Programmation lettrée & `Star-Engine` : développement d'un moteur de rendu par Monte-Carlo

|Méso|Star>

29 août 2023

Table des matières

Chapitre 1

Introduction

Ce document décrit un programme de rendu rédigé suivant le paradigme de la programmation lettrée [Knu84]. Dans ce paradigme, l'acte de programmation s'assimile à un acte d'écriture dans lequel les auteurs décrivent les choix de conception et le fonctionnement du programme. Ce faisant, en même temps qu'ils écrivent un ouvrage, les auteurs développent un programme dont la structure suit leurs seuls choix d'écriture. En ce sens, un programme ainsi rédigé s'adresse avant tout à une personne et non à un ordinateur.

La programmation lettrée est une proposition radicale, qui rompt avec la pratique de la programmation telle qu'on l'attend usuellement. Dans un programme conventionnel, l'auteur écrit en premier lieu pour un programmeur qui lui ressemble, ce lecteur cible pouvant se limiter à lui même. L'ensemble du message est alors porté par les seules lignes de code, le programmeur adoptant une architecture, des noms de variables et de fonctions, ou plus généralement un guide de style à destination de ce lecteur ; les commentaires venant compléter de manière ciblée ce que le seul code ne permet pas d'exprimer. Dans un programme lettré, l'auteur porte avant tout un discours qui va au-delà du message porté par les seules lignes de codes, et s'appuie alors sur l'écriture pour en expliciter la teneur. Ces deux paradigmes ne s'affrontent donc pas mais sont employés pour rédiger des programmes à destination de deux publics distincts. Or, au delà de son succès d'estime, la programmation lettrée n'a trouvé que très peu d'écho dans la communauté informatique dont elle est issue, et ce malgré des réalisations unanimement saluées [Han96, PJH16]. On notera que ces réalisations sont l'œuvre d'universitaires où les dimensions "transmission de savoir" et "formations" sont centrales. Et dans ce contexte, la programmation lettrée est un outil particulièrement adapté. Pour autant, l'essentiel des programmes aujourd'hui écrits ont pour premier objectif de traiter des données et d'effectuer des calculs, et non d'être lus et étudiés. Seule une poignée de programmeurs est ciblée en tant que lecteurs, et le seul code source, de part son expressivité serrée, reste adapté à ce lectorat.

Cependant, appréhender un programme comme un écrit donne à l'auteur la possibilité de l'inscrire dans une pensée, et élargit son lectorat. Aujourd'hui pourtant la programmation lettrée reste très peu répandue et aucun commun autour de sa pratique n'a véritablement émergé. Le présent document est lui même un programme lettré dont l'objectif est justement d'éprouver ce paradigme afin d'en ébaucher une pratique. Il se veut suffisamment dense pour donner à voir ce que représente un programme lettré, aussi bien dans l'acte d'écriture que dans celui de programmation. Dans cet objectif, non seulement les codes sources extraits de ce document sont tous distribués sous licence libre, mais ce document est lui même régit par le même type de licence, en l'occurrence la *GNU General Public License v3* [GPL]. Cette licence donne au lecteur le plein accès aux sources du document pour lui permettre de l'étudier, de le modifier et de le redistribuer. Au delà de ses seules sources, le lecteur dispose également du système de génération dudit document. En d'autres termes, nous distribuons l'ensemble du contenu qui matérialise un tel écrit, de ses sources à sa mise en forme. Ce contenu est versionné sur un dépôt `git` et utilise `git-lfs` pour archiver les données binaires. En assumant que ces deux logiciels sont installés sur le système, d'aucun peut dès

lors disposer de l'ensemble des ressources de cet écrit en clonant le dépôt `git` en question à l'aide de la commande suivante :

```
~ $ git clone git@gitlab.com:meso-star/meso-litpt.git
~ $ ls -lR meso-litpt # List the directory content
```

