Projet Informatique – Sections Electricité et Microtechnique

Printemps 2016 : *Fiat Lux !* © R. Boulic

1. Introduction

Le but de ce projet est de réaliser une simulation de la propagation de la lumière lorsqu'elle se propage en ligne droite en se réfléchissant sur les surfaces se trouvant sur son chemin. On cherche à mettre en valeur la dimension temporelle de cette propagation en représentant la lumière par des points se déplaçant à une vitesse constante dans un espace 2D (voir exemples de plusieurs états successifs de la simulation ci-dessus). Avertissement : ce modèle de « bille de lumière » est très incomplet du point de vue de la Physique mais il suffit pour nos objectifs du cours d'informatique. Nous utilisons aussi nos propres constantes *(vitesse, etc…)* pour faciliter la mise en oeuvre et les tests.

Il n'y a aucune dépendance vis-à-vis du précédent projet excepté la mise en œuvre des grands principes (*abstraction, ré-utilisation*), les conventions de présentation du code et les connaissances accumulées jusqu'à maintenant dans ce cours. Le but du projet est surtout de se familiariser avec un autre grand principe, celui de *séparation des fonctionalités* (*separation of concerns*) qui devient nécessaire pour structurer un projet important en *modules* indépendants. Nous mettrons l'accent sur le lien entre *module* et *structure de données*, et sur la robustesse des modules aux erreurs (notion de type *opaque* et de *contrat*). Par ailleurs *l'ordre de complexité* des algorithmes sera testé avec des fichiers tests plus exigeants que les autres. L'évaluation du travail portera essentiellement sur la résolution du problème du point de vue informatique : structuration des données, modularité du programme, robustesse et ré-utilisabilité des modules, stratégie de test, ordre de complexité calcul/mémoire des algorithmes mis en oeuvre, compromis performance/occupation mémoire.
Le projet étant réalisé par groupes de deux personnes, il comporte un oral final individuel noté pour lequel nous demandons à chaque membre du groupe de comprendre le fonctionnement de l'ensemble du projet. Une performance faible à l'oral peut conduire à un second oral approfondi et une possible baisse des notes des 3 rendus.

La suite de la donnée utilise un certain nombre de constantes (en MAJUSCULE) disponibles dans les fichiers **constantes.h** et **tolerance.h**. Ces fichiers sont fournis en Annexe A (px).

2. *Monde 2D* et composants de la simulation

Un **Photon** est représenté par une **position** 2D (x,y) et un angle α en rd dans l'intervalle $[-\pi, \pi]$ pour indiquer la direction de déplacement. Les photons n'interagissent pas entre eux même s'ils ont la même position. On a choisi de le représenter par un petit cercle noir (fig 1) de façon à mieux le voir mais on considère seulement la position du point pour les calculs de ce projet.

NBPH photons sont produits par un **projecteur** à chaque mise à jour de la simulation ; la figure 1 montre onze groupes de photons successivement produits par le projecteur situé en bas à gauche. Un photon se déplace toujours en ligne droite avec une vitesse **VPHOT**. Sa trajectoire est réfléchie parfaitement lorsqu'elle intersecte un segment **Réflecteur** comme celui visible à droite sur la Fig1. Un photon est détruit si sa trajectoire intersecte un segment de la ligne brisée d'un **Absorbeur** comme celui visible en haut de la Fig1**.**

Nous vous demandons de travailler avec un système de coordonnée ***Monde*** dont **l'origine (0,0) est au centre de l'espace visualisé**, **X est l'axe horizontal, orienté positivement vers la droite**, **et Y est l'axe vertical, orienté positivement vers le haut**. Le domaine de la simulation n'est pas limité dans l'espace.

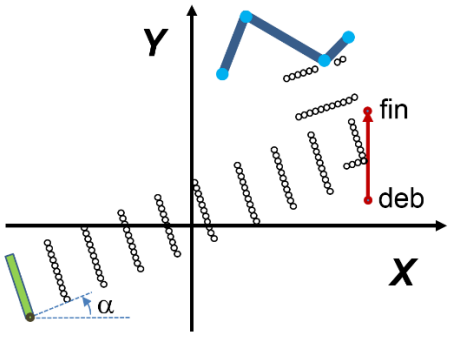

Fig 1 : Système de coordonnées ***Monde*** et exemple d'entités manipulées : **Projecteur** (rectangle vert à gauche), **photons** (petits cercles noir), **réflecteur** (segment rouge), **absorbeur** (ligne brisée bleue).

Projecteur: segment rectiligne défini par une position (x,y) et un angle α en rd dans l'intervalle $[-\pi, \pi]$ qui indique la direction de déplacement des futurs photons. Par construction le segment fait donc un angle de $\alpha+\pi/2$ avec l'axe X (Fig 2). A chaque mise à jour de la simulation, il crée **NBPH** photons séparés par une distance de **EPSIL_PROJ** comme sur la figure 2. Les projecteurs peuvent s'intersecter ou se superposer.

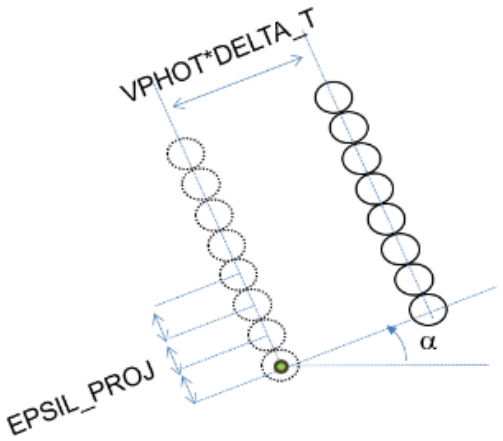

Fig 2 : Paramètres d'un Projecteur. Les pointillés indiquent la position initiale des nouveaux photons avant leur propagation. S'il n'y a aucun obstacle, ils parcourent tous une distance de VPHOT*DELTA_T en ligne droite lors d'une mise à jour.

Reflecteur : segment rectiligne défini par deux points nommés **deb** et **fin** (Fig 1, 3). Agit comme un miroir des deux cotés du segment. Si le segment de déplacement d'un photon (Fig 3a) intersecte un réflecteur alors la trajectoire est réfléchie (Fig 3b). Il faut prendre en compte le réflecteur le plus proche du début de la trajectoire (Fig 3b). Il faut poursuivre la recherche d'intersection jusqu'à ce qu'il n'y ait plus d'intersection, par exemple à partir de P1 puis ensuite à partir de P2 sur la Fig 3c. Les réflecteurs ne doivent PAS avoir de points communs entre eux.

Absorbeur : ligne brisée définie par une suite de points (au moins deux) comme sur Fig 1 et 4. Cette ligne peut s'auto-intersecter. De même plusieurs absorbeurs peuvent s'intersecter ou se superposer. Un photon est détruit si sa trajectoire traverse l'un des segments de l'absorbeur (Fig 1 & 4).

