RAY OF GOD
Rapport Conception

Yuko Baudet - Valentin Bonnesoeur - Bastien Schatt

07 janvier 2015

Clients : Mathias Paulin - David Vanderhaeghe

Table des matières

1 Introduction

Ce rapport entre dans le cadre du Chef d'oeuvre du Master Image et Multimédia. Notre sujet concerne le développement d'une approche physique réaliste d'un modèle d'ombrage dans les milieux participants, en temps réel, destiné au laboratoire de l'IRIT. Ce troisième rapport sert à définir la conception détaillée du projet d'un point de vue de l'architecture logicielle.

Après avoir replacer la problématique de notre chef d'oeuvre, nous établirons une vue d'ensemble de notre système. Nous définirons les structures de données que nous utiliserons lors du développement de notre chef d'oeuvre. Au travers de plusieurs diagramme de classe, nous donnerons une vision d'ensemble de notre système.Nous définirons aussi les structures de données et la façon dont elles intéragissent ensembles. Ensuite, nous décrirons, les tests garantissant l'intégrité du programme. Enfin, nous mettrons à jour notre planning prévisionnel intial (cf. « rapport Spécifications ») à l'aide de plusieurs diagrammes de Gantt et nous traiterons des possibles problèmes que nous seront possiblement ammenés à rencontrer.

2 Vue d'ensemble détaillée du système

Nous présentons dans cette partie l'organisation en module de notre chef d'oeuvre. Nous pouvons y voir le détails des entrées/sorties, ainsi que les structures de données.

2.1 Diagramme des cas d'utilisation

FIGURE 1 – Diagramme des cas d'utilisation

Le diagramme de cas d'utilisation ci-dessus présente les différentes interactions supplémentaires que pourra avoir un utilisateur et avec le moteur graphique après l'ajout de notre module de rendu de milieu participant. La participation du milieu ou non est aussi un paramètre mis à la disposition de l'utilisateur tout comme le réglage des paramètres de diffusions. Il pourra également visualiser les différentes étapes de l'algorithme via un affichage des textures et des arbres min-max créé à partir des shadow map*s* ainsi que enregistrer les arbres dans un fichier dot.

2.2 Diagramme d'activités

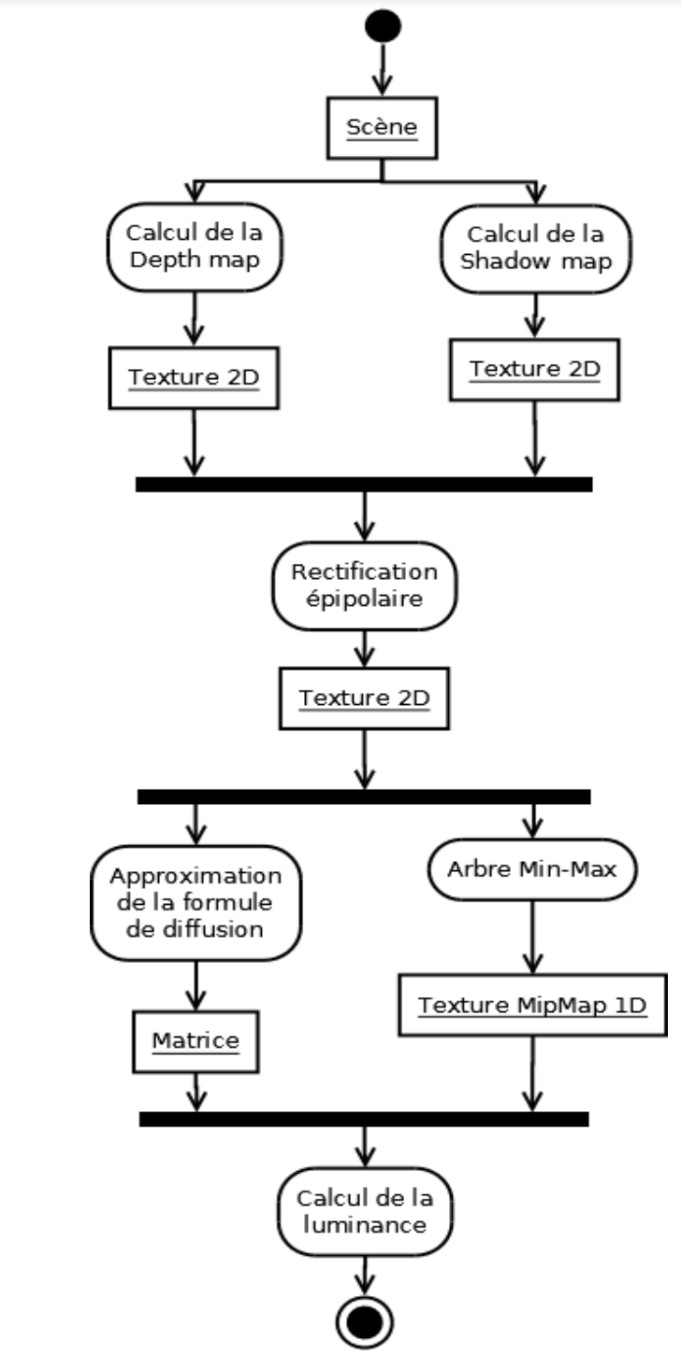

FIGURE 2 – Diagramme d'activités

Dans la figure 2, il est aisé de repérer les étapes qui pourront être traitées en parallèles. En effet, le calcul de la *shadow map* et de la *depth map* sont indépendantes l'une de l'autre. Le calcul de ces deux map fournira alors deux textures, chaque pixel contenant une distance (profondeur).

