

EnLight'INT

Outil pédagogique permettant de visualiser différents effets de lumière dans une scène 3D en temps réel.

Antoine MORRIER
Cédric PENAFIEL
David THERY-BULHA
Victor ROUQUETTE

Responsable : M.PREDA

Version 0.1-a



20 mars 2015

Sommaire

I. Analyse du problème	4
a- Objectif.....	4
b- Considérations théoriques.....	4
c- Outils et bibliothèques informatiques	6
II. Spécifications fonctionnelles.....	7
a- Configuration technique.....	7
b- Moteur de rendu	7
c- Interface graphique	9
d- Tests.....	9
III. Code source actuel.....	10
a- Interface graphique	10
b- Moteur de rendu	12

Introduction

Ce document constitue un pré-rapport, mis à jour en fonction de l'avancement du projet, dans le cadre du module de première année CSC 3502, nommé projet informatique. Il présente le programme que nous voulons créer, ses spécifications et notre démarche pour le réaliser.

Le but est de créer un moteur de rendu 3D en temps réel capable d'utiliser différents effets de lumières. Celui-ci sera couplé avec une interface graphique permettant de contrôler ces effets et de voir les changements apportés à la scène de manière interactive.

Ce livrable est constitué de l'analyse du problème, d'un cahier des charges et de la description des spécifications fonctionnelles du programme.

Une dernière partie présente un exemple de code source à l'état actuel.

I. Analyse du problème

a- Objectif

Que ce soit pour les jeux vidéo, le cinéma ou la CAO, l'illumination de modèles en 3 dimensions est de plus en plus utilisée et de plus en plus complexe au fur et à mesure que l'on essaye de se rapprocher de la réalité.

Avec ce projet nous poursuivons un double objectif, à la fois créer un moteur de rendu avec illumination globale et surtout permettre à quiconque de comprendre comment celle-ci fonctionne. L'idée est donc de se rapprocher le plus possible d'un moteur graphique malléable, avec, ou sans connaissance de la programmation. Il permettra donc de se familiariser et de visualiser l'effet de différents outils et méthodes d'illumination afin d'en améliorer la compréhension.

En effet, une personne sans aucune notion de programmation pourra utiliser le moteur graphique à l'aide d'une interface homme machine et ainsi modifier facilement la plupart de paramètres. Ceux-ci seront de plus expliqués en détails. Une personne qui a des connaissances en programmation pourra tout, elle aussi, utiliser ce moteur directement en C++.

La partie interface graphique sera pensée de la même manière qu'un plugin à « patcher » sur notre moteur.

b- Considérations théoriques

Le principe de l'illumination globale est de prendre en compte deux paramètres dans le calcul de la couleur de chaque pixel : Celle qui provient directement de la source lumineuse et aussi celle qui est due aux autres objets de la scène.

Pour cela on calcule la radiance qui représente l'énergie émise par un point. Celle-ci découle de l'équation suivante (elle permet en plus de tenir compte des réflexions et réfractions) :

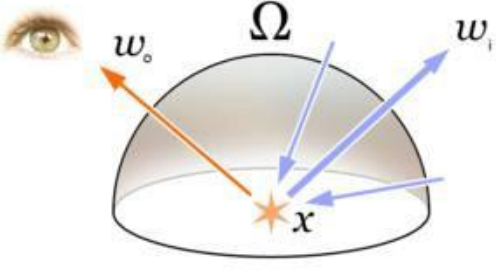
$$L_o(\mathbf{x}, \omega_o, \lambda, t) = L_e(\mathbf{x}, \omega_o, \lambda, t) + \int_{\Omega} f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t) L_i(\mathbf{x}, \omega_i, \lambda, t) (\omega_i \cdot \mathbf{n}) d\omega_i$$


FIGURE 1 : THE RENDERING EQUATION : JAMES T. KAJIYA (SIGGRAPH 1986)

Le premier terme de cette équation est la radiance sortante au point x , c'est à dire la « couleur » vu depuis la caméra. Ce terme-là sera donc la « couleur » finale du pixel.