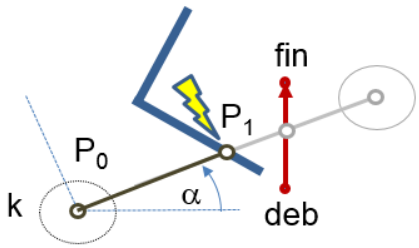

Fig 4 : interception par un absorbeur

Important : **absorbeurs**, **réflecteurs** et **projecteurs** ne peuvent PAS avoir de points communs avec des entités de type différents.

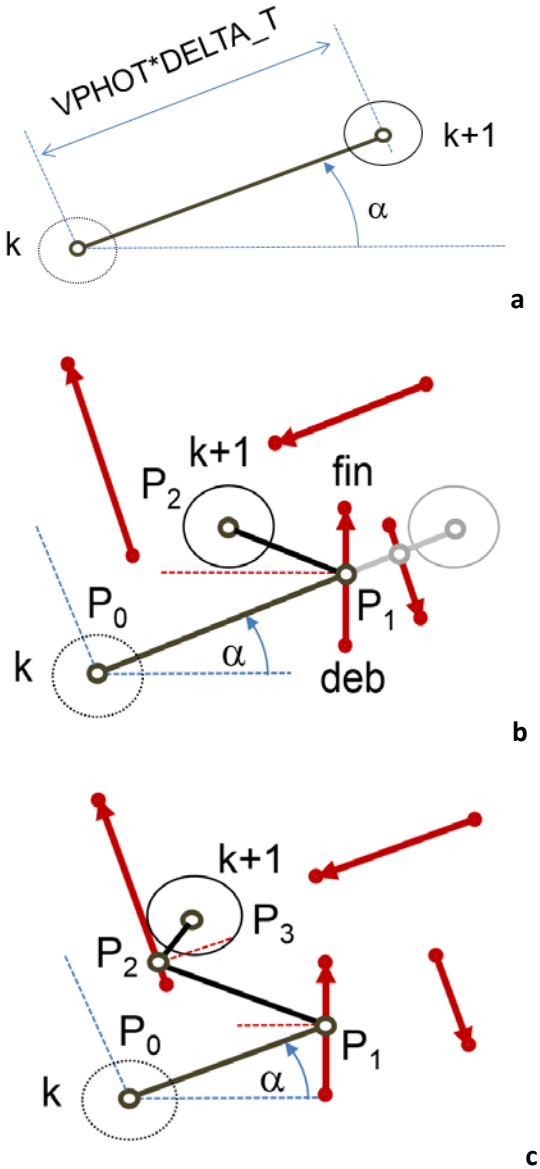

Fig 3 : mouvement d'un photon sur une mise à jour (intervalle de temps DELTA_T). a) Cas sans obstacles, b) une seule réflexion, c) réflexions multiples

3 Actions à réaliser

3.1 Gestion du contexte de la simulation (détails en sections 4 et 6)

Il est possible de réaliser les actions suivantes par l'intermédiaire de l'interface graphique (section 6) lorsque la simulation est en mode Pause :

- **Lecture** ou **écriture** d'un fichier décrivant la simulation (format en section 4)
- **Lancement** de la simulation en continu ou seulement pour un intervalle DELTA_T
- **création, sélection, destruction** d'un projecteur, réflecteur ou absorbeur
- **destruction** de tous les photons qui ne sont pas visibles car situés en dehors du cadre de la fenêtre de visualisation.

3.1.1 Vérification des conditions de non-intersection ou superposition

Les conditions imposées sur les **projecteurs, réflecteurs, absorbeurs** (section 2) doivent être vérifiées après les opérations de **lecture** et de **création** de ces trois sortes d'entités.

Dans ce contexte il est inutile de calculer l'endroit précis où se produit l'intersection ou la superposition. Il faut seulement indiquer qu'on a trouvé ce type d'erreur en appelant la fonction fournie dans le module **error**. Le message indique les types d'éléments fautifs et leur numéro entre **0** et **nb_élément-1**.

Si l'erreur est détectée après la **lecture** d'un fichier, l'ensemble du contexte de la simulation est détruit. On se reportera à la section 6.2.2 si un problème est détecté au moment de la **création** d'un élément.

3.2 Mise à jour de la simulation

La mise à jour de la simulation contient les deux actions suivantes :

- Génération de photons par les projecteurs : dans la position initiale définie par la Fig 2. Ces photons devront aussi effectuer une mise à jour de leur position comme ceux qui existent déjà dans le monde.

- Propagation des photons : elle peut contenir des réflexions multiples et une destruction par un absorbeur sur une portion de la trajectoire. Dans ce contexte il faut savoir où l'intersection se produit sur les réflecteurs pour pouvoir poursuivre la propagation du photon.
  Cas limites:
  - La trajectoire du photon est parallèle avec un absorbeur : le photon est intercepté
  - La trajectoire du photon est parallèle avec un réflecteur = réflexion rasante-> il passe au travers du réflecteur.
  - la position initiale du photon se trouve sur un réflecteur : il est réfléchi.

Tous les paramètres et les calculs de trajectoire doivent être faits en virgule flottante double précision.

3.3 Tâche de bas-niveau

Plusieurs des actions mentionnées ci-dessus ont besoin de résoudre le sous-problème de la détermination de *de la superposition ou de* l'*intersection de deux segments de droite*.

Notations utilisées : soit deux segments s1 et s2. Un vecteur ou un point est noté en **gras**. Un scalaire est noté en *italique*. Chaque segment est construit à partir d'une paire de points (**deb**, **fin**) définissant un vecteur noté **v** qui est normalisé pour obtenir un vecteur unitaire noté **u**. De plus, ayant un vecteur **u**(*x,y*) on obtient un vecteur **n** perpendiculaire à **u** avec **n**(*-y,x*) (**Fig**. 5). On a pour s1:
v1 = (*fin1x – deb1x, fin1y – deb1y*)
nv1 = norme(**v1**) et **u1** = **v1** /*nv1* .

Même chose pour s2.

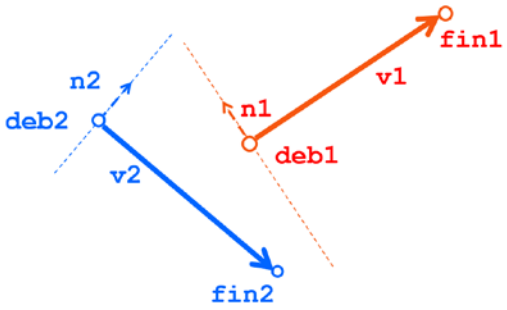

Fig 5 : notations utilisées pour les segments

Nous proposons les méthodes suivantes pour classer les différents cas. On analyse d'abord la superposition à l'aide du produit vectoriel. Si les segments ne sont pas parallèles, on détecte la non-intersection avec le théorème de séparation des convexes. Enfin, si nécessaire, la position de l'intersection est calculée en représentant les segments de manière paramétrique.