Par la suite, il sera nécessaire de faire une rectification épipolaire de la texture associée à la *shadow map* afin de pouvoir accéder plus rapidement aux informations d'éclairement (se reporter au rapport « Méthodes et Algorithmes »pour plus de détails).

Une fois la *shadow map* rectifiée, la création des structures en arbres peuvent commencer. En effet, il est nécessaire d'attendre que la *shadow map* soit rectifiée afin de pouvoir rapidement trouver les segments éclairés et les segments se trouvant dans l'ombre. Le programme calculera un arbre binaire min-max pour chaque ligne de la *shadow map* rectifiée. De cette étape en sortira une *mip-map* permettant d'accéder encore plus rapidement aux informations d'éclairements. En même temps, il est possible d'approximer la *scattering intégrale*. Lors de cette étape, la scène sera discretisée dans une matrice 64x64. Pour cela, on utilisera la décomposition en éléments simples.

Et enfin, le calcul final de la *single scattering* est effectué : en partant de l'approximation de la *single scattering* précédente, le programme parcourera les arbres en ajoutant pour chaque arbres, la bonne quantité de lumière. Cette étape est la dernière de notre projet.

3 Structures de données et décomposition en classe

3.1 Structures de données

- Utilisation de trois sortes textures différentes : Texture de profondeur, Texture d'ombre, Texture d'ombre rectifiée
  Ces textures seront stockée dans la mémoire du GPU. L'allocation de cette mémoire et les accès seront gérés par OpenGL. On utilisera l'API pour définir leurs dimensions : quand la fenêtre est redimensionné, leur taille sera adaptée en conséquence. Nous devrons également indiquer à OpenGL le format de la texture à utiliser. Dans notre cas, nous voulons une valeur flottante (de profondeur) par pixel ; le format correspondant sera $GL\_DEPTH\_COMPONENT32F$.

- Arbre Min-Max
  Le but de cet arbre est de faciliter la parallelisation et devra donc être utilisé par un *shader*, executé sur GPU. Il est donc logique de le stocker aussi sur GPU. Pour cela nous allons utiliser une texture 1D de plusieurs niveaux (*mip-map*). Chaque pixel de chaque ligne (sur chaque niveau) contiendra deux valeurs : la valeur minimum et maximum de ses fils. La ligne du niveau le plus bas contiendra un seul élément (de 2 valeurs) : la racine de l'arbre. Chaque ligne contiendra les fils de la ligne précédente jusqu'à arriver à la dernière ligne qui contiendra une ligne compète de la texture de profondeur.
  On utilisera OpenGL pour allouer et remplir la structure *mip-map* sur le GPU. Le format de chaque ligne sera GL_RG32F car on veut stocker des minimums/maximum de la carte d'ombre rectifiée elle même stockée en float 32 bits. Le niveau le plus haut de la *mip-map* devra avoir la même longueur que la fenêtre.

- Matrice contenant l'approximation de la scène
  Une structure matrice sera créé pour contenir le résultat de la décomposition en valeurs simple. Cette matrice contiendra 64x64 éléments de type *float*.

3.2 Décomposition en "classe" au sens large

Dans le figure 3, le diagramme de classe permet d'avoir un rapide aperçut de le décomposition que nous ferons de notre projet. Les classes *Texture*, *Camera*, *Renderer* et *MainWindow* existent déjà dans le moteur fournit par les clients.
Par soucis de visibilité, les classes *Texture* et *Camera* ne sont pas remplie sur le diagramme de classe. De même, nous avons souhaité ne faire apparaître dans la classe *Renderer* uniquement les attributs et méthodes que nous auronq à implémenter. Cependant, plus de détails sont fournis dans le paragraphe suivant.