Le second terme de cette équation est la radiance émise au point x , il est nul lorsque ce n'est pas une source lumineuse, sinon, il est égale à la « couleur » de la lumière.

Le troisième terme, est une intégrale prise sur un hémisphère (cf. figure 2) qui représente tout ce qui est illumination (éclairage direct, réflexions, illumination globale), la réfraction sera alors une donnée « ajoutée » à cette équation.

L'objectif sera donc d'approximer cette équation le mieux possible et en temps réel.

Pour simplifier les calculs, nous allons utiliser deux types de réflexions : La réflexion diffuse (la réflexion d'un rayon incident donne lieu à une multitude de rayons émergents, on peut ainsi tenir compte des objets réfléchissants : si on prend un mur rouge sur une route blanche, et que l'on éclaire le mur, de la lumière rouge, très faible, sera réfléchi sur la route), et spéculaire (la réflexion d'un rayon incident ne donne lieu qu'à un seul émergent, comme sur un miroir).

La réflexion diffuse correspond à notre modèle d'illumination globale et sera calculé en « rajoutant » des lumières virtuelles dans la scène.

La réflexion spéculaire correspond à notre modèle de réflexion pure et sera calculé en utilisant une technique d'environnement mapping.

c- Outils et bibliothèques informatiques

Nous allons utiliser le langage C++, l'API OpenGL 4.4, la bibliothèque SDL2, le Framework Qt5, et les bibliothèques GLEW, GLM et Assimp.

Afin de mieux s'organiser sur ce projet, nous allons également utiliser Git (hébergé sur GitHub afin de pouvoir publier notre documentation en ligne) couplé à Doxygen. Le premier nous permettra de travailler chacun sur sa partie de manière efficace en parallèle, et le second nous permettra de fournir une documentation complète de notre projet. Étant donné que nous développons une bibliothèque en premier lieu, cela est plus que nécessaire.

À l'instar du C, le C++ présente les pointeurs comme fonctionnalité ce qui en fait un langage de bas niveau, cependant il possède également de nombreuses fonctionnalités haut niveau qui nous seront très utiles dans le développement de notre projet.

La première est la RAI (Acquisition de la ressource à l'initialisation), grâce au paradigme objet cela permet d'éviter des fuites de mémoires liées aux allocations dynamiques.

- OpenGL 4.4 nous permettra d'avoir un « plein » contrôle sur le GPU. En effet, cette API nous permet un accès relativement bas niveau, ce qui permettra de détacher au maximum l'utilisation du CPU dans le rendu. Le CPU qui est le principal goulot d'étranglement dans un moteur de rendu en temps réel.
- La SDL2 nous permettra de créer une fenêtre, d'y initialiser un contexte OpenGL, et de pouvoir gérer les événements utilisateur (principalement, bouger la caméra).
- Qt nous permettra d'avoir une interface graphique (GUI : Graphic User Interface) avec des boutons, des curseurs, etc... afin de paramétrer les éléments qui vont constituer notre scène(Lumière,...).
- GLEW permet de charger toutes les fonctions OpenGL sous forme de pointeurs de fonctions.
- GLM est une bibliothèque mathématique permettant de gérer les matrices, quaternions et autres vecteurs que nous aurons à manipuler dans nos équations.
- Assimp quant à elle permet de charger en mémoire vive les modèles aux formats 3D (OBJ, MD2, ...).

II. Spécifications fonctionnelles

a- Configuration technique

Le but est de créer une interface simple à utiliser pour un utilisateur n'ayant aucune expérience avec le rendu 3D mais lui permettant de tester des fonctionnalités.

Le programme devra fonctionner sous Linux, Windows et MacOS (sous réserve de mise à jour des drivers pour gérer OpenGL 4.4).

Il requiert toutefois une carte graphique supportant OpenGL 4.4 (cela concerne la plupart des GPU sortie après 2012). Si ce n'est pas le cas le programme en informera l'utilisateur.