3.3.1 Détection du parallélisme et/ou de la superposition

Le produit vectoriel est noté 'X'. Le produit scalaire est noté '.' .
On a: *pv* = **u1** X **u2** , ce qui donne avec **u**(*x,y*) : *pv* = *x1*y2 – y1*x2*
vd1d2 =(*deb2x – deb1x, deb2y – deb1y*) puis *nvd1d2*=norme(**vd1d2**) et **ud1d2** = **vd1d2**/*nvd1d2*

Si |*pv*|≤ EPSIL_PARAL // les 2 segments sont parallèles, éventuellement colinéaires et superposés
pvdeb = **u1** X **ud1d2**
Si |*pvdeb*| ≤ EPSIL_PARAL // les 2 segments sont colinéaires, éventuellement superposés
Déterminer s'il y a aussi superposition à l'aide du signe des produits scalaires suivants (détails **Fig** 6):
psu1v2 = **u1.v2** // directions identiques ou opposées ; produit la norme signée de s2
psdeb = **u1.vd1d2** // produit la distance signée des origines des deux segments
Sinon les 2 segments n'ont aucun point commun
Sinon // les 2 segments ne sont pas parallèles, ils s'intersectent éventuellement

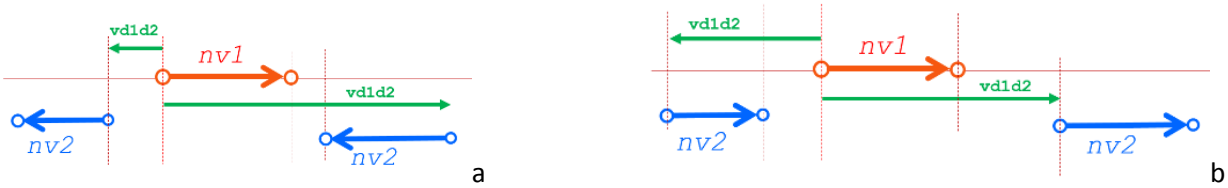

Fig 6 : détermination de la superposition en fonction de la direction des segments et des distances signées.
On prend en compte une marge de EPSIL_CONTACT pour éviter de conserver des segments très proches :
***Il n'y a pas superposition* Si :**
(a) direction opposée : **Si** *psdeb>0 : psdeb + psu1v2 > nv1 +* EPSIL_CONTACT, **sinon**: *psdeb* < -EPSIL_CONTACT
(b) même direction : **Si** *psdeb>0 : psdeb> nv1 +* EPSIL_CONTACT, **sinon**: *psdeb + psu1v2* < -EPSIL_CONTACT

3.3.2 Détection d'une éventuelle intersection pour des segments non-parallèles

vd1f2 =(*fin2x – deb1x, fin2y – deb1y*)
vd2f1 =(*fin1x – deb2x, fin1y – deb2y*)
// vecteurs début-début et début-fin projetés sur les vecteurs n1 et n2
ps1d1d2 = **n1.vd1d2** , *ps1d1f2* = **n1.vd1f2**
ps2d2d1 = -**n2.vd1d2** , *ps2d2f1* = **n2.vd2f1**

Si|ps1d1d2|ou |ps1d1f2| est plus petit que EPSIL_CONTACT ou **Si** ps1d1d2*ps1d1f2 < 0
Si|ps2d2d1|ou |ps2d2f1| est plus petit que EPSIL_CONTACT ou **Si** ps2d2d1*ps2d2f1 < 0
// sortir un booléen VRAI (Fig 7) ou effectuer le calcul du point d'intersection $\mathbf{P_i}$

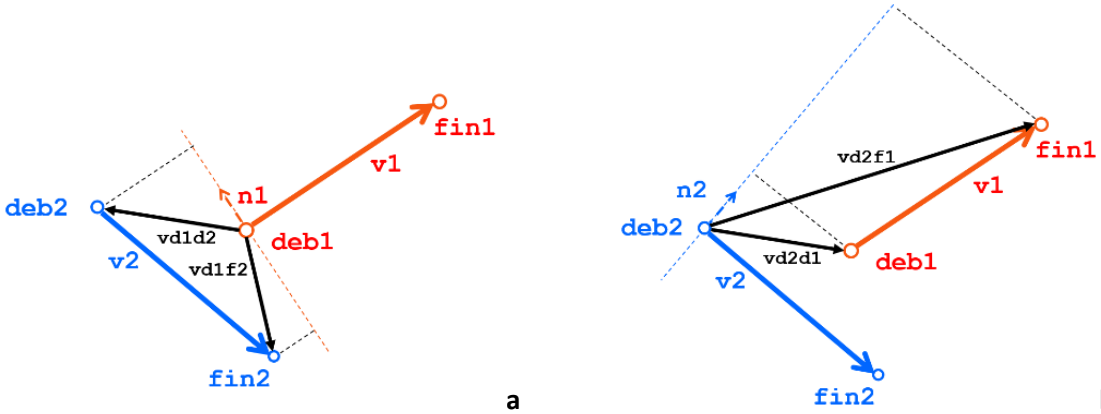

Fig 7 : il y a intersection des deux segments lorsque le signe des produits scalaires est opposé dans les deux cas a et b, ce qui n'est pas vérifié dans cet exemple.

Le point P_i est obtenu en égalisant les équations paramétriques des deux segments. Cette méthode fonctionne dans tous les cas car on suppose qu'on a écarté le cas des segments parallèles. Voici la solution avec l'équation paramétrique de s1 à partir de **deb1** et selon la direction **u1** :

$$P_i = deb1 + [(u2 X vd1d2)/(u2 X u1)]*u1$$

3.3.3 Calcul du vecteur unitaire w de la trajectoire réfléchie d'un photon

Si on appelle **t** le vecteur unitaire incident à un segment de normale **n**, alors **w** est construit comme la somme d'une part, de l'opposé de la composante de **t** selon la normale **n** et d'autre part, de la composante tangentielle de **t** au segment. En simplifiant, cela donne (**Fig** 8) :

$$w = t - 2*(t.n)*n$$

Attention : un segment possède deux normales, une de chaque coté ; il s'agit de choisir la bonne normale pour le calcul de **w** ! Solution : la bonne normale est celle dont le produit scalaire avec **t** est négatif.

