange); // On redonne la bonne taille à la matrice
    return matrix;
}

```

```

void projectObject(Object &object, Scene &scene, MatrixXf & zBuffer, MatrixXcolor & colorMap){

    MatrixXf verticesOnScreen = matrixProjectionFromCamera(coordonateForCamera(object.getVerticesMatrix(), scene.camera), scene.camera);

    for (Face &face : object.faces)
    {
        int min_y = max(0, static_cast<int>(min({verticesOnScreen( 1, face.vertexIndex( 0)),
            verticesOnScreen( 1, face.vertexIndex( 1)),
            verticesOnScreen( 1, face.vertexIndex( 2))})))-1);
        int max_y = min(scene.camera.height, static_cast<int>(max({verticesOnScreen( 1, face.vertexIndex( 0)),
            verticesOnScreen( 1, face.vertexIndex( 1)),
            verticesOnScreen( 1, face.vertexIndex( 2))})))+1);
        int min_x = max(0, static_cast<int>(min({verticesOnScreen( 0, face.vertexIndex( 0)),
            verticesOnScreen( 0, face.vertexIndex( 1)),
            verticesOnScreen( 0, face.vertexIndex( 2))})))-1);
        int max_x = min(scene.camera.width, static_cast<int>(max({verticesOnScreen( 0, face.vertexIndex( 0)),
            verticesOnScreen( 0, face.vertexIndex( 1)),
            verticesOnScreen( 0, face.vertexIndex( 2))})))+1);

        for (int y = min_y; y < max_y; y++)
        {
            for (int x = min_x; x < max_x; x++)
            {
                if(pointIsInTriangle(face, x, y, verticesOnScreen))
                {
                    float zPixel = getZofPoint(x, y, face, verticesOnScreen); /* coordonnée z du pixel sur le triangle */
                    if(zBuffer(y,x) < zPixel)
                    {
                        zBuffer(y,x) = zPixel;
                        colorMap(y,x) = scene.previewMode ?(face.highlight ? White : face.previewColor) : (scene.smoothShading ?
getColorOfPoint(x, y, face, verticesOnScreen) : face.lightAmount);
                    }
                }
            }
        }
    }
}

```



```

void drawColorMap(MatrixXcolor colorMap, sf::RenderWindow &window)
{
    sf::VertexArray pixel (sf::Points, colorMap.cols()*colorMap.rows());
    for (int i = 0; i < colorMap.rows(); i++)
    {
        for (int j = 0; j < colorMap.cols(); j++)
        {
            sf::Vertex vertex(sf::Vector2f(j, i), colorMap(i,j));
            pixel[i*colorMap.cols() + j] = vertex;
        }
    }
    window.draw(pixel);
}

void drawLightSources(Scene & scene, sf::RenderWindow &window, MatrixXf & zBuffer)
{
    for (LightSource& lightSourcePointer : scene.getLightSources())
    {
        MatrixXf projectedSource = matrixProjectionFromCamera(coordonateForCamera(lightSourcePointer.origin.pos, scene.camera),
scene.camera);

        if(projectedSource( 0,0) >=0 && projectedSource( 1,0) >=0 && projectedSource( 0,0) < scene.camera.width && projectedSource( 1,0) <
scene.camera.height)
        {
            if(projectedSource( 2,0) > zBuffer( static_cast<int>(projectedSource( 1,0)), static_cast<int>(projectedSource( 0,0))))
            {
                sf::CircleShape shape( 3);
                shape.setFillColor(lightSourcePointer.color);
                shape.setOutlineColor(sf::Color( 0,0,0));
                shape.setOutlineThickness( 1);
                shape.setPosition(projectedSource( 0,0)-3, projectedSource( 1,0)-3);
                window.draw(shape);
            }
        }
    }
}

```

```
void renderScene(Scene &scene, sf::RenderWindow &window)
{

    MatrixXf zBuffer = MatrixXf::Constant(window.getSize().y, window.getSize().x, -INFINITY);
    MatrixXcolor colorMap = MatrixXcolor::Constant(window.getSize().y, window.getSize().x, sf::Color(40,40,40,255));
    // 20 20 20

    for(Object& object : scene.getObjects())
    {
        projectObject(object, scene, zBuffer, colorMap);
    }
    window.clear();
    drawColorMap(colorMap, window);
    if(scene.previewMode)
    {
        drawBoundingBoxes(scene, window, zBuffer);
        drawLightSources(scene, window, zBuffer);
    }
    window.display();
    window.setTitle(to_string(rand()%00000));
}
```

world.h

```
#pragma once
#include <iostream>
#include <vector>
#include <Eigen/Dense>

#include <SFML/Window.hpp>
#include <SFML/Graphics.hpp>
#include <SFML/System.hpp>

using Eigen::MatrixXf;
using Eigen::Matrix3Xf;
using Eigen::Vector3f;

#define Point 0
#define Sun 1
#define White sf::Color( 255,255,255)

using namespace std;

extern const float FOV_ANGLE;
extern const int SCREEN_HEIGHT;
extern const int SCREEN_WIDTH;
extern const float SCREEN_HEIGHT_FLOAT;
extern const float SCREEN_WIDTH_FLOAT;
extern const float Z_FAR;
extern const float Z_NEAR;

class Material; class Origin; class Face; class BoundingBox; class Object; class Camera; class Scene;

#include "BSP.h"