Afin d'avoir une expérience agréable nous souhaitons obtenir un rendu en moins de 50ms, mais nous nous accordons un rendu d'environ 200ms (toutes options comprises : réflexion spéculaire, ombre douce, occlusion ambiante et illumination globale).

b- Moteur de rendu

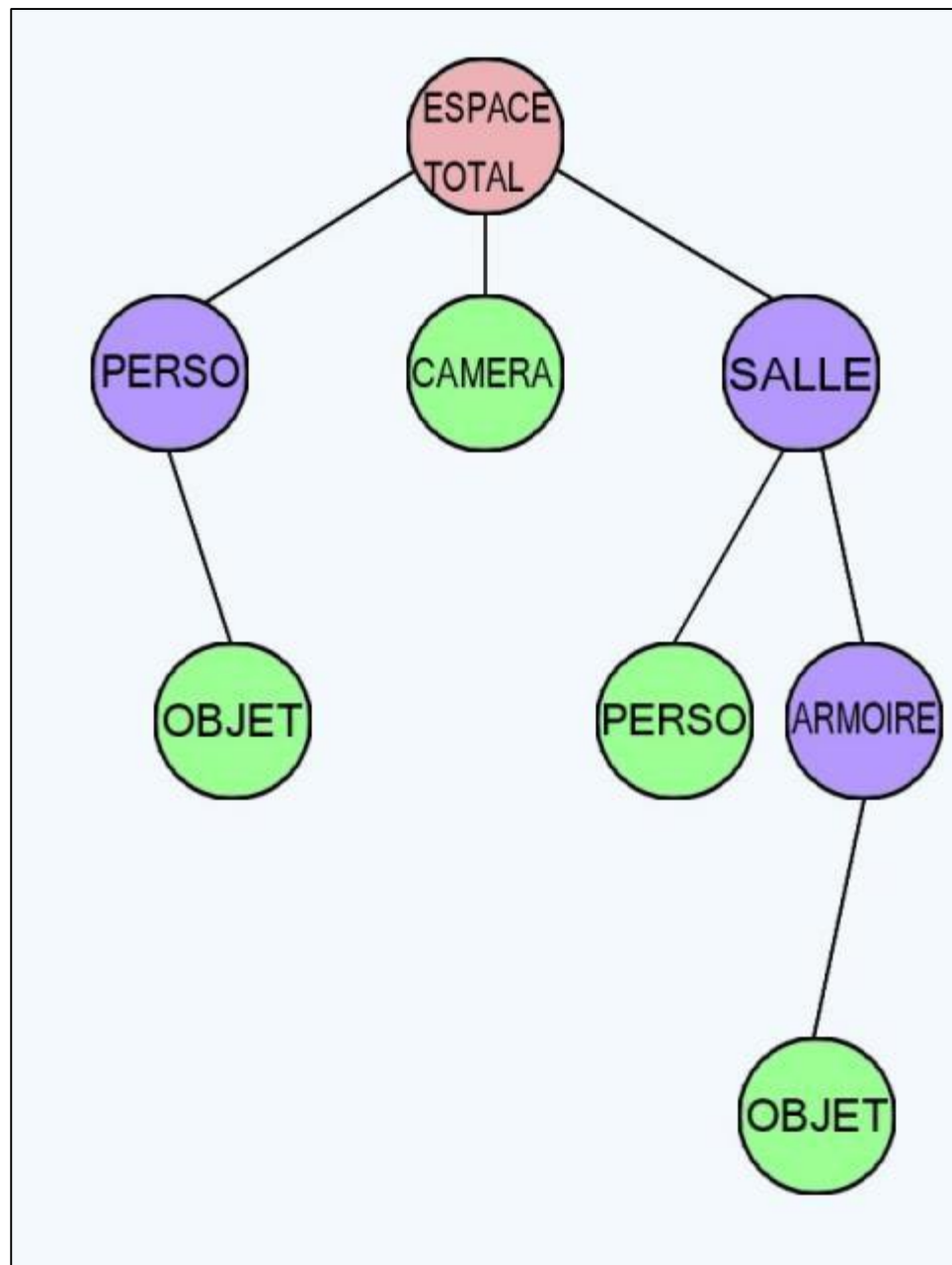
La partie moteur graphique comportera les modules suivants :