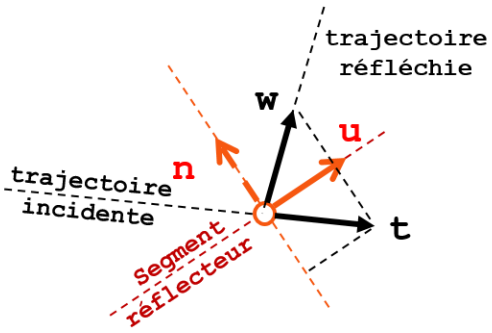

Fig 8 : Calcul de la direction réfléchie de la trajectoire d'un photon

4. Sauvegarde et lecture de fichiers tests : format du fichier

Pour tester les 3 rendus de votre projet nous exécuterons plusieurs scénarios de complexité progressive dont une partie sera publique et disponible dans des fichiers.

Votre programme doit donc être capable de lire de tels fichiers. Il doit aussi pouvoir mémoriser la configuration courante de la simulation. Cela vous permettra de pouvoir reproduire une simulation donnée et de créer vos propres scénarios de tests avec un éditeur de texte.

Le format des fichiers est indiqué ci-contre. Les lignes vides commençant par **\n** ou **\r**, les commentaires commençant par **#** et les espaces avant ou après les données doivent être ignorés ; *il peut y en avoir un nombre différent d'un fichier à un autre*. Le mot-clef **FIN_LISTE** commence en début de ligne et désigne la fin de la liste d'entité en cours de lecture.

Cas de l'absorbeur : MAX_PT points sur une seule ligne
Nombre maximum de caractères par ligne = MAX_LINE

```
# Nom du scenario de test
#
nb de projecteurs
posx  posy  alpha
. . .
FIN_LISTE

nb de reflecteurs
debx deby  finx finy
. . .
FIN_LISTE

nb d'absorbeurs
# un absorbeur = une seule ligne
nbp  x1 y1  x2 y2 …
. . .
FIN_LISTE

nb de photons
posx posy   alpha
. . .
FIN_LISTE
```

5. Description de l'interface utilisateur (GUI)

Comme dans les séries 17-19, nous nous en tiendrons à deux fenêtres graphiques : une pour l'interface graphique et l'autre pour le dessin de la simulation.

La fenêtre d'interface utilisateur doit contenir:

- **Exit** : ferme les fenêtres et fichiers et quitte le programme
- **Open File** : remplace la simulation par le contenu du fichier dont le nom est fourni. Si la vérification détecte un problème, l'ensemble du contexte de la simulation est détruit (sections 2 et 3.1.1). La simulation reste en mode Pause après une lecture.
- **Save File** : mémorise l'état actuel de la simulation dans le fichier dont le nom est fourni

- **Start** : bouton pour commencer/stopper la simulation
- **Step** (lorsque la simulation est stoppée) : calcule seulement un pas

- **Radio-Buttons (lorsque la simulation est en pause):**
  - **Action** : choix entre Sélection ou Création
  - **Entité** : choix entre Projecteur / Réflecteur / Absorbeur

le programme tiendra à jour et affichera les informations suivantes :

- Nb de photons **nbPh**, de projecteurs **nbPr**, de réflecteurs **nbR**, et d'absorbeurs **nbA**

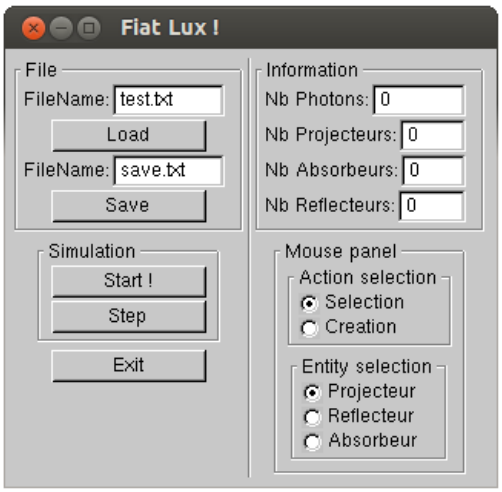

Fig 9 : Interface graphique (GUI)

6. Affichage et interaction dans la fenêtre graphique

6.1 Affichage dans la Fenêtre de visualisation de la simulation :

Cette fenêtre sert à afficher l'évolution de la simulation ou la modification du contexte. La taille initiale de l'espace visualisé est [**-DMAX, DMAX**] selon X et Y. Le cadrage est ajusté seulement lorsqu'on change la taille de la fenêtre ; cet ajustement doit être fait SANS introduire de distorsion dans le dessin.

6.2 Interaction avec la souris et le clavier dans la fenêtre de visualisation :

Les boutons droit et gauche de la souris étant utilisés pour deux familles d'action distinctes, il n'est pas autorisé de les utiliser simultanément.

6.2.1 Action possible à tout moment :

Zoom-in : à tout moment, même en cours de simulation, on doit pouvoir sélectionner un espace rectangulaire avec le **bouton gauche** de la souris. Le début (resp. la fin) de sélection correspond au moment où on *appuie* (resp. *relâche*) le bouton. Un nouveau cadrage sans distorsion doit contenir cet espace rectangulaire. La distance entre les deux points de la diagonale du nouveau cadrage doit être strictement supérieure à EPSIL_CREATION pour valider le nouveau cadrage. Il doit être possible de faire plusieurs actions successives de zoom-in.

Reset zoom : on ré-initialise le cadrage à l'espace [**-DMAX, DMAX**] en pressant sur la touche **r** du clavier.

Destruction de tous les photons en dehors de la fenêtre de visualisation : en appuyant sur la touche **k** du clavier.

6.2.2 Action possible seulement lorsque la simulation est en Pause :

Le radio-button **Action** indique comment utiliser le bouton droit de la souris. L'action par défaut est la Sélection. Le radio-button **Entité** indique le type d'entité concerné par l'**Action** choisie. L'élément par défaut est le Projecteur.

Sélection. Tant que le **bouton droit** de la souris est dans l'*état appuyé* on sélectionne le plus proche élément. On se contente de calculer la *distance entre le pointeur de la souris et les points de début et fin des segments*. L'élément sélectionné est dessiné différemment. Lorsque le bouton droit est *relâché* on conserve le dernier élément sélectionné.

Destruction de l'entité sélectionnée : S'il y en a une, l'entité sélectionnée peut être détruite en appuyant sur la touche **d** du clavier.

Création : à chaque fois que le **bouton droit** est *relaché*, on obtient un point qui sert à créer une entité SAUF si ce point est très proche du point précédent (distance inférieure à EPSIL_CREATION). Dans ce cas l'élément n'est pas créé ; l'action se termine et on passe dans le mode d'action de **Sélection**. Voici comment utiliser les points obtenus selon le type d'entité (au moins deux points sont requis pour une création d'entité):

- **Projecteur** : le premier point est la position du projecteur. Le second sert à définir l'angle alpha du projecteur. On continue en créant un nouveau projecteur avec le troisième et le quatrième point, etc..
- **Réflecteur** : les deux premier points servent à construire les points **deb** et **fin** d'un réflecteur. On continue en créant de nouveaux réflecteurs avec les paires suivantes.
- **Absorbeur** : l'ensemble des points définit un seul absorbeur. On autorise au maximum MAX_PT points par absorbeur ; on repasse automatiquement en mode **Sélection** si ce maximum est atteint.