/* Renvoie les coordonnées du vecteur vect après rotation autour de X. */
Matrix3Xf rotateX(Matrix3Xf vect, float angle);

/* Renvoie les coordonnées du vecteur vect après rotation autour de Y. */
Matrix3Xf rotateY(Matrix3Xf vect, float angle);

/* Renvoie les coordonnées du vecteur vect après rotation autour de Z. */
Matrix3Xf rotateZ(Matrix3Xf vect, float angle);
```

```

/* Renvoie les coordonnées du vecteur vect après rotations successives autour de X, Y et Z. */
Matrix3Xf rotate3d(Matrix3Xf vector, Vector3f rotVect);

/* Renvoie les coordonnées du vecteur vect après translation par le vecteur t. */
Matrix3Xf translate(Matrix3Xf vector, Vector3f t);

/* Calcule le produit scalaire de deux vecteurs.*/
float scalar(Vector3f v1, Vector3f v2);

/* Calcule le produit vectoriel de deux vecteurs.*/
Vector3f cross(Vector3f v1, Vector3f v2);

/*Renvoie la distance entre deux points de l'espace sous la forme d'un vecteur 3D*/
Vector3f distanceVect(Vector3f v1, Vector3f v2);

/*Renvoie la distance entre deux points de l'espace*/
float distance(Vector3f v1 = Vector3f::Zero(), Vector3f v2 = Vector3f::Zero());

/*Renvoie la norme du vecteur V*/
float norme(Vector3f v);

/*Renvoie la norme du vecteur V au carré*/
float normeSquare(Vector3f v);

/*Renvoie le cosinus de l'angle entre les deux vecteurs*/
float cosAngle(Vector3f v1, Vector3f v2);

/*Renvoie les nouvelles coordonnées du vecteur V dans la base du monde, en supposant que ses coordonnées soient données dans la base Origin */
Matrix3Xf coordonateInWorld(Matrix3Xf v, Origin origin);

/* Renvoie les nouvelles coordonnées d'un vecteur de coordonnées COORD dans la base de la CAMERA, en supposant que ses coordonnées soient données dans la base du monde */
Matrix3Xf coordonateForCamera(Matrix3Xf v, Camera cam);

// /* STRUCTURES */

```

```

class Material{
    public:
        bool emissive; //Lumière émise par le matériau (par défaut aucune).
        sf::Color color; //Couleur propre du matériau (par défaut blanche).
        Material(sf::Color col = White, bool emit = false);
};

class Face{
    private:
        bool area__calculated = false;
        bool center__calculated = false;
        bool normal__calculated = false;
        float area__;
        Vector3f center__;
        Vector3f normal__;
    public:
        int verticesIndex[3]; //Indices des éléments de vertices dans la matrice verticies de Object.
        int id; // Correspond à l'indice de la face dans la liste des faces de la scène ET sa ligne dans la matrice C
        bool highlight;
        Scene* scenePointer; // Pointeur vers la scene dont la face dépend (Dangereux)
        int objectIndex; // Index de l'objet parent dnas la liste d'objets de la scene.
        sf::Color lightAmount; //Couleur de la face après calcul de lumière.
        sf::Color previewColor;
        Material material; //Matériau associé à la face.
        Vector3f normalVector(); // Vecteur normal à la surface de la face.
        float area(); // Aire de la face
        Vector3f center(); // Centre de la face, en coordonnées globales.
        Object & parentObject(); // Objet dont dépend la face.
        Vector3f vertex(int i); //Renvoie le ième vertex de la face (0<=i<3), en coordonnées globales
        int vertexIndex(int i); //Renvoie le n° de colonne dans la matrice de l'objet correspondant au ième vertex de la face (0<=i<3)
        bool isIntersectedByRay(Vector3f rayOrigin, Vector3f rayDirection); // Renvoie si la face est traversée par la rayon
        Face(int v1, int v2, int v3, Material material);
        void modifyVertex(int vertexToModify, int newVertexIndex);
        void reload();
};

```

```

class Origin{
public:
    Vector3f pos; //Position du référentiel de l'objet dans celui de la scène ( (0, 0, 0, 1) initialement)
    Vector3f rot; //Angles de rotation du référentiel de l'objet dans celui de la scène ( (0, 0, 0, 1) initialement).
    Origin(Vector3f position=Vector3f::Zero(), Vector3f rotation=Vector3f::Zero());
};

class BoundingBox{
public:
    float x_min;
    float x_max;

    float y_min;
    float y_max;

    float z_min;
    float z_max;
    BoundingBox(vector<Face> faces);
    bool isInterectedByRay(Vector3f origin, Vector3f direction);
};

class Object{
private:
    Matrix3Xf localVerticesMatrix = Matrix3Xf::Constant( 3,0,1); //Matrice 3xN des coordonnées LOCALES des vertices de l'objet, où N
est le nombre de vertices.
    Matrix3Xf globalVerticesMatrix; //Matrice 3xN des coordonnées GLOBALES des vertices de l'objet, où N est le nombre de vertices.
    Origin origin; //Référentiel de l'objet. (PRIVE)
public:
    vector<Face> faces; //Tableau des faces de l'objet.
    vector<vector< int>> facesAttachedToVertex; //Associe à chaque indice de vertex la liste des indices des faces auxquelles elle
appartient (anciennement: vertFaces).
    Material material; //Matériau associé à l'objet. (inutile ?)
    void setOrigin(Origin newOrigin); // Modifie l'origine de l'objet.
    void setPosition(Vector3f newPos); // Modifie l'origine de l'objet.
    Origin getOrigin(); // Renvoie l'origine de l'objet.
    Matrix3Xf getVerticesMatrix(); // Renvoie la matrice GLOBALE de vertexs.
    vector<Face*> getFacesPointer(); // Renvoie une liste des pointeurs des faces de l'objet.
    Object(string filename, Origin org, Material mat);
    BinaryTree* binaryTree;
    BoundingBox* boundingBox;

```