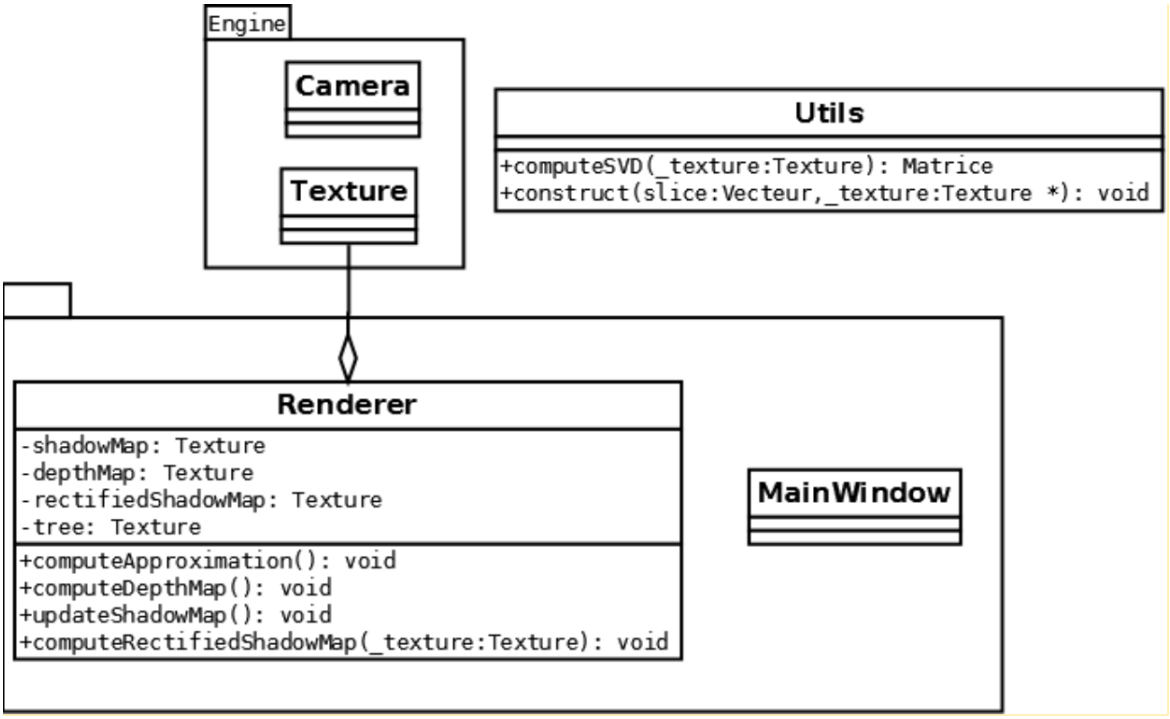

FIGURE 3 – Diagramme de classe

La classe *utils* contient 2 fonctions que nous avons de séparer au maximum du projet car elles ne sont pas spécifiques au projet. La première permet de calculer une *SVD*. Il est intéressant de la séparer du reste car on pourra plus simplement effectuer des tests unitaires et on pourra la réutiliser dans d'autres projets. La seconde permet de remplir un arbre min-max à partir d'un vecteur. N'ayant pas l'utilité d'une structure arbre sur le CPU, on remplit directement une zone mémoire linéaire dans laquelle chaque niveau de l'arbre sera stocké à la suite (le plus grand niveau en premier et la racine en dernier). Quand on devra utiliser cette fonction il nous suffira de passer une zone mémoire du GPU par pointeur (en utilisant OpenGL pour « mapper »cette zone dans la mémoire virtuelle accessible depuis le GPU).
'choses se passe plutôt du côté GPU, dans des *shaders* ce qui ne se prête pas bien à une modélisation avec un diagramme de classe. Nous avons donc opté pour un diagramme de séquence décrivant le déroulement d'une frame.

3.3 Diagramme de séquence

Un diagramme de séquence est présenté dans les deux figures suivantes. Pour plus de visibilité, ce diagramme a été séparé en deux.