Comme indiqué en section 3.1.1 il faut vérifier *si chaque nouveau segment* produit une intersection ou une superposition interdite (section 2). Si c'est le cas, on ne crée pas l'élément. Si c'est un absorbeur, il n'est pas créé du tout. Ensuite on passe dans le mode d'action de **Sélection**.

7. Architecture logicielle

7.1 Décomposition en sous-systèmes

L'architecture logicielle de la figure 10 décrit l'organisation minimum du projet en sous-systèmes avec leur responsabilité (Principe de Séparation des Fonctionnalités) :

- **Sous-système de Contrôle :** mis en œuvre avec Le module principal **main ;** c'est le seul module écrit en C++**.** Il est responsable de la gestion du dialogue utilisateur avec l'interface graphique (rassemble toutes les dépendances GLUT-GLUI et la gestion de la projection OPENGL). Si une action de l'utilisateur impose un changement de l'état de la simulation, ce sous-système doit appeler une fonction du **sous-système du modèle** qui est le seul responsable de gérer la simulation (voir point suivant). Le module main.cpp est aussi responsable de traiter les arguments transmis au programme (section 8).

- **Sous-système du Modèle** (rectangle en pointillé dans la figure 10): est responsable de gérer la simulation. Il est mis en œuvre avec plusieurs modules sur plusieurs niveaux d'abstractions selon les Principe d'Abstraction et de ré-utilisation (section 7.2).

- **Sous-Système de Visualisation :** Les dépendances d'affichage avec OPENGL sont concentrées dans le module de bas niveau **graphic** écrit en C (fourni en TP et à compléter). Ce module offre des fonctions pour dessiner des éléments géométriques (ex : segment ou cercle)… Elles seront utilisées dans les modules du modèle pour le dessin des photons, projecteurs, réflecteurs et absorbeurs (Principe de Ré-utilisation).

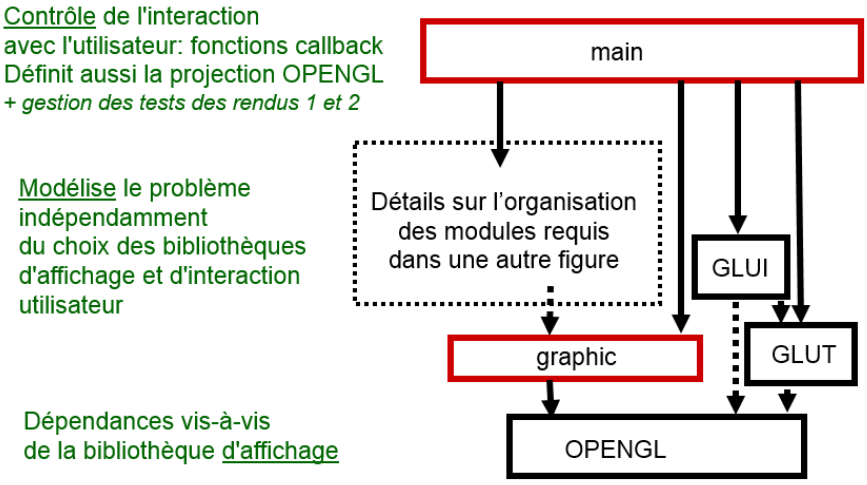

Fig 10 : Architecture logicielle minimale à respecter

7.2 Décomposition du sous-système Modèle en plusieurs modules

Le Modèle doit être organisé en plusieurs modules, tous écrits en C, pour maîtriser la complexité du problème et faciliter sa mise au point. Nous adoptons une approche par **type opaque** où un module exporte seulement des fonctions et/ou des symboles constants ; en particulier l'interface d'un module opaque n'exporte PAS la description des types et n'expose aucune variable globale. La Figure 11 présente l'organisation minimale à adopter en termes d'organisation des modules :

- ***Au plus haut niveau***, le module **modele** gère l'évolution de la simulation et centralise les opérations de dessin, de lecture et d'écriture de fichier ainsi que l'édition des éléments. Ce module doit garantir la cohérence globale du Modèle. C'est pourquoi, en vertu du principe d'abstraction le module **modele** *est le seul module* dont on peut appeler des fonctions en dehors du Modèle (*traduction* : si le sous-système de Contrôle veut modifier l'état de la simulation cela doit se faire par un appel d'une fonction de **modele.h** comme le montre l'unique flèche de dépendance entre le module **main** et le module **modele**).

- ***Niveau(x) intermédiaire(s)***: **projecteur, reflecteur, absorbeur** et **photon.** Chaque module doit réaliser les actions utiles pour chaque entité (création, modification, destruction, lecture, écriture, dessin, etc…)**.**

- Module de bas niveau **utilitaire** (Principe de ré-utilisation): c'est le seul module qui n'est pas opaque. En effet, les modules précédents ayant besoin de travailler avec **des éléments 2D tels que des points, des vecteurs**, **des segments constitués de deux points,** il est demandé de créer un module **utilitaire** de bas niveau mettant à disposition un ou plusieurs **types concrets** pour réaliser des opérations sur ces éléments (ex : opérations vectorielles, calcul de la position d'un point, etc…). IMPORTANT: un tel module de bas niveau doit être général ; il travaille seulement sur les types élémentaires tels que point, vecteur ou segment et il ignore les notions de projecteur, réflecteur, absorbeur, photon utilisés aux niveaux supérieurs du modèle. En particulier, comme le montre la Fig11, il n'y aucune dépendance allant du module **utilitaire** vers les modules projecteur, reflecteur, absorbeur et photon ; les dépendances sont seulement en sens inverse.

- Module de bas niveau **error** : ce module est fourni. Il devra être utilisé lorsqu'il y a détection d'une erreur au moment de la lecture d'un fichier (rendu1) et de la vérification des intersection/superpositions interdites (rendu2).