- Lecture et affichage de modèles 3D.
- Chargement automatique des textures des modèles 3D.
- Système de dépendances des objets.
- Éclairage direct.
- Gestion des ombres.
- Gestion des réflexions, et réfractions.
- Première approximation de l'illumination globale (jusqu'au second rebond de la lumière).
- L'occlusion ambiante, comme première approximation des ombres liées à l'illumination globale.

Note :

- La partie chargement des fichiers se fera à l'aide des bibliothèques Assimp et SDL.
- Le système de dépendances des objets sera représenté sous un graphe de scène. C'est-à-dire comme un arbre, les éléments fils dépendant des données relatives de leurs parents (voir figure 2).

FIGURE 2 : LE
GRAPHE DE
SCENE



c- Interface graphique

Toutes les options présentées ci-dessus seront réglables grâce à notre interface graphique. Celle-ci offrira ces différentes fonctionnalités :

- Activation l'occlusion ambiante
- Activation de l'illumination globale (rebonds de la lumière)
- Activation de la réflexion sur certains objets (miroirs)
- Module de gestion des lumières avec ajout, suppression et paramètres (position, couleur, rayon, ombre, intensité) pour chacune
- La possibilité d'ouvrir différents objets ou scènes et de les charger à l'écran (certains seront fournis avec le programme mais l'utilisateur pourra importer les siens)
- Possibilité de contrôler la caméra (configuration des touches)

Chaque option de l'interface permettra d'ouvrir une infobulle expliquant les effets qui seront produits. Un panneau sera disponible avec une aide et des explications complètes.

Enfin le programme sera aussi capable d'afficher certains indicateurs :

- FPS
- nombre de millisecondes par étapes
- erreurs de compilation des shaders.

En cas de problème (crash, erreur de compilation ou de lancement de certaines fonctionnalités...) il sera possible de récupérer un journal d'erreur simple.

d- Tests

Les tests unitaires concernant les effets du moteur de rendu seront fait « à la main ». Il est en effet difficile de tester leur bon fonctionnement autrement que par l'œil humain.

La solution pourrait être d'utiliser une référence en faisant un rendu photo-réaliste (grâce à OptiX par exemple) puis comparer les images et l'erreur quadratique entre la référence et notre image. Cette solution est toutefois longue à réaliser et elle requiert des outils supplémentaires qu'il faudrait apprendre à maîtriser.

De même l'interface graphique ne pourra pas être testée avec un autre programme. Nous essayerons donc de simuler les comportements d'un utilisateur dans les moindres détails.

III. Code source actuel

Nous allons maintenant présenter une partie du code source.

a- Interface graphique

Il s'agit ici d'une partie du fichier « `mainwindow.cpp` » qui constitue la classe *mainwindow* gérant la fenêtre principale de notre interface graphique :

```
#include "mainwindow.h"

//Constructeur de Mainwindow à l'aide de GroupBox et de GridLayout. Code
directement inspiré de la documentation Qt
MainWindow::MainWindow()
    : QWidget()
{
    QGridLayout *mainGrid = new QGridLayout;    //GridLayout principale de la
fenetre principale. Permet d'organiser Les GroupBox.
    mainGrid->addWidget(createMenuGroupBox(), 0, 0);
    mainGrid->addWidget(createLumiereGroupBox(), 1, 0);
    mainGrid->addWidget(createObjetGroupBox(), 2, 0);
    mainGrid->addWidget(createBasGroupBox(), 3, 0);
    setLayout(mainGrid);

    setWindowTitle(tr("EnLight'INT")); //Titre de la fenetre.
}
```