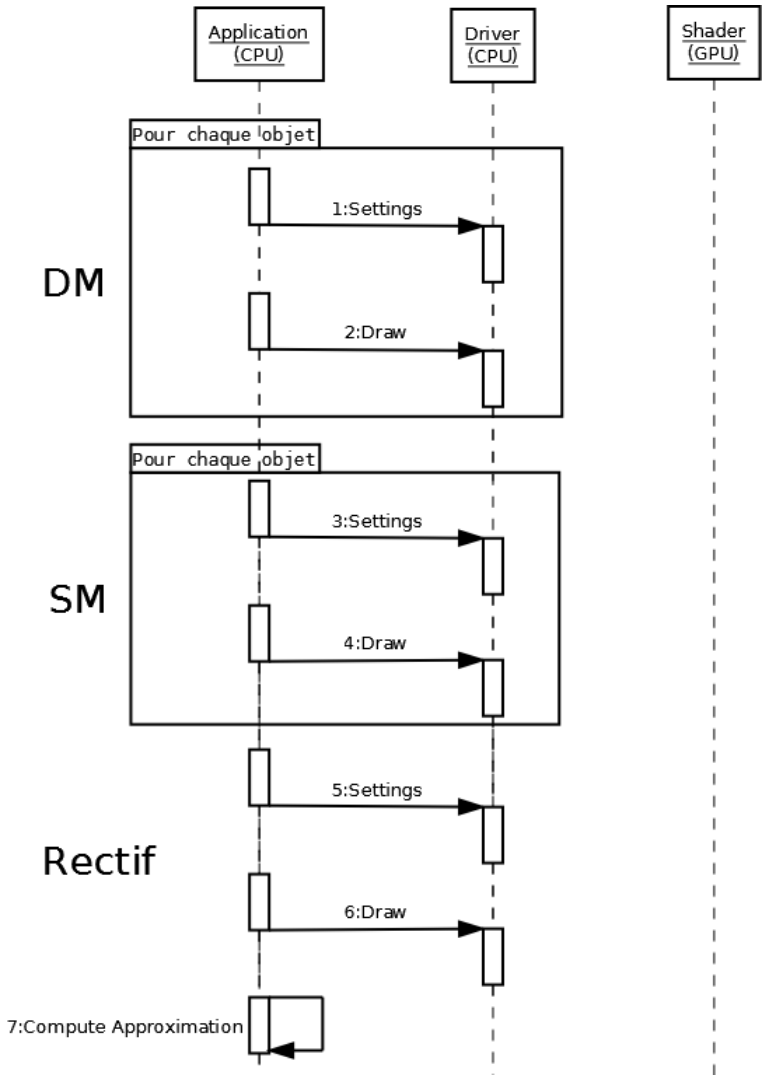

FIGURE 4 – Diagramme de séquence 1

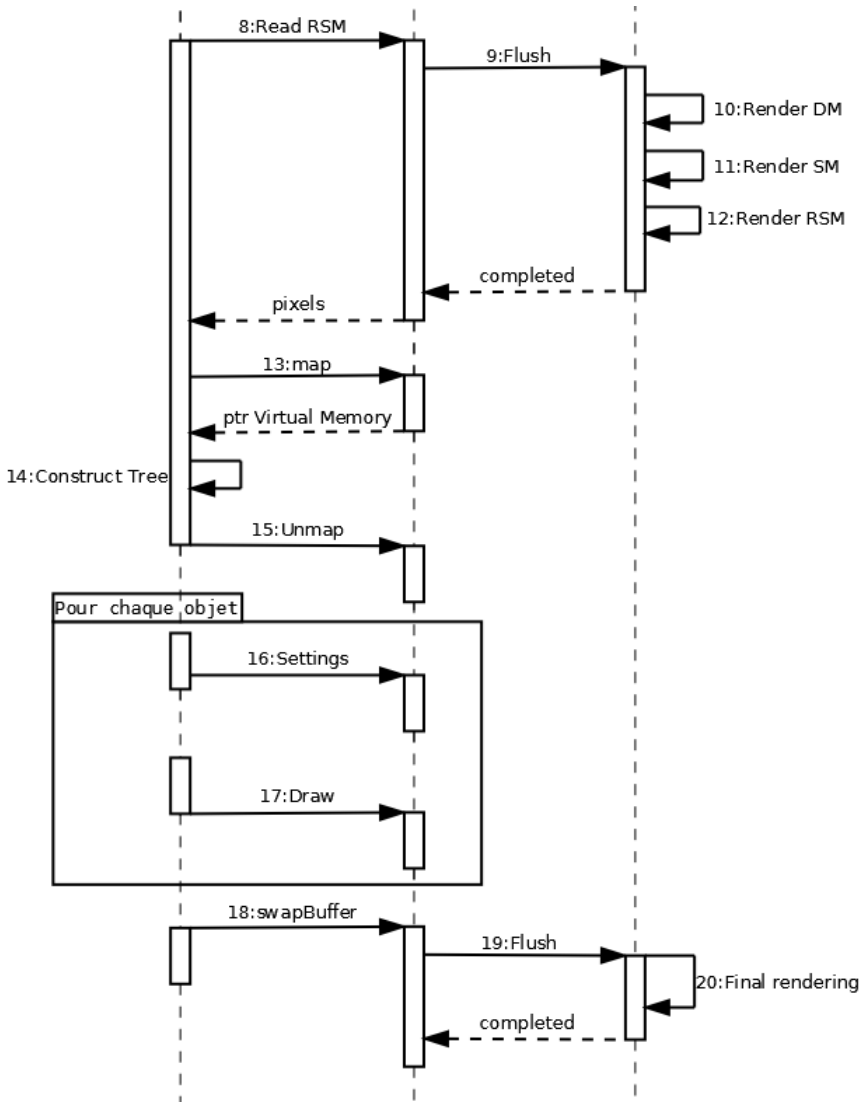

FIGURE 5 – Diagramme de séquence 1

Sur les diagrammes présentés dans les figure 4 et 5, la première étape consiste à dessiner la *Depth Map (DM)*. L'action 1 paramètre donc le dessin de chaque objet : selection du *shader*, envoie de la matrice de transformation, desactivation de l'écriture de la couleur ? Pour chaque objet un ?draw call ? (2) est émis : on demande à OpenGL de dessiner un objet. Le driver place cette commande dans le ?command buffer ? qui sera envoyé d'un seul coups

à la carte graphique.

La deuxième étape (3 et 4), le calcul de la *shadow map* (SM) est à peu près identique sauf que le paramètrage des objets différe légèrement : les matrices de transformations sont différentes puisque le point de vue n'est pas le même (ici c'est la lumière).

La troisième étape (rectification épipolaire) est plus simple (5 et 6) :

- on configure un *shader* spécifique (on bind les textures de profondeurs calculés précédemment)
- on dessine un carré sur la totalité de l'écran

Le *Shader* contiendra le code permettant de faire la rectification épipolaire en fonction de deux textures.

L'approximation grace à la SVD (7) sera effectué intégralement sur CPU.

Pour construire l'arbre min-max on a besoin de lire la *Shadow Map* rectifiée (RSM). On utilise l'API à notre disposition pour lire la texture (8). Le driver OpenGL va alors attendre que les opérations sur cette texture soient terminés avant de faire la lecture. Si necessaire le command buffer sera ?flushé ? (9) : toutes les commandes sont envoyés à la carte graphique pour être effectué le plus rapidement possible. Pour simplifier le diagramme celui-ci est représenté comme si aucune opération n'avait commencé sur le GPU avant la demande de lecture. Une fois toutes les opérations terminées la lecture est effectuée et retournée à l'application.