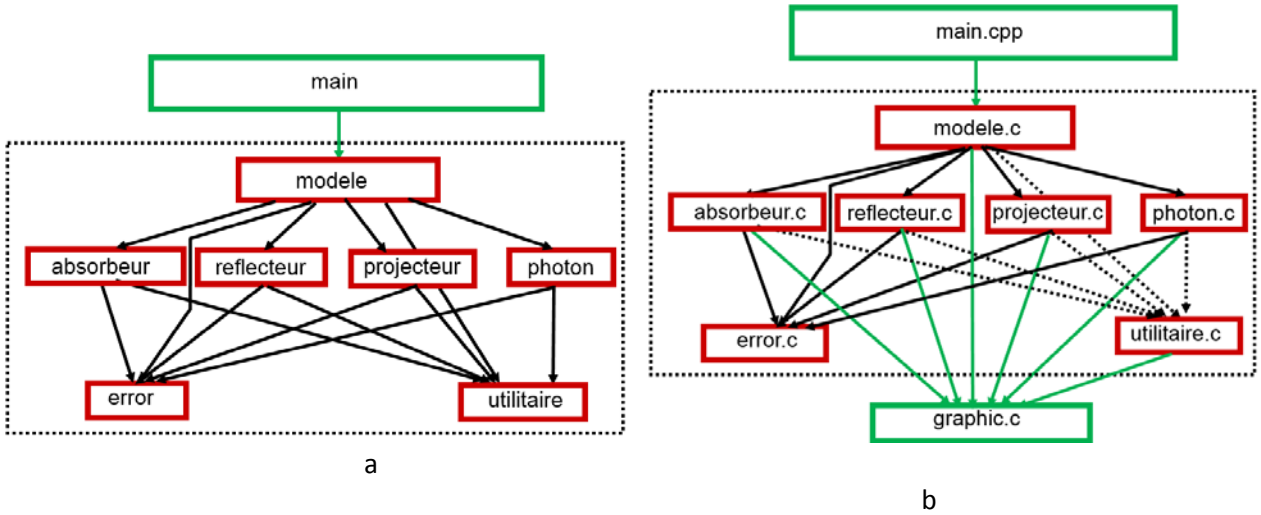

Figure 11 : architecture minimale pour le rendu1 (a), pour le rendu2 (b). le module **utilitaire** est le seul module responsable de types concrets dont la description complète est visible dans **utilitaire.h**. La figure montre les seules dépendances externes autorisées en vert.

Remarque : il est autorisé d'ajouter des dépendances supplémentaires entre les modules **projecteur, reflecteur, absorbeur** et **photon** si vous le jugez nécessaire. Il suffira de justifier ces dépendances dans le rapport du rendu2.

8. Méthode d'évaluation des rendus avec les fichiers de test

8.1 Construction/ré-écriture partielle du programme

Le projet intègre les connaissances du semestre d'automne et les compléments vus pendant les sept premières semaines du semestre de printemps. Pour ces raisons, le programme devra être partiellement ré-écrit entre les rendus car on ne dispose pas de toutes les connaissances nécessaires dès le début de semestre.

Résumé des aspects importants:
Rendu1 : test du format des fichiers (nb d'éléments, proximité des points). PAS de graphique.
Rendu2 : vérification après lecture (3.1.1.), initialisation de la fenêtre graphique et du GUI (section 6), Analyse
Rendu final : simulation, gestion des entités en fonction de l'interaction utilisateur, performances

A chaque rendu, le code source sera évalué selon les conventions de programmation et le respect des types **opaques** (section 7.2). Le module **utilitaire** doit aussi offrir des types concrets qui sont utilisés dans la définition des structures PROJECTEUR, REFLECTEUR, ABSORBEUR et PHOTON.

Détermination du *mode de test* du programme :
L'analyse des arguments transmis sur la ligne de commande (détails en 8.3) permet au module principal **main** de déterminer le ***mode de test*** du programme et le ***nom du fichier*** de test. Ces deux informations doivent être transmises à la fonction de lecture de fichier **`modele_lecture()`** du module **modele**. Les sections suivantes détaillent comment cette fonction doit travailler pour chaque rendu. On précise également les aspects techniques qui dépendent de l'avancement du cours.

8.2 Description des rendus

La répartition des points est détaillée dans le barème en Annexe D.

8.2.1 Rendu1 / test des erreurs de format de fichier simulation:

Ce mode de test est noté **Error**. L'architecture à adopter est donnée par la figure 11a. La fonction **`modele_lecture()`** travaille au niveau d'abstraction supérieur du modèle ; elle doit déléguer la mémorisation des entités et la détection des erreurs au modules **projecteur**, **reflecteur, absorbeur** et **photon**. La liste des erreurs à détecter pour le rendu1 est visible dans **error.h** car il faut appeler les fonctions du module **error.c** pour afficher le message d'erreur standardisé. Il s'agit d'erreurs simples sur le nombre d'éléments (trop ou pas assez de données dans le fichier) et sur la proximité des points qui définissent chaque élément (les deux points constituant un segment doivent être séparés par une distance strictement supérieure à EPSIL_CREATION).

L'erreur doit être détectée dès que l'information est disponible. On quitte la fonction de lecture dès la première erreur trouvée. S'il n'y a pas d'erreur dans un fichier vous devez appeler la fonction **`error_success()`**. Dans ce mode, le programme s'arrête juste après l'exécution de la fonction **`modele_lecture()`**.

Au stade du cours au moment du rendu1, l'allocation dynamique de mémoire n'est pas encore traitée ; c'est pourquoi il est autorisé de mémoriser les entités dans des tableaux de structure de taille constante. Nous garantissons que le nombre d'entités ne dépassera pas la taille de **MAX_RENDU1**. Idem pour le nombre de points d'un absorbeur. Prenons un exemple, dans le module **reflecteur** un tableau de structure REFLECTEUR est défini en dehors de toute fonction avec le mot clef **static** :

```
static REFLECTEUR tab_r[MAX_RENDU1] ;
```

C'est à vous de définir la structure REFLECTEUR. Cette approche devra être revue pour le rendu 2 avec l'allocation dynamique de mémoire. Nous testerons 6 fichiers publics et 6 fichiers supplémentaires.

8.2.2 Rendu2 / test de la vérification des intersections/superposition interdites

Ce mode de test est noté **Verification**. L'architecture à adopter est donnée par les figures 10 et 11b. Les vérifications du fichier (8.2.1) doivent continuer à être faites. Cette fois-ci l'allocation dynamique de mémoire doit être utilisée pour créér le nombre exact d'entités dans les modules **projecteur, reflecteur, absorbeur** et **photon**. Dans ce mode la vérification des intersection/superposition interdites (section 3.1.1 et 5) doit être faite, immédiatement après la lecture du fichier, avec la fonction suivante juste:

```
void modele_verification_rendu2(void)
```

Si un problème est détecté par cette fonction, on doit appeler la fonction dédiée du module **error** pour l'affichage du message d'erreur. Il faut transmettre à la fonction les deux types d'éléments fautifs et leur numéro entre **0** et **nb_élément-1**. On arrête la vérification dès la première erreur trouvée. Dans ce mode, en l'absence d'erreur on quitte le programme après l'appel de la fonction **modele_verification_rendu2**.