```

/* Opérations sur les vertex et les faces */
int addVertex(Vector3f v); // Renvoie l'indice local associé au vertex %v qui vient d'être ajoutée
void modifyVertexOfFace( int local_face_id, int axis, int new_vertex_index); //
void removeFaceAttachedToVertex( int vertexIndex, int faceIndex);
int divideEdge(Face & face, int edge); // Renvoie l'indice du nouveau vertex ajouté correspondant au milieu du côté %edge
int addFace(Face f); // Renvoie l'indice local associé à la face qui vient d'être ajoutée
void subdivideFace( int local_face_id); // Subdivise la face selon son plus long côté.
void subdivideFaceFull( int local_face_id); // Subdivise la face en trois (nul).
void subdivideFaceFromEdge( int local_face_id, int edge); // Subdivise la face selon un côté donné.
vector<int> getNeighborOfFace( int local_face_id);
};

```

```

class LightSource{
public:
    int local_id; // Correspond à l'indice de la source dans la liste des sources de la scène
    int global_id; // Correspond au numéro de ligne dans c
    float strength;
    sf::Color color;
    int type;
    Origin origin;
    LightSource( int type = Point, Origin origin = Origin(), sf::Color color = White, float strength = 1.);
};

```

```

class Camera{
public:
    Origin origin; //Référentiel de la caméra.
    int width; //Largeur de la caméra (1280 par défaut).
    int height; //Hauteur de la caméra (720 par défaut).
    float distance; //Profondeur de la caméra (20).
    float elevation = 0;
    float fov; //Champ de vision de la caméra (angle en (radians?), 1 initialement).
    void rotate(int x=0, int y=0); // Tourne la caméra d'un angle x (gauche/droite) et y (haut/bas)
    void elevate(int z); // Tourne la caméra d'un angle x (gauche/droite) et y (haut/bas)
    Camera(Origin org = Origin(), float fieldOfView=1);
};

```

```

class Scene{
    private:
        vector<Object> objects; //Tableau des objets de la scène.
        vector<LightSource> lightsources; //Tableau des sources lumineuses de la scène.
    public:
        Scene(Camera cam = Camera());
        Camera camera; //Caméra associée à la scène.
        bool previewMode = true; // Affichage des faces multicolores.
        bool smoothShading = false;
        // Objets:
        void addObject(Object obj); // Ajoute un objet à la scene
        void removeObject(int index); // Supprime l'objet d'indice n°i à la scene
        vector<Object> & getObjects(); // Renvoie (une référence de) la liste des objets de la scene.
        // Sources lumineuses:
        void addLightSource(LightSource light); // Ajoute une source lumineuse à la scene
        // void removeLightSource(int index); // Supprime la source lumineuse d'indice n°i à la scene
        vector<LightSource> & getLightSources(); // Renvoie (une référence de) la liste des sources lumineuses de la scene.
        vector<LightSource*> getLightSourcesPointer(); // Renvoie (une référence de) la liste des sources lumineuses de la scene.
        // Faces:
        void initializeElementsId(); // Initialise les indices des faces dans la matrice C.
        vector<Face> getFaces(); // Renvoie la liste de toutes les faces de la scene
        vector<Face*> getFacesPointer(); // Renvoie la liste de toutes les pointeurs vers les faces de la scene
        void subdivideFaces(float lengthThreshold = 1.);
        void smartSubdivision();
        void test(int id);
};

```


world.cpp

```
#include <math.h>
#include <fstream>
#include <assert.h>

#include "world.h"

using Eigen::Matrix3Xf;
using Eigen::Vector3f;
using Eigen::VectorXf;

typedef Eigen::Matrix<bool, Eigen::Dynamic, Eigen::Dynamic> MatrixXb;

using namespace std;
using namespace Eigen;

const float FOV_ANGLE = 3.1415/2;
const int SCREEN_HEIGHT = 720; // 720 900
const int SCREEN_WIDTH = 1280; // 1280 1600
const float SCREEN_HEIGHT_FLOAT = static_cast<float>(SCREEN_HEIGHT);
const float SCREEN_WIDTH_FLOAT = static_cast<float>(SCREEN_WIDTH);
const float Z_FAR = 700;
const float Z_NEAR = 0.2;

// Divise une chaine de caractères selon un delimitateur
vector<string> split(string s, string delimiter) {
    size_t pos_start = 0, pos_end, delim_len = delimiter.length();
    string token;
    vector<string> res;

    while ((pos_end = s.find(delimiter, pos_start)) != string::npos) {
        token = s.substr (pos_start, pos_end - pos_start);
        pos_start = pos_end + delim_len;
        res.push_back (token);
    }

    res.push_back(s.substr(pos_start));
    return res;
};
```

```

/* Renvoie les coordonnées du vecteur vect après rotation autour de X */
Matrix3Xf rotateX(Matrix3Xf vect, float angle){
    Matrix3f rotMat{{ 1, 0          , 0          },
                    { 0, cos(angle), -sin(angle)},
                    { 0, sin(angle), cos(angle) }};
    return rotMat * vect;
}