Nous avons décidé de structurer notre fenêtre grâce au widget *QGroupBox*, qui permet d'obtenir facilement une boîte avec cadre et titre et qui peut contenir d'autres widgets à l'intérieur. Un appel au constructeur permet ainsi de créer directement la fenêtre.

```

//Créer GroupBox du menu de base file/control/constantes
QGroupBox *Mainwindow::createMenuGroupBox()
{
    //Création GroupBox
    QGroupBox *g_main = new QGroupBox;
    //Création PushButtons
    QPushButton *p_file = new QPushButton(tr("File"));
    QPushButton *p_control = new QPushButton(tr("Control"));
    QPushButton *p_const = new QPushButton(tr("Constant"));
    //Organisation via GridLayout
    QGridLayout *lumiereGrid = new QGridLayout(g_main);
    lumiereGrid->addWidget(p_file, 0, 0);
    lumiereGrid->addWidget(p_control, 0, 1);
    lumiereGrid->addWidget(p_const, 0, 2);

    return g_main; //return GroupBox créée
}

```

Chaque méthode qui permet de créer une *groupBox* fonctionne de la même manière : on crée à l'intérieur les widgets dont on a besoin (ici des boutons) que l'on organise grâce à *QGridLayout* (une classe Qt permettant de disposer les différents éléments selon une grille).

```

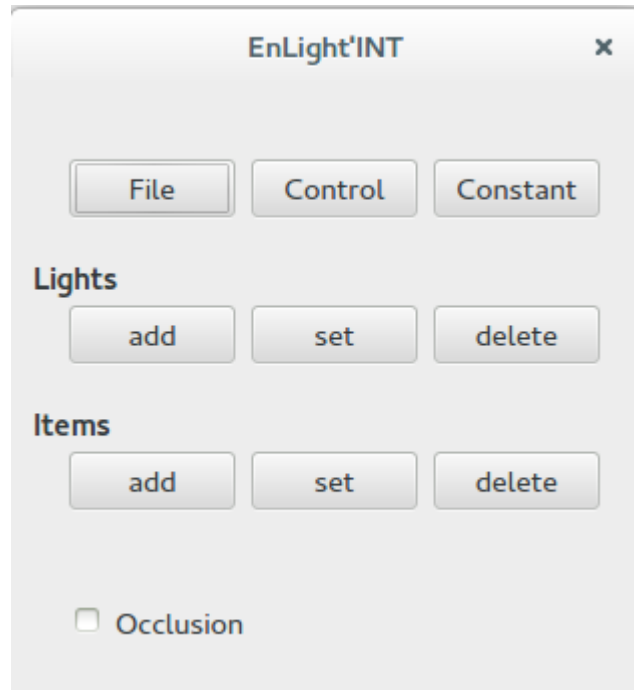
//Créer GroupBox des options annexes. CheckBox occlusion.
QGroupBox *Mainwindow::createBasGroupBox()
{
    //Création GroupBox
    QGroupBox *g_bas = new QGroupBox;
    //Création PushButtons
    QCheckBox *c_occlusion = new QCheckBox(tr("Occlusion"));
    //Organisation via GridLayout
    QGridLayout *basGrid = new QGridLayout(g_bas);
    basGrid->addWidget(c_occlusion, 0, 0);

    return g_bas; //return GroupeBox créée
}

```

Enfin nous créons ici une *box* avec une case à cocher.