Nous testerons 2 fichiers publics et 2 fichiers supplémentaires. Pour réduire les différences entre le programme de démo et le vôtre, nos fichiers de test auront au maximum une erreur et nous accepterons les 2 variantes de messages possibles dus à l'ordre des éléments fautifs dans le message.

8.2.3 Rendu2 / test de l'initialisation du GUI et de la fenêtre de simulation

Ce mode de test est noté **Graphic**. L'architecture à adopter est la même que pour 8.2.2. Les vérifications du fichier (8.2.1 et 8.2.2) doivent continuer à être faites ; la seule différence est que, si une erreur est détectée à la lecture d'un fichier ou ensuite à la vérification, l'ensemble du contexte de la simulation est détruit mais l'exécution se poursuit. Dans tous les cas, le contrôle est passé à GLUI et initialise la fenêtre graphique et l'interface utilisateur comme dans les TP 17-19.

Nous vérifierons manuellement que l'affichage de l'état initial s'effectue correctement dans les 2 fenêtres avec 2 fichiers publics. La lecture de plusieurs fichiers sera aussi faite avec le bouton **Open file** dans un ordre quelconque sans redémarrer le programme. De plus, nous testerons la sauvegarde dans un nouveau fichier et la lecture de ce nouveau fichier pour l'un des fichiers publics et supplémentaires.

8.2.4 Rendu2 / Rapport de la phase d'Analyse (max 1 page)

L'architecture prévue pour le rendu final doit être décrite dans le rapport de la phase d'analyse :

- **Justifier l'architecture du rendu final :** conservez-vous l'architecture de la figure 11b ? Si vous ajoutez des relations supplémentaires entre les modules **projecteur, reflecteur, absorbeur** et **photon**, justifiez chaque nouvelle relation en indiquant la ou les fonctions d'un module qui appelle la ou les fonctions de l'autre module. Justifiez aussi tout module supplémentaire que vous jugez utile.
- **Décrire les structures de données pour le rendu final** (indiquer les changements par rapport au rendu1). Tous les éléments peuvent augmenter/diminuer au cours d'une session; comment cela est-il géré au niveau des structures de données ?
- **Donner la liste des fonctions du module modele qui sont appelées dans le module main** (à quoi servent-elles?). Ces fonctions doivent exister dans le code source du rendu2 à l'état de stub = chaque fonction affiche seulement son nom dans le terminal avec un printf.

8.2.5 Rendu final / interaction

Le mode de test est noté **Final**. Nous testerons manuellement 4 fichiers publics et 2 fichiers supplémentaires. Leur complexité sera progressive pour tester les performances de votre programme pour les fonctionnalités demandées.

8.2.6 Rendu final /Rapport final

Entre 2 (min) et 4 pages maximum (SVP : ne gaspillez pas de papier avec des pages de titre et de table des matières). Le rapport est écrit en français ou en anglais ; une orthographe ou une grammaire défaillante peut induire les correcteurs en erreur. Le rapport contient :

Mise à jour de l'architecture logicielle et de la description de l'implémentation:

- décrivez les compléments ou remises en question depuis le rendu précédent. Documentez la nature des modifications si elles proviennent du rendu public. Fournissez le dessin de l'architecture logicielle finale seulement si elle est différente de celle de la figure 11b.
- précisez votre approche pour la gestion efficace des données. Donnez une estimation du coût calcul et mémoire lors du calcul de la mise à jour d'un pas de la simulation. Indiquez seulement ***le terme dominant*** en fonction des paramètres, par ex : O(nbProj^2 *nbRefl *(nbAbs+nbPhot)); ne PAS remplacer les paramètres par N.
- Illustration avec des images sur un exemple comportant les 4 types d'entités : montrer au moins 2 images de l'évolution de la simulation. Vous pouvez utiliser Alt-PrtSc pour récupérer une image de l'exécution de votre programme.

Méthodologie et conclusion : comment avez-vous organisé votre travail à plusieurs, indiquer la personne responsable de chaque module et comment vous avez organisé le travail au sein du groupe (par quels modules avez-vous commencé, comment les avez-vous testés, comment le feriez-vous maintenant avec le recul, quel était le bug le plus fréquent, pourquoi ? et celui qui vous a posé le plus de problème et comment a-t-il été résolu, ...). Pour conclure fournissez une brève auto-évaluation de votre travail et de l'environnement mis à votre disposition (points forts, points faibles, améliorations possibles, trouvez-vous utile de pouvoir disposer des rendus intermédiaires publics)

8.3 Syntaxe du lancement du programme

Le nom de votre exécutable change selon les rendus (Annexe B). On doit pouvoir lui transmettre deux arguments optionnels au lancement, sur la ligne de commande. La syntaxe est la suivante :

```
sim.x [Error|Verification|Graphic|Final, nom_fichier]
```

Si **aucun argument** n'est transmis, le programme initialise l'interface graphique comme le programme de démonstration. On pourra ensuite utiliser le bouton **Open File** pour initialiser et exécuter une **simulation**.

Argument optionnel: désigne le mode de test du programme. Lorsqu'on indique l'un des cinq mots clef parmi **Error**, **Verification**, **Graphic** et **Final** il faut ajouter un **nom de fichier** de test à la suite (section 4). Ex :

```
sim.x    Error    r1/E01.txt
```

ANNEXE A : constantes utilisées dans le projet

Fichier tolerance.h : constantes générales

```
#define  EPSILON_ZERO          1e-9
```

Fichier constantes.h : constantes spécifiques au projet

```
#include "tolerance.h"
#define  DELTA_T               0.1
#define  VPHOT                 0.2
#define  NBPH                  5
#define  EPSIL_PROJ            0.01
#define  EPSIL_PARAL           0.001
#define  EPSIL_CONTACT         0.001
#define  EPSIL_CREATION        0.02
#define  DMAX                  0.5
#define  MAX_PT                6
#define  MAX_LINE              130
#define  MAX_RENDU1            5
```

Remarque concernant les constantes:
Il est recommandé de tester le projet avec NBPH valant 1 pour faciliter la mise au point du calcul de trajectoire. De même, la trajectoire des photons devrait être toujours la même quelle que soit la valeur de VPHOT. Vous devez donc vérifier si votre projet fonctionne correctement en changeant cette valeur. Essayez par exemple des valeurs entre 0.05 et 2.

V 1.03

ANNEXE B : contenu du fichier fiatlux.zip

Responsable : rhicheek.patra@epfl.ch

4 répertoires: **source, r1, r2, r3**

Le répertoire **source** contient le module error (.c et .h), constantes.h et tolerance.h. Vous devrez y mettre votre code source pour les rendus. Mettez à jour le fichier mysciper.txt en remplaçant les 2 numéros par vos numéros SCIPER (le premier numéro SCIPER est celui de la personne qui télécharge le rendu). SI vous êtes un groupe de trois, mettez les 3 numéros de SCIPER.