/* Renvoie les coordonnées du vecteur vect après rotation autour de Y */
Matrix3Xf rotateY(Matrix3Xf vect, float angle){
    Matrix3f rotMat{{cos(angle) , 0, sin(angle)},
                    { 0          , 1, 0          },
                    {-sin(angle), 0, cos(angle)}};
    return rotMat * vect;
}

/* Renvoie les coordonnées du vecteur vect après rotation autour de Z */
Matrix3Xf rotateZ(Matrix3Xf vect, float angle){
    Matrix3f rotMat{{cos(angle), -sin(angle), 0},
                    {sin(angle), cos(angle) , 0},
                    { 0          , 0          , 1}};
    return rotMat * vect;
}

Matrix3Xf rotate3d(Matrix3Xf vector, Vector3f rotVect){
    return rotateX(rotateY(rotateX(vector, rotVect( 0)), rotVect(1)), rotVect(2));
}

Matrix3Xf translate(Matrix3Xf vector, Vector3f t){
    for (int i = 0; i < vector.cols(); i++)
    {
        vector.col(i) = vector.col(i) + t;
    }
    return vector;
}

/*Renvoie la distance entre deux points de l'espace sous la forme d'un vecteur 3D*/
Vector3f distanceVect(Vector3f v1, Vector3f v2){
    return v2 - v1;
}

```

```

/*Renvoie la norme du vecteur V*/
float norme(Vector3f v){
    return sqrtf(v(0)*v(0) + v(1)*v(1) + v(2)*v(2));
}

/*Renvoie la distance entre deux points de l'espace*/
float distance(Vector3f v1, Vector3f v2){
    return norme(distanceVect(v1, v2));
}

/*Renvoie la norme du vecteur V au carré*/
float normeSquare(Vector3f v){
    return v(0)*v(0) + v(1)*v(1) + v(2)*v(2);
}

/*Renvoie le cosinus de l'angle entre les deux vecteurs*/
float cosAngle(Vector3f v1, Vector3f v2){
    float s = scalar(v1,v2);
    return s/(norme(v1)*norme(v2));
}

/* Calcule le produit vectoriel de deux vecteurs.*/
Vector3f cross(Vector3f v1, Vector3f v2){
    return Vector3f{
        {v1(1)*v2(2) - v1(2)*v2(1)},
        {v1(2)*v2(0) - v1(0)*v2(2)},
        {v1(0)*v2(1) - v1(1)*v2(0)};
}

/* Calcule le produit scalaire de deux vecteurs.*/
float scalar(Vector3f v1, Vector3f v2){
    return (v1(0)*v2(0) + v1(1)*v2(1) + v1(2)*v2(2));
}

/*Renvoie les nouvelles coordonnées du vecteur V dans la base du monde, en supposant que ses coordonnées soient données dans la base Origin */
Matrix3Xf coordonateInWorld(Matrix3Xf v, Origin origin){
    return translate(rotate3d(v, origin.rot), origin.pos);
}

```

```

/* Renvoie les nouvelles coordonnées d'un vecteur de coordonnées COORD dans la base de la CAMERA, en supposant que ses coordonnées
soient données dans la base du monde */
Matrix3Xf coordonateForCamera(Matrix3Xf v, Camera cam){
    return rotate3d(translate(v, Vector3f {{-cam.origin.pos( 0)}, {-cam.origin.pos( 1)}, {-cam.origin.pos( 2)}}), -1*cam.origin.rot);
}

// /* STRUCTURES */

// // Matériau:

Material::Material(sf::Color col, bool emit){
    emissive = emit;
    color = col;
};

// Face:

Face::Face(int v1, int v2, int v3, Material mat){
    material = mat;
    lightAmount = sf::Color( 0,0,0);
    previewColor = sf::Color(rand()% 256,rand()%256,rand()%256);
    verticesIndex[ 0] = v1;
    verticesIndex[ 1] = v2;
    verticesIndex[ 2] = v3;
    highlight = false;
};

Vector3f Face::normalVector(){
    if (!normal__calculated){
        Vector3f edge1 = distanceVect(vertex( 1), vertex(0));
        Vector3f edge2 = distanceVect(vertex( 2), vertex(1));
        normal__ = cross(edge1,edge2);
        normal__calculated = true;
    }
    return normal__;
}

```

```
Vector3f Face::center(){
    if (!center__calculated){
        center__ = Vector3f {{{(vertex( 0)(0) + vertex(1)(0) + vertex(2)(0))/3},
                             {(vertex( 0)(1) + vertex(1)(1) + vertex(2)(1))/3},
                             {(vertex( 0)(2) + vertex(1)(2) + vertex(2)(2))/3}}};
        center__calculated = true;
    }
    return center__;
}
```

```
float Face::area(){
    if (!area__calculated){
        area__ = norme(normalVector())/ 2;
        area__calculated = true;
    }
    return area__;
}
```

```
Object& Face::parentObject(){
    return scenePointer->getObjects()[objectIndex];
}
```

```
Vector3f Face::vertex( int i){
    assert(i>=0 && i<3);
    assert(scenePointer!= NULL);
    return parentObject().getVerticesMatrix().col(verticesIndex[i]);
}
```

```
int Face::vertexIndex( int i){
    assert(i>=0 && i<3);
    return verticesIndex[i];
}
```