Le résultat final est le suivant :



b- Moteur de rendu

Nous allons présenter ensuite une partie du *fragment shader*. Un *shader* est un petit programme s'exécutant sur la carte graphique, le *fragment shaders* s'occupe de calculer la couleur finale de chaque pixel rendu à l'écran (ce programme s'exécute ici pour chaque pixel de l'écran et pour chaque lumière).

```
#version 440 core

// Uniform
#define CONTEXT 0
#define FRUSTRUM 1

// Shader Storage
#define COMMAND 0
#define CLIP 1
#define WORLD 2
#define AABB 3
#define MATERIAL 4
#define PROJECT_LIGHT 5
#define POINT_LIGHT 6
#define WORLD_POINT_LIGHT 7
```

```
flat in int ID; //!< ID of the Light
in vec2 texCoord; //!< Position on the screen
```

Ici le *shader* récupère en entrée (on utilise le mot-clé *in*) les positions écran de la projection de la lumière sur l'écran

```
// Differents textures
layout(binding = 0) uniform sampler2D samplerPosition;
layout(binding = 1) uniform sampler2D samplerNormal;
layout(binding = 2) uniform sampler2D samplerShininessAlbedo;

// Own the shadow maps
layout(binding = 3) uniform samplerCubeArray samplerPointLightShadowMaps;

/**
 * @brief Describe some informations to give at Shader for PointLight Lighting
 */
struct PointLight
{
    vec4 positionRadius; //!< .xyz = position, w = radius
    vec4 color; //!< .rgb = color, a = intensity
    ivec4 shadowInformation; //!< .x = index of Cube Shadow Map
};

layout(binding = FRUSTRUM, shared) uniform FrustrumBuffer
{
    mat4 frustrumMatrix; //!< Is the projectionMatrix product viewMatrix
    vec4 posCamera; //!< .xyz = posCamera or PosLight for shadowMaps for example
    vec4 planesFrustrum[6];
    uvec4 numberMeshesPointLights; //!< NumberMeshed : .x, .y = NumberPointLights
};

// Buffer which own Point Lights
layout(binding = POINT_LIGHT) buffer PointLightBuffer
{
    PointLight pointLights[];
};

out vec3 color;
```

On récupère en sortie (mot-clé *out*) la couleur du pixel correspondant à la lumière ID sur l'écran.

```

void main(void)
{
    // Info at this texel
    vec3 position = texture(samplerPosition, texCoord).xyz;
    vec3 normal = texture(samplerNormal, texCoord).xyz;
    //float shininess = texture(samplerShininessAlbedo, texCoord).x;

    // Info light
    vec4 positionLightRadius = pointLights[ID].positionRadius;
    vec3 colorLight = pointLights[ID].color.rgb;
    int shadowMap = pointLights[ID].shadowInformation.x;
    float radiusSquare = positionLightRadius.w * positionLightRadius.w;

    // Relation between light and the "texel"
    vec3 vertexToLight = positionLightRadius.xyz - position;
    float distanceLightVertex = length(vertexToLight);
    vec3 vertexToLightNormalized = normalize(vertexToLight);

    float attenuation = max(0.0, 1.0 - distanceLightVertex /
positionLightRadius.w);

    // Lambert Cosine Law
    float lambertCoeff = dot(normal, vertexToLightNormalized) * attenuation;

    if(lambertCoeff > 0.0)
    {
        //vec3 dirEye = normalize(posCamera.xyz - position);

        //float shine = max(0.0, pow(dot(dirEye, reflect(-vertexToLightNormalized,
normal)), shininess));
        color = colorLight.rgb * (lambertCoeff);

        // If we use shadow mapping
        if(shadowMap > -1)
        {
            // Simple Exponential Shadow Mapping
            float z = texture(samplerPointLightShadowMaps, vec4(-vertexToLight,
shadowMap)).x;
            float d = distanceLightVertex / positionLightRadius.w;

            color *= min(exp(-positionLightRadius.w * 0.0625 * (d - z)), 1.0);
        }
    }

    else
        color = vec3(0.0);
}

```

Enfin dans la fonction principale, on récupère les informations contenues dans les textures (position, normale, couleur, éclairage du point considéré) et on calcule la couleur finale en utilisant l'équation de rendu de Kajiya (voir *figure 1*). La couleur sera ajoutée en utilisant le mode "BLEND" d'OpenGL dans le *FrameBuffer* associé à l'éclairage direct.