Une fois disponible (l'opération de lecture étant bloquante) on peut construire un arbre pour chaque ligne de cette texture et le stocker directement dans la mémoire du GPU (en utilisant de la mémoire virtuelle, comme expliqué plus haut, action 1). On construit notre abre min-max en remplissant la mémoire GPU (14) puis on libère la mémoire virtuelle (15).

Pour finir on configure le *shader* final pour faire le rendu de chaque objet de la scène dans le milieu participant (16 et 17).
Le ?swap buffer ? est généré automatiquement par Qt (18). OpenGL va alors vider son ?command buffer ? pour que toutes les opérations se terminent le plus rapidement possible (par exemple le rendu final de la scène).

3.4 Détails des méthodes et attributs

Camera Cette classe permet de créer un objet caméra. Les méthodes présentent dans cette classe permettent d'initialiser la position de la caméra au centre de la fenêtre, d'effectuer une rotation, une translation, ou un zoom ou dezoom.

Texture

Cette classe permet de créer une texture. Une texture est composée de :

- un nom
- un identifiant
- une cible
- un type
- une taille
- le nombre de composante couleurs
- un numéro de couche

Enumération TexType : cette énumération permet de fixe le type de la texture
$TEX\_1D, TEX\_2D, TEX\_3D, TEX\_CUBE, TEX\_RECT$

Les différentes méthodes de cette classe sont :

- Constructeur Texture(std : :string name, GLenum target, TexType type = TEX_2D, GLuint zOffset=0) ;
  name permet de spécifier le nom de la texture
  target specifie la cible à laquelle la texture sera liée. Ce paramètre peut avoir les valeurs : $GL\_TEXTURE\_2DorGL\_TEXTURE\_CUBE\_MAP$
  type définie le type de la texture et par défaut est de type TEX_2D
  zOffset permet de préciser la couche (n'a d'utilité uniquement pour les textures 3D)

- static Texture *loadFromImage(std : :string filename)
  Cette méthode permet de créer une texture à partir d'une image

- void saveToImage(std : :string filename)
  Cette méthode permet d'enregistrer la texture courante dans un fichier image

- void initGL(int byteperpixel, int width, int height, int format, int type, void * data)
  Cette méthode va permettre de générer la texture dans le contexte d'openGL en utilisant notamment les fonctions *glGenTextures* (pour générer une texture), *glBindTexture* (pour lier une texture). C'est également ici que les paramètres de la texture seront ajoutés (glTexParameter).
  byteperpixel : spécifie le nombre de composantes couleurs dans le texture
  width et *height* : informe de la taille de le texture
  format : specifie le format des texels (RGBA, DEPTH, ...)
  data : indique la location mémoire dans laquelle la texture devra chercher ses texels

- void bind(int unit) ;
  Cette méthode permet d'activer la texture avec l'utilisation de la méthode openGL *glActiveTexture*.
  unit indique le numéro de la texture

- void deleteGL()
  Cette méthode permet de détruire une texture en appelant la méthode openGL *glDeleteTextures*.

- Getters
  Un getter sera implémenté par attribut

- Setters
  Aucun setter des attributs ne sera développés car il n'est pas possible de modifier les élements d'une texture une fois crée (comme la taille par exemple).

- Utilisation des mip map :
  void useMipMap(GLenum minFilter, GLenum magFilter) ;
  Cette méthode permet d'activer les *mip maps*.
  minFilter : fonction de minification utilisée pour les *mip maps*
  magFilter : fonction de magnification utilisée pour les *mip maps*

MainWindow

Cette classe permet de créer la fenêtre principale de l'application. C'est ici que sont crées les composants et où sont activées les actions.

Les attributs sont :

- les composants de la fenetres (QMenu, QToolBar, ...)
- la taille de la fenêtre
- le fichier courant

Les méthodes et constructeur sont :

- Constructeur : MainWindow()
  Le constructeur initialise les composants et les attributs de la fenêtre. C'est ici que les méthodes lancant les signals liés aux actions sont lancées.

- Destructeur : MainWindow()
  Suppression des attributs *openGLWidget*, *fileMenu*, *renderMenu*

- void createActions()
  Méthode qui lance les signals et les slots des composants

- Méthodes de créations des composants :
  void createMenus() - void createToolBar() - void createStatusBar()

- void loadFile(const QString & filename)
  Méthode permettant de charger une scène à partir d'un fichier

Les slots sont :

- void open()
  Ce slot permet à l'utilisateur d'ouvrir une scène à partir d'une fichier

- void resetCamera()
  Permet de diriger la caméra vers le centre la fenêtre. Dans ce slot, la caméra courante est supprimée et une nouvelle est créer.

- void reload*shader*s() : les *shader*s sont rechargés

Renderer

Classe permettant le rendu de la scène. C'est dans cette classe que nous implémenteront la plupart de notre code. L'objet Renderer est crée dans la classe OpenGlWidget.

Les méthodes sont :

- Renderer(SceneManager *sceneManager, int width, int height)
  Dans le constructeur sont crées tous les FBO, ainsi que les textures.

- ~Renderer()
  Destructeur dans lequel sont supprimés les FBO, les textures et le quad écran.