```
bool Face::isIntersectedByRay(Vector3f rayOrigin, Vector3f rayDirection){
    const float EPSILON = 0.0000001;
    Vector3f vertex0 = vertex( 0);
    Vector3f vertex1 = vertex( 1);
    Vector3f vertex2 = vertex( 2);
    Vector3f edge1, edge2, h, s, q;
    float a, f, u, v;
    edge1 = vertex1 - vertex0;
    edge2 = vertex2 - vertex0;
    h = rayDirection.cross(edge2);
    a = edge1.dot(h);

    if (a > -EPSILON && a < EPSILON) return false; // Le rayon est parallèle à la face

    f = 1.0 / a;
    s = rayOrigin - vertex0;
    u = f * h.dot(s);

    if (u < 0.0 || u > 1.0) return false;

    q = s.cross(edge1);
    v = f * rayDirection.dot(q);

    if (v < 0.0 || u + v > 1.0) return false;

    float t = f * edge2.dot(q); // On calcule le paramètre de collision
    return (t > 0.001 && t < 0.99999);
};
```

```
void Face::reload()
{
    normal__calculated = false;
    center__calculated = false;
    area__calculated = false;
}

void Face::modifyVertex( int vertexToModify, int newVertexIndex)
{
}

// Longueur du cote le plus long d'une face.
float lengthLongestEdge(Face & face){
    int res = 2;
    float length = distance(face.vertex( 0), face.vertex(1));
    if(distance(face.vertex( 1), face.vertex(2)) > length)
    {
        res = 0;
        length = distance(face.vertex( 1), face.vertex(2));
    }
    if(distance(face.vertex( 2), face.vertex(0)) > length)
    {
        res = 1;
        length = distance(face.vertex( 2), face.vertex(0));
    }
    return length;
};

// Origine:

Origin::Origin(Vector3f position, Vector3f rotation){
    pos = position;
    rot = rotation;
};
```

```
// BoundingBox:
```

```
BoundingBox::BoundingBox(vector<Face> faces)
```

```
{  
    x_min = INFINITY;    y_min = INFINITY;    z_min = INFINITY;  
    x_max = -INFINITY;   y_max = -INFINITY;   z_max = -INFINITY;  
    for (int i = 0; i < faces.size(); i++)  
    {  
        x_min = min({x_min, faces[i].vertex( 0)(0), faces[i].vertex( 1)(0), faces[i].vertex( 2)(0)});  
        x_max = max({x_max, faces[i].vertex( 0)(0), faces[i].vertex( 1)(0), faces[i].vertex( 2)(0)});  
        y_min = min({y_min, faces[i].vertex( 0)(1), faces[i].vertex( 1)(1), faces[i].vertex( 2)(1)});  
        y_max = max({y_max, faces[i].vertex( 0)(1), faces[i].vertex( 1)(1), faces[i].vertex( 2)(1)});  
        z_min = min({z_min, faces[i].vertex( 0)(2), faces[i].vertex( 1)(2), faces[i].vertex( 2)(2)});  
        z_max = max({z_max, faces[i].vertex( 0)(2), faces[i].vertex( 1)(2), faces[i].vertex( 2)(2)});  
    }  
};
```

```
// Intersection Rayon/ plan orthogonal à l'axe %axis, de coordonnée %center le long de cet axe
```

```
float intersectionPlanAxis(Vector3f origin, Vector3f direction, int axis, int center)
```

```
{  
    float t = direction[axis]; // Produit scalaire normale du plan et direction du rayon  
    if (abs(t) < 0.00001) return INFINITY; // Si il est parallèle on renvoie l'infini.  
    return (center - origin[axis])/t;  
}
```

```
bool BoundingBox::isInterectedByRay(Vector3f origin, Vector3f direction)
```

```
{  
    float coef = intersectionPlanAxis(origin, direction, 0, x_min);  
    if (coef > 0.01 && coef < 0.999 && origin(1)+ direction(1)*coef <= y_max && origin( 1)+ direction(1)*coef >= y_min && origin( 2)+  
direction(2)*coef <= z_max && origin( 2)+ direction(2)*coef >= z_min) return true;  
  
    coef = intersectionPlanAxis(origin, direction, 0, x_max);  
    if (coef > 0.01 && coef < 0.999 && origin(1)+ direction(1)*coef <= y_max && origin( 1)+ direction(1)*coef >= y_min && origin( 2)+  
direction(2)*coef <= z_max && origin( 2)+ direction(2)*coef >= z_min) return true;  
  
    coef = intersectionPlanAxis(origin, direction, 1, y_min);  
    if (coef > 0.01 && coef < 0.999 && origin(0)+ direction(0)*coef <= x_max && origin( 0)+ direction(0)*coef >= x_min && origin( 2)+  
direction(2)*coef <= z_max && origin( 2)+ direction(2)*coef >= z_min) return true;
```