Les répertoires **r1, r2, r3** contiennent les fichiers tests publics pour les modes de test des 3 rendus: **Error**, **Verification**, **Graphic** et **Final**. La première lettre du nom des fichiers tests est celle du mode de test. La réponse attendue pour les cas de message d'erreur est en commentaire au début du fichier. Votre fichier Makefile correspondant à chaque rendu doit se trouver dans les répertoires **r1, r2, r3**.

Les 2 fichiers script rendu1.sh et rendu2.sh sont fournis respectivement dans les répertoires **r1** et **r2**. Il s'agit de fichiers texte écrits dans le langage de commande du système d'exploitation qui servent à automatiser l'exécution de votre exécutable sur les fichiers test du rendu1 et du **r**endu2. On peut les exécuter comme on lance un programme. Ils fonctionnent presque de la même manière :

./rendu1.sh a besoin d'avoir une copie de votre fichier Makefile dans **r1**. Le script se charge de le copier dans le répertoire **source** pour recompiler systématiquement le code source qui s'y trouve. Important : votre Makefile doit produire un exécutable qui s'appelle **rendu1.x** . Le script copie ensuite cet exécutable dans r1 pour l'exécuter successivement sur tous les fichiers tests en mode **Error**. Le texte affiché est redirigé vers des fichiers texte « résultat ». Si le script est exécuté plusieurs fois, les anciens fichiers résultats sont remplacés par leur nouvelle version.

Le principe est le même pour le fonctionnement de **./rendu2.sh** à part qu'il utilise un nom d'exécutable **rendu2.x** pour automatiser le test en mode **Verification**. Une copie de votre fichier Makefile doit donc aussi être présente dans **r2**.

Les modes **Graphic** (rendu2), **Final** (rendu3) et sans argument seront testés manuellement après recompilation du code avec la commande make.

Le fichier demo.x est l'unique exécutable de démo (version de la machine virtuelle du Sem2).

Ne modifiez pas la structure de dossiers et fichiers ni leurs noms, c'est important pour l'utilisation des scripts.

ANNEXE C : Forme des rendus

Responsable : rhicheek.patra@epfl.ch

Pour chaque rendu, **UN SEUL membre d'un groupe** (noté **SCIPER1** ci-dessous) doit télécharger un fichier **zip** sur moodle (pas d'email). Toute intervention manuelle de notre part sera pénalisée par 50% des points. Le nom de ce fichier **zip** a la forme :

SCIPER1_ SCIPER2.zip

Le fichier zip doit être organisé comme celui que nous vous fournissons avec les sous-répertoires **source**, **r1**, **r2** et **r3**. Votre code source doit être dans le répertoire **source**. Votre fichier **Makefile** doit être dans le répertoire contenant les fichiers de test et l'éventuelle script du rendu (**r1**, **r2** ou **r3**). Le code sera recompilé et DOIT fonctionner correctement sur la nouvelle Machine Virtuelle du Sem2.

Si un rapport est demandé, il doit aussi être sous forme d'un seul fichier en format pdf dans le fichier zip.

V 1.03

Remarques: Vous devez utiliser la Nouvelle Machine Virtuelle (VM) du **Sem2** car votre projet sera évalué sur cette VM. Les performances de la nouvelle machine virtuelle en *remote access* sont pathétiques pour le graphique. Donc, installez la nouvelle VM en local sur votre ordi et accédez à votre compte myNAS via VPN.

Rappel : Il y a un backup automatique seulement sur votre compte myNAS. Vous êtes responsable de faire votre copie de sauvegarde du projet. Par exemple en vous envoyant un fichier par email.

Gestion du code au sein d'un groupe : les groupes qui utilisent la plateforme **GitHub** doivent limiter l'accès aux seuls membres du groupe. Il est aussi possible de créer un compte sur **`git.epfl.ch`** .

ANNEXE D : Dates et Barème indicatif (max 50 pts)

Les conventions de programmation doivent être respectées. La mise en page du code source ne doit pas produire de passage à la ligne lorsqu'on demande un « ***print preview*** » avec geany (**Print** dans le menu **File**).

Critère	pts	
Rendu intermédiaire 1 : dimanche 20 mars, dernier délais 23h59 – 10 pts		
Adoption de l'architecture Fig11a et description des structures dans les fichiers .c (opaque)	**3**	**8**
Respect des conventions de programmation et clarté du code	**2**	
8.2.1 Error / détection d'erreurs de format dans les fichiers de simulation	**3**	
Rendu intermédiaire 2 : jeudi 21 avril, dernier délais 23h59 – 17 pts		
Adoption de l'architecture Fig11b, type opaque, et allocation dynamique pour l'initialisation	**2**	**19**
Respect des conventions de programmation et clarté du code	**3**	
Rapport de la phase d'analyse (copie imprimée INJ 141 max vendredi 22 avril à midi)	**2**	
8.2.2 Verification / détection de problème d'intersection/superposition	**4**	
8.2.3 Graphic / initialisation des 2 fenêtres avec les données des fichiers	**2**	
Zoom multiple, reset zoom, zoom multiple sans distorsion avec modification de la fenêtre	**2**	
Lancement sans argument et utilisation de Open_file sur 4 fichiers dans ordre quelconque	**3**	
Sauvegarde de l'état courant puis lecture du nouveau fichier	**1**	
Rendu final : dimanche 22 mai, dernier délais 23h59 – 23 pts		
Respect du type opaque dans choix d'architecture finale	**1**	**23**
Respect des conventions de programmation et clarté du code	**1**	
Rapport final (copie imprimée INJ 141 max lundi 23 à midi)	**3**	
Interface : mise à jour des données dans fenêtre GUI selon les actions utilisateurs et la simulation	**1**	
OPENGL : sélection du projecteur ou absorbeur ou reflecteur le plus proche	**1.5**	
OPENGL : création d'un projecteur avec vérifications	**1**	
OPENGL : création d'un reflecteur avec vérifications	**1**	
OPENGL : création d'un absorbeur avec vérifications	**2**	
OPENGL : absence de distorsion et fonctionnement correct après redimensionnement de la fenêtre	**1**	
Simulation : fonctionnement correct des boutons Start/Stop et Step	**1.5**	
Simulation : fonctionnement projecteur correct	**1**	
Simulation : reflection simple correcte	**2**	
Simulation : reflections multiples correctes	**2**	
Simulation : fonctionnement absorbeur correct	**2**	
GLUT : destruction de l'entité sélectionnée courante avec la touche d	**1**	
GLUT : Destruction des photons en dehors du cadre de la projection orthographique avec la touche k	**1**	

Remarque finale : souscrivez au forum PROJET pour toute information complémentaire