- void setViewPort(int width, int height)
  Cette méthode permet d'initialiser les textures en appelant la méthode de la classe texture *initGL*. C'est également dans cette méthode que sont attachée les textures au FBO.

- void renderFilled(const glm : :mat4x4 & modelViewMatrix, const glm : :mat4x4 & projectionMatrix)
  Dans cette méthode les différentes passe sont lancées : la passe de rendu de la lumière ambiante, la passe de rendu pour chaque source de lumière

- void computeApproximation()
  Dans cette méthode, une première approximation de la *scattering intégrale* est effectuée. Le programme lance dans un premier temps la fonction *computeSVD()* de la classe *Utils*. Cela permet de calculer la décomposition en éléments simples de l'image. Ensuite, la méthode finie le calcul par une somme de Riemann en conservant uniquement les termes visibles.

- void computeDepthMap()
  Dans cette méthode sera effectué le calcul de la *depth map*

- void computeShadowMap()
  Cette méthode permet la création des *shadow map*s. Une *shadow map*

est crée pour chaque source de lumière.

4 Tests unitaires

Concernant le calcul de la *depth map* et de la *shadow map*, on ne peut pas utiliser de tests unitaires pour valider leur fonctionnement. Ces deux étapes doivent donc être validées visuellement. Cette validation pourra être effectuée de la façon suivante :

- lancement du programme,
- chargement de la scène de test,
- affichage du menu des propriétés (Ctrl+F),
- sélection de la *depth map* (ou de la *shadow map*) dans la liste des « render target »
- validation visuelle du résultat

Pour ces deux cartes de profondeurs (*depth map* et *shadow map*), la validation visuelle sera très simple : on devra voir que les pixels foncés représentent les éléments proche du point de vue utilisé (caméra ou source lumineuse respectivement) alors que les pixels clairs représentent les éléments lointains.

La seconde grande étape de l'algorithme, la rectification épipolaire devra aussi être validée visuellement. Bien que les étapes permettant sa visualisation soient les même que précédemment, la validation sera moins aisée ! En effet, il est très simple d'interpréter le rendu visuel des cartes de profondeurs, ce qui ne sera pas le cas pour la compréhension visuelle du résultat de la rectification ! Pour cette étape on devra donc valider globalement (on sait à peu près à quoi s'attendre) le résultat mais on ne pourra pas détecter des petites erreurs.

Pour l'arbre min-max on pourra utiliser des tests unitaires. En passant des valeurs simples à l'étape en question on pourra comparer les résultats en sortie avec les valeurs attendues. Par exemple, l'image ci-dessous, tiré de notre article de référence : *Real-Time Volumetric Shadows using 1D Min-Max Mipmaps*, montre une série de données simples (les gamma, en bas de l'image) pour un rayon donné (beta = 5). On pourra faire un test unitaire utilisant ces valeurs en entrée et ainsi comparer le résultat (en sortie) de notre im-

plémentation avec le résultat de l'image (arbre min-max situé dans la partie supérieur de l'image).

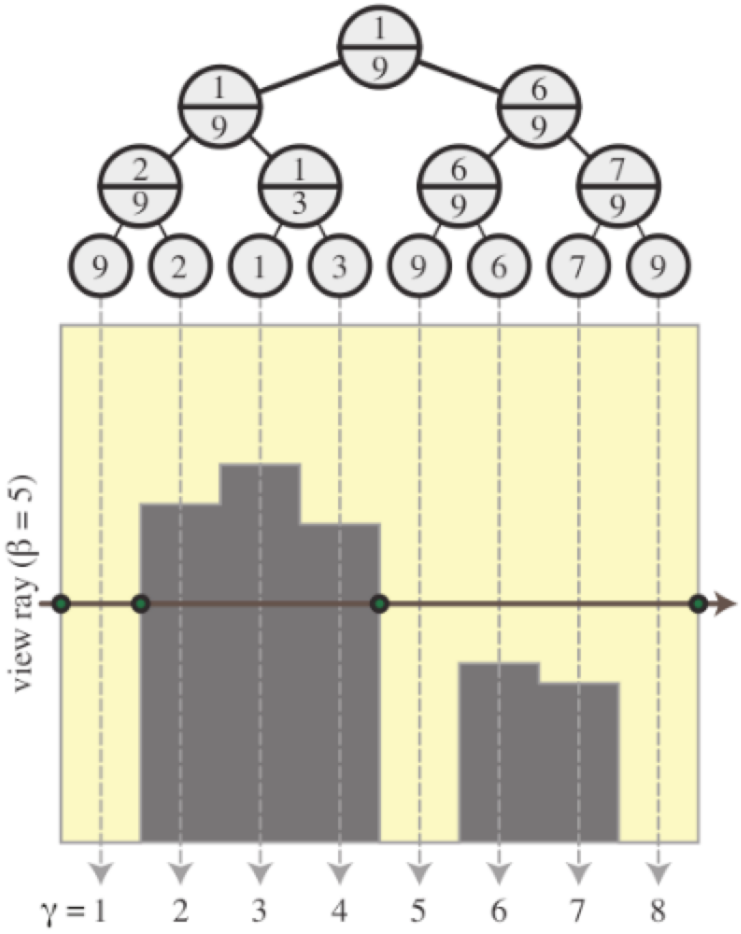

FIGURE 6 – Schéma d'arbre min-max

Pour la décomposition en valeur singulière (SVD), on pourra aussi faire des tests unitaires pour s'assurer que le résultat est correct. Une SVD propose une décomposition d'une matrice M (de taille mxn) en 3 matrices :

- U : matrice unitaire (de taille mxm),
- Σ : matrice diagonale (de taille mxn),
- V* : matrice adjointe à V (de taille nxn).