```

coef = intersectionPlanAxis(origin, direction1, y_max);
    if ( coef > 0.01 && coef < 0.999 && origin(0)+ direction(0)*coef <= x_max && origin(0)+ direction(0)*coef >= x_min && origin(1)+ direction(1)*coef <= y_max &&
origin(1)+ direction(1)*coef >= y_min) return true;

    coef = intersectionPlanAxis(origin, direction2, z_min);
    if ( coef > 0.01 && coef < 0.999 && origin(0)+ direction(0)*coef <= x_max && origin(0)+ direction(0)*coef >= x_min && origin(1)+ direction(1)*coef <= y_max &&
origin(1)+ direction(1)*coef >= y_min) return true;

    coef = intersectionPlanAxis(origin, direction2, z_max);
    if ( coef > 0.01 && coef < 0.999 && origin(0)+ direction(0)*coef <= x_max && origin(0)+ direction(0)*coef >= x_min && origin(1)+ direction(1)*coef <= y_max &&
origin(1)+ direction(1)*coef >= y_min) return true;

    return false;
}

// Objet:

// Ajoute le nouveau vertex %v à l'objet en évitant les doublons
int Object::addVertex(Vector3f v)
{
    for(int i = 0; i < localVerticesMatrix.cols(); i++)
    {
        if(v.isApprox(localVerticesMatrix.col(i)))return i;
    }
    // On rajoute un vecteur
    facesAttachedToVertex.push_back(vector<int>());
    localVerticesMatrix.conservativeResize(NoChange, localVerticesMatrix.cols()+1); // Rajoute une colonne
    localVerticesMatrix.col(localVerticesMatrix.cols()-1) = v; // Rempli la dernière colonne avec le vertex lu
    setOrigin(origin);
    return localVerticesMatrix.cols()-1;
}

// Ajoute la nouvelle face %f à l'objet
int Object::addFace(Face f)
{
    assert(faces.size() !=0);
    f.scenePointer = faces[0].scenePointer;
    f.objectIndex = faces[0].objectIndex;
    facesAttachedToVertex[f.vertexIndex0].push_back(faces.size());
    facesAttachedToVertex[f.vertexIndex1].push_back(faces.size());
    facesAttachedToVertex[f.vertexIndex2].push_back(faces.size());
    faces.push_back(f);
    setOrigin(origin);
    return faces.size() - 1;
}

```

```

// Créé un nouveau vecteur correspondant au milieu du coté %edge de %face
int Object::divideEdge(Face & face, int edge)
{
    Vector3f newVector = (face.vertex((edge+ 1)%3) + face.vertex((edge+ 2)%3))/2;
    return addVertex(newVector);
}

// Renvoie la liste des pointeurs des faces de l'objet
vector<Face*> Object::getFacesPointer(){
    vector<Face*> faceList;
    for (int i = 0; i < faces.size(); i++)
    {
        faceList.push_back(&(faces[i]));
    }
    return faceList;
}

// Supprime l'indice %faceIndex de la liste des dépendances du vertex d'indice %vertexIndex
void Object::removeFaceAttachedToVertex( int vertexIndex, int faceIndex)
{
    for (int i = 0; i < facesAttachedToVertex[vertexIndex].size(); i++)
    {
        if (facesAttachedToVertex[vertexIndex][i] == faceIndex) {
            (facesAttachedToVertex[vertexIndex]).erase(facesAttachedToVertex[vertexIndex].begin() + i);
        }
    }
}

void Object::modifyVertexOfFace( int local_face_id, int axis, int new_vertex_index)
{
    assert(new_vertex_index < localVerticesMatrix.cols());
    faces[local_face_id].verticesIndex[axis] = new_vertex_index;
    removeFaceAttachedToVertex(faces[local_face_id].vertexIndex(axis), local_face_id);
    facesAttachedToVertex[new_vertex_index].push_back(local_face_id);
    faces[local_face_id].reload();
}

void Object::setOrigin(Origin newOrigin){
    origin = newOrigin;
    globalVerticesMatrix = coordonateInWorld(localVerticesMatrix, origin);
}

```

```

void Object::setPosition(Vector3f vect){
    Vector3f newPos = vect;
    origin.pos = newPos;
    globalVerticesMatrix = coordonateInWorld(localVerticesMatrix, origin);
}

Origin Object::getOrigin(){
    return origin;
}

Matrix3Xf Object::getVerticesMatrix(){
    return globalVerticesMatrix;
}

void Object::subdiviseFaceFromEdge( int local_face_id, int edge){
    assert(local_face_id < faces.size());
    Face& faceToCut = faces[local_face_id];
    int newVertexIndex = divideEdge(faceToCut, edge);
    Face secondFace = Face(faceToCut.vertexIndex(edge), newVertexIndex, faceToCut.vertexIndex((edge+ 2)%3), faceToCut.material);
    modifyVertexOfFace(local_face_id, (edge+ 2)%3, newVertexIndex);
    addFace(secondFace);
}

void Object::subdiviseFace( int local_face_id){
    Face& face = faces[local_face_id];
    int edge = 2;
    float length = distance(face.vertex( 0), face.vertex(1));
    if(distance(face.vertex( 1), face.vertex(2)) > length)
    {
        edge = 0;
        length = distance(face.vertex( 1), face.vertex(2));
    }
    if(distance(face.vertex( 2), face.vertex(0)) > length)
    {
        edge = 1;
        length = distance(face.vertex( 2), face.vertex(0));
    }
    subdiviseFaceFromEdge(local_face_id, edge);
}

```

```

void Object::subdivideFaceFull(int local_face_id){
    Face& face = faces[local_face_id];
    // On rajoute le centre
    int center_0 = divideEdge(face, 0);
    int center_1 = divideEdge(face, 1);
    int center_2 = divideEdge(face, 2);
    Face newFace_cent = Face(center_0, center_1, center_2, face.material);
    addFace(newFace_cent);
    Face newFace_1 = Face(center_2, face.vertexIndex(1), center_0, face.material);
    addFace(newFace_1);
    Face newFace_2 = Face(center_1, center_0, face.vertexIndex(2), face.material);
    addFace(newFace_2);
    modifyVertexOfFace(local_face_id, 2, center_1);
    modifyVertexOfFace(local_face_id, 1, center_2);
}

vector<int> Object::getNeighborOfFace(int local_face_id){
    vector<int> res;
    for (int v1 = 0; v1 < 3; v1++)
    {
        int v2 = (v1+1)%3;
        vector<int> s1 = facesAttachedToVertex[faces[local_face_id].vertexIndex(v1)];
        vector<int> s2 = facesAttachedToVertex[faces[local_face_id].vertexIndex(v2)];
        bool exit = false;
        for (int i1 = 0; i1 < s1.size() && !exit; i1++)
        {
            for (int i2 = 0; i2 < s2.size() && !exit; i2++)
            {
                if (s1[i1] == s2[i2]){
                    res.push_back(s1[i1]);
                    exit = true;
                }
            }
        }
    }
    vector<int> res2;
    for (size_t i = 0; i < 3; i++)
    {
        vector<int> y = facesAttachedToVertex[faces[local_face_id].vertexIndex(i)];
        res2.insert(res2.end(), y.begin(), y.end());
    }
    return res2;
}

```

```

Object::Object(string filename, Origin org, Material mat){
    srand(time(NULL));
    material = mat;
    ifstream file(filename);
    assert(!file.fail());
    string line; // Construction de l'objet à partir du fichier
    while (getline(file, line)){
        vector<string> tab = split(line, " "); // line:"o 0.1 0.2 0.3" devient tab:["o", "0.1", "0.2", "0.3"]
        if (tab[0] == "v")
        {
            facesAttachedToVertex.push_back(vector< int>());
            localVerticesMatrix.conservativeResize(NoChange, localVerticesMatrix.cols()+ 1); // Rajoute une colonne
            localVerticesMatrix.col(localVerticesMatrix.cols()- 1) = Vector3f(stof(tab[ 1]), stof(tab[ 2]), stof(tab[ 3])); // Rempli la
dernière colonne avec le vertex lu
        } else if(tab[0] == "f")
        {
            vector<string> vert1 = split(tab[ 1], "/");
            vector<string> vert2 = split(tab[ 2], "/");
            vector<string> vert3 = split(tab[ 3], "/");
            Face face(stoi(vert1[ 0])-1,stoi(vert2[ 0])-1 ,stoi(vert3[ 0])-1, mat);
            faces.push_back(face);
            facesAttachedToVertex[stoi(vert1[ 0])-1].push_back(faces.size()- 1);
            facesAttachedToVertex[stoi(vert2[ 0])-1].push_back(faces.size()- 1);
            facesAttachedToVertex[stoi(vert3[ 0])-1].push_back(faces.size()- 1);
        }
    }
    setOrigin(org); // Doit être à la fin pour actualiser la matrice
};

// Sources lumineuses:

LightSource::LightSource( int type, Origin origin, sf::Color color, float strength){
    this->origin = origin;
    this->color = color;
    this->strength = strength;
    this->type = type;
}

```

```
// Camera:
```

```
Camera::Camera(Origin org, float fieldOfView){  
    width = SCREEN_WIDTH;  
    height = SCREEN_HEIGHT;  
    distance = 20;  
    origin = org;  
    fov = fieldOfView;  
    rotate(0, 0);  
};
```

```
void Camera::rotate( int x, int y){  
    float SENSIBILITY = 0.002;  
    float rotX = -x*SENSIBILITY;  
    float rotY = y*SENSIBILITY;  
    origin.rot = origin.rot + Vector3f {{ 0},{rotX},{rotY}};  
    origin.pos = distance*Vector3f(cos(-origin.rot( 2))*sin(origin.rot( 1)),  
                                   sin(-origin.rot( 2)) + elevation/distance,  
                                   cos(-origin.rot( 2))*cos(origin.rot( 1)));  
}
```

```
void Camera::elevate( int z){  
    float SENSIBILITY = 1;  
    elevation += z*SENSIBILITY;  
    rotate();  
}
```

```
// Scene:
```

```
Scene::Scene(Camera cam){  
    camera = cam;  
    objects = vector<Object>();  
};
```

```
void Scene::addObject(Object obj){
    objects.push_back(obj);
    for (int i = 0; i < objects.size(); i++)
    {
        Object& object = objects[i];
        for (Face* face : object.getFacesPointer())
        {
            face->scenePointer = this;
            face->objectIndex = i;
        }
    }
    initializeElementsId();
    Object& object = objects[objects.size()- 1];
    object.boundingBox = new BoundingBox(object.faces);
    object.binaryTree = buildBinaryTreeFromList(object.getFacesPointer());
};
```

```
void Scene::removeObject( int index){
    objects.erase(objects.begin()+index);
    initializeElementsId();
}
```

```
vector<Object> & Scene::getObjects(){
    return objects;
};
```

```
void Scene::addLightSource(LightSource light){
    lightsources.push_back(light);
    initializeElementsId();
}
```

```
vector<LightSource> & Scene::getLightSources(){
    return lightsources;
}
```

```
vector<LightSource*> Scene::getLightSourcesPointer(){
    vector<LightSource*> lightList;
    for(int i = 0; i < lightsources.size(); i++)
    {
        lightList.push_back(&(lightsources[i]));
    }
    return lightList;
}
```