Or on n'aura en entrée que des matrices de taille 64x64. Donc U, Σ et V* seront toutes de taille 64x64.

Sachant cela, on pourra vérifier que les sorties de notre SVD (U, ? et V*) répondent aux propriétés suivantes (en prenant en compte les imprécisions dus aux calculs flottants) :

- UUT = I
- VVT = I
- M = U Σ V*

Et bien entendu, des tests unitaires comparant le résultat de la SVD avec un résultat connu seront mis en place. Par exemple, en utilisant les valeurs trouvées sur Wikipedia :

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 & 2 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{U} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{\Sigma} = \begin{bmatrix} 4 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & \sqrt{5} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{V^*} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ \sqrt{0.2} & 0 & 0 & 0 & \sqrt{0.8} \\ 0 & 0 & 0 & 1 & 0 \\ -\sqrt{0.8} & 0 & 0 & 0 & \sqrt{0.2} \end{bmatrix}$$

FIGURE 7 – Matrices de décomposition en valeurs propres

Tous les tests ci-dessus devront passés avant de pouvoir implémenter la « scattering equation » qui produira le rendu final de notre scène immergée

dans un milieu participant : si un des tests ne passent pas le résultat ne sera pas correct ! En effet les étapes précédentes servent d'entrées au *shader* qui calcul le rendu final. Cependant, valider les étapes intermédiaires n'est pas suffisant pour dire que le résultat final est correct. L'implémentation de l'équation finale doit bien entendu être fonctionnelle et donc validée. Aucun test unitaire ne peut être implémenté pour cette tape finale. On devra donc la valider visuellement : connaissant notre scène de test (positions des objets et lumières, matériaux des objets, couleurs des lumières ?) et les propriétés du milieu participant (intensité du brouillard par exemple) nous pouvons imaginer le résultat. Le résultat affiché devra correspondre à nos attentes ? mais aussi à celle du client ! En effet cette étape devra impérativement être validée (visuellement) par le client avant la recette.

5 Planning prévisionnel et analyse de risque (mis à jour)

5.1 Les étapes principales

La conception de la tâche 1 de notre chef d'oeuvre, à savoir d'obtenir le rendu d'une scène avec milieu participant (se reporter au rapport « Spécifications »pour plus de détails) se déroulera en deux grandes étapes. Ces étapes ne doivent pas être faites indépendemment les unes des autres.

Chaque sous-tâches des étapes ne sont pas que phases de développement pures. Elles comprennent des tests de validations.

Etape 1

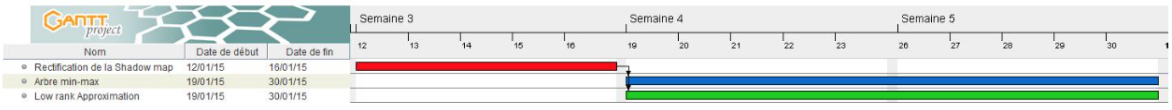

FIGURE 8 – Etape 1

Les calculs de la *Shadow map* et la *Depth map* sont déjà implémentés dans le moteur, nous pouvons dès maintenant appliquer la rectification épipolaire

à la *Shadow map*.

Une fois la shadow map calculée, nous pouvons lui appliquer la rectification épipolaire.

La *single* scattering intégrale : $L(\alpha, \beta) = \int_0^{D(\alpha,\beta)} V(\alpha, \beta, \gamma) I(\beta, \gamma) \mathrm{d}\gamma$ peut se découper en deux parties :

- le terme de la visibilité : $V(\alpha, \beta, \gamma)$
- tout les termes autres que la visibilité : $I(\beta, \gamma)$

Ainsi nous pourrons traiter en parallèle les deux fonctions suivantes :

- Approximation de I grâce à la décomposition en valeurs singulières
- Création d'un arbre min-max pour chaque ligne de la shadow map rectifiée. Les arbres permettent de trouver rapidement quel segment est visible ou dans l'ombre.
- La création des fichiers *dot* peut s'effectuer en parallèle de la création des arbres car la structure des fichiers n'est pas propre à aux arbres min-max, mais fonctionnent pour tous les graphes

Ainsi nous pourrons traiter en parallèle les deux fonctions suivantes :

- Approximation de I grâce à la décomposition en valeurs singulières
- Création d'un arbre min-max pour chaque ligne de la *shadow map* rectifiée. Les arbres permettent de trouver rapidement quel segment est visible ou dans l'ombre.
- La création des fichiers *dot* peut s'effectuer en parallèle de la création des arbres car la structure des fichiers n'est pas propre à aux arbres min-max, mais fonctionnent pour tous les graphes

Etape 2

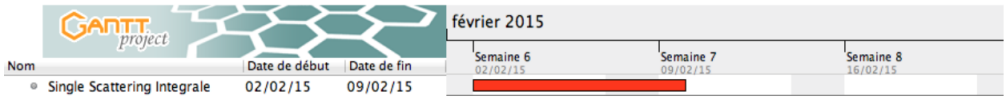

FIGURE 9 – Etape 2

Le calcul final de la luminance ne pourra être effectué une fois que toutes les tâches précédentes auront été implémentées et tester, car son implémentation en dépend fortement. En effet, ce calcul illustrera le résultat final de

tous notre chef d'oeuvre, et avant d'y arriver, il faudra que toutes les fonctionnalités fonctionnent parfaitement. Ainsi, une fois que I est approximé et qu'il est rapide de trouver les segments éclairés grâce aux arbres, nous pouvons réunir les deux afin de calculer la luminance finale approximée grâce à la single *scattering intégrale*.

5.2 Tâches secondaires

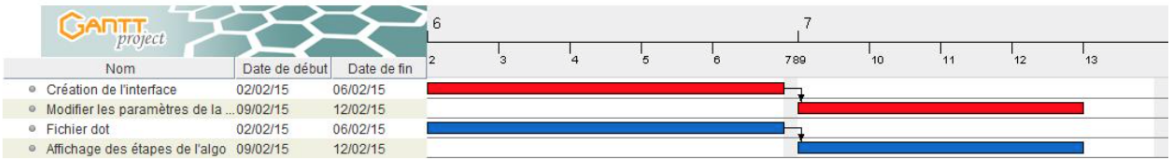

FIGURE 10 – Les tâches secondaires

La création de l'interface ainsi que la création des fichiers *dot* peuvent se faire en parallèle des tâches principales. En effet, cela ne nécessite pas le développement des tâches précédentes. Cependant, les deux autres étapes sont dépendantes des tâches principales et ne pourront être traitées qu'une fois les étapes précédentes validées.

5.3 Les risques

Lors de la conception du projet, le groupe devra faire face à des imprévus. Le but de cette partie est de prévoir les risques les plus probables et de trouver une solution afin de ne pas perdre plus de temps.

- **Mauvaise gestion du planning**
  Conséquence : risque de ne pas pouvoir implémenter toute les fonctionnalités
  Solution :
  - x revoir le planning à temps si le groupe perçoit qu'il ne pourra pas tenir les délais indiqué lors des premières versions du planning
  - x dans le pire des cas, mettre de côté les fonctionnalités les moins importantes (c'est-à-dire les taches secondaires indiqués dans la section précédente)

- **Échec d'une fonctionnalité**
  Conséquence :
  - x perte de temps
  - x ne pas pouvoir développer les autres fonctionnalités qui en dépendent

  Solution :
  - x revoir la méthodologie
  - x prévoir des méthodes alternatives :
  - x si le problème vient des librairies utilisées (ex : Eigen) alors le groupe implémentera la méthode nécessaire (ex : SVD). Si le problème vient d'une méthode codée par le groupe, alors chercher une librairie qui fournit la méthode.

- **Impossibilité de terminer l'étape 1 (calcul des cartes de pronfondeurs) :**
  Conséquence : impossibilité d'achever les autres tâches
  Probabilité d'echec : cela parait peu probable puisque le rendu de la depth map est fourni dans le code du client et nous avons déjà effectué plusieurs implémentations de test concernant le calcul de la shadow map.

- **Impossibilité de terminer l'étape 2 (implémentation des étapes clés de l'algorithme) :**
  Conséquence : impossibilité d'implémenter l'étape 3
  Probabilité d'echec : la rectification épipolaire qu'on a prévu d'implémenter dans cette étape semble assez compliqué à première vue et à l'heure actuelle c'est l'étape qui nous semble la plus difficile !
  Solution :
  - x si le problème concerne la rectification épipolaire, aucune solution de secours n'est possible pour implémenter cet article ! On devrait donc changer d ?objectif et tenter d ?implémenter un autre papier (le premier papier de notre bibliographie par exemple)
  - x si le problème concerne l'implémentation du *low-rank approximation*, utiliser une librarie pour calculer la SVD (par exemple la librairie Eigen)
  - x si le problème concerne la génération de l'arbre minmax, on pourrait tenter de continuer avec une approche plus naïve au risque d'augmenter considérablement le temps de calcul (l'arbre étant une structure permettant d'accélérer l'algorithme) !

- **Impossibilité de terminer l'étape 3 (calcul final de la scène dans un milieu participant) :**
  Conséquence : impossible d'avoir le rendu attendu !
  Probabilité d'echec : une fois que toutes les étapes précédentes ont été implémentées, le calcul final assemble simplement ces étapes. Cependant cette étape est prévu assez tardivement dans le planning, en cas de retard dans les étapes précédentes cette étape finale pourrait ne pas être implémenté à temps.
  Solution : se rabattre rapidement sur une implémentation moins réalise mais plus facile (comme le premier papier de notre bibliographie) si il reste assez de temps
- **Étape non validée par le client :**
  Conséquence :
  - x perte de temps
  - x implémenter de nouveau l'étape (entièrement ou une partie)

  Solution :
  - x communiquer avec le client dès que le groupe a un doute sur un souhait du client
  - x devancer les questionnements qui peuvent se poser lors de l'implémentation pendant les rendez vous avec le client