```
vector<Face> Scene::getFaces(){
    vector<Face> faceList;
    for(Object& obj : getObject())
    {
        faceList.insert(faceList.end(), obj.faces.begin(), obj.faces.end());
    }
    return faceList;
}
```

```
void Scene::initializeElementsId(){
    int counter = 0;
    for (Object& object : objects)
    {
        for (Face& face : object.faces)
        {
            face.id = counter;
            counter++;
        }
    }
    int local_counter = 0;
    for (LightSource& light : lightsources)
    {
        light.local_id = local_counter;
        light.global_id = counter;
        counter ++;
        local_counter ++;
    }
}
```



```

vector<Face*> Scene::getFacesPointer(){
    vector<Face*> faceList;
    for(Object& obj : getObjects())
    {
        vector<Face*> tmp = obj.getFacesPointer();
        faceList.insert(faceList.end(), tmp.begin(), tmp.end());
    }
    return faceList;
}

void Scene::subdivideFaces( float lengthThreshold){
    for (int i = 0; i < objects.size(); i++)
    {
        Object& object = objects[i];
        int j = 0;
        while (j < object.faces.size())
        {
            Face& faceToCut = object.faces[j];
            if(lengthLongestEdge(faceToCut) > lengthThreshold)
            {
                object.subdivideFace(j);
            }
            else
            {
                j++;
            }
        }
        object.binaryTree = buildBinaryTreeFromList(object.getFacesPointer());
    }
    initializeElementsId();
}

```

```

void Scene::smartSubdivision()
{
    MatrixXb faceLightVisibily = MatrixXb::Constant(getLightSources().size(), getFaces().size(), false);
    cout << "source lumineuses: " << getLightSources().size() << endl;
    for(Face* face : getFacesPointer())
    {
        for (LightSource* light : getLightSourcesPointer())
        {
            Vector3f distanceVector = distanceVect(face->center(), light->origin.pos);
            faceLightVisibily(light->local_id, face->id) = false; // !rayIsIntersectedInScene(face->center(), distanceVector, this);
        }
    }
    vector<vector<bool>> sub_list;
    for (int obj_index = 0; obj_index < objects.size(); obj_index++)
    {
        Object& object = objects[obj_index];
        vector<bool> subObject;
        for (int j = 0; j < object.faces.size(); j++)
        {
            bool added = false;
            subObject.push_back(false);
            Face& face = object.faces[j];
            assert(object.getNeighborOfFace(j).size() > 0);
            for(int k = 0; k < object.getNeighborOfFace(j).size(); k++)
            {
                int nbh = object.getNeighborOfFace(j)[k]; // Optimisable.
                assert(nbh < object.faces.size());
                if (faceLightVisibily.col(face.id) != faceLightVisibily.col(object.faces[nbh].id) && cosAngle(face.normalVector(),
object.faces[nbh].normalVector()) > 0.99999 && false)
                {
                    vector<int> tuple = {obj_index, j};
                    subObject[subObject.size()- 1] = true;
                    break;
                }
            }
        }
        assert(subObject.size() == object.faces.size());
        sub_list.push_back(subObject);
    }
}

```

```
assert(sub_list.size() == objects.size());
for (int i = 0; i < sub_list.size(); i++)
{
    for (int j = 0; j < sub_list[i].size(); j++)
    {
        if (sub_list[i][j]) objects[i].subdivideFaceFull(j);
    }
}
initializeElementsId();
for (Object& object : objects)
{
    object.binaryTree = buildBinaryTreeFromList(object.getFacesPointer());
}
initializeElementsId();
}
```

```

void Scene::test(int id)
{
    MatrixXb faceLightVisibily = MatrixXb::Constant(getLightSources().size(), getFaces().size(), false);
    cout << "source lumineuses: " << getLightSources().size() << endl;
    for(Face* face : getFacesPointer())
    {
        for (LightSource* light : getLightSourcesPointer())
        {
            Vector3f distanceVector = distanceVect(face->center(), light->origin.pos);
            faceLightVisibily(light->local_id, face->id) = !rayIsIntersectedInScene(face->center(), distanceVector);
        }
    }
    vector<vector<int>> todo; // (objet, face à subdiviser)
    int cpt = 0;
    for (int obj_index = 0; obj_index < objects.size(); obj_index++)
    {
        Object& object = objects[obj_index];
        for (int j = 0; j < object.faces.size(); j++)
        {
            Face& face = object.faces[j];

            bool exit = false;
            assert(object.getNeighborOfFace(j).size() > 0);
            for(int k = 0; k < object.getNeighborOfFace(j).size(); k++)
            {
                int nbh = object.getNeighborOfFace(j)[k]; // Optimisable.
                if (cpt==id)
                {
                    Face& facenb = object.faces[nbh];
                    facenb.lightAmount = sf::Color( 255, 0);
                    face.lightAmount = sf::Color(65, 0, 0);
                }
                if (faceLightVisibily.col(face.id) != faceLightVisibily.col(object.faces[nbh].id) && cosAngle(face.normalVector(), object.faces[nbh].normalVector()) >
0.99999)
                {
                    vector<int> tuple = {obj_index, j};
                    todo.push_back(tuple);
                    exit = true;
                    break;
                }
            }
            cpt++;
        }
    }
    cout << "todo size: " << todo.size() << endl;
}

```