Dans le chapitre 2 nous commençons par définir et travailler les outils qui nous permettent de rédiger ce document. À l'issue de cette étape, le lecteur dispose d'un environnement de travail basé sur l'outil de programmation lettrée `noweb` autorisant l'écriture d'un programme lettré semblable à cet écrit. Nous assumons que cet environnement est déployé sur un système GNU/Linux, c'est à dire un système d'exploitation GNU s'appuyant sur un noyau Linux. Nous prônons ici explicitement l'utilisation d'un système constitué entièrement de logiciels libres par la volonté d'instituer l'utilisateur en tant que *responsable* de son système, c'est à dire en capacité de le maîtriser dans son entier.

Dans le chapitre suivant nous développons une application complète : de son système de compilation, à sa programmation à proprement parler. Ce programme, écrit en `C` [Ker88], est un outil en ligne de commande qui calcule l'image d'une scène 3D à l'aide d'un algorithme Monte-Carlo de suivi de chemins. L'objectif de ce chapitre est double. Tout d'abord il permet d'éprouver l'environnement de travail défini dans le chapitre précédent. Ensuite nous profitons de l'exercice pour rédiger un programme de calcul par Monte-Carlo à l'aide de l'environnement de développement `Star-Engine`. Grâce à la programmation lettrée, ce chapitre documente non seulement comment utiliser le `Star-Engine` pour développer une application de calcul scientifique par Monte-Carlo interagissant avec des géométries 3D, mais aussi comment programmer en `C` un outil en ligne de commande.

Ce document s'adresse donc aux personnes désirant pratiquer la programmation lettrée. Il vise aussi les développeurs familiers du langage `C` qui cherchent à développer des applications à l'aide du `Star-Engine`.

Chapitre 2

Écrire un programme lettré avec noweb

Ce chapitre propose une suite d'outils pour rédiger des programmes lettrés dont le présent document. Cet environnement de développement se base sur `noweb`, un outil de programmation lettrée indépendant des langages de programmation, et qui utilise LaTeX comme langage de description.

Plusieurs raisons nous poussent à utiliser `noweb`. Tout d'abord LaTeX est un puissant langage de description, installé depuis longtemps dans une large communauté très active. De plus, de par son indépendance aux langages de programmation, `noweb` permet de rédiger des codes dans différents langages, et ce dans un seul et même document ; cet écrit contient par exemple des scripts `GNU Bash` et des codes sources en `C`. Au delà de ses seules fonctionnalités affichées, `noweb` s'inscrit avant tout dans une pensée UNIX : il se présente comme un ensemble d'outils, combinés les uns aux autres à l'aide de *pipe* UNIX et de fichiers textes. Chaque outil reste donc simple, s'intéresse à une seule tâche et est facilement extensible comme l'illustre la section 2.3. `noweb` s'insère dès lors parfaitement dans un écosystème UNIX dont il adopte les principes. Et il devient alors tout aussi naturel d'utiliser `noweb` comme un nouvel outil à côté de `awk` ou `gcc`, que de le piloter à l'aide d'un système de génération automatique de fichiers tel que `GNU Make`.

`noweb` présente cependant plusieurs limites et en premier lieu desquelles celle de ne plus être en développement actif. Le problème est ici plus large que `noweb` et touche plus généralement l'ensemble des outils de programmation lettrée. Ce paradigme reste aujourd'hui marginal dans la communauté informatique. Par conséquent, peu d'outils de programmation lettrée sont aujourd'hui encore développés, maintenus ou utilisés. Dans ce contexte, `noweb` reste néanmoins un des outils les plus aboutis et un des plus éprouvés.

Dans la section 2.1 nous commençons par décrire comment installer `noweb` sur un système GNU/Linux. Dans le section 2.2 nous proposons d'utiliser l'éditeur de texte `vim` pour rédiger un programme lettré. Pour ce faire, nous décrivons comment modifier et configurer `vim` pour rendre plus ergonomique l'édition de fichiers `noweb` à travers notamment le support de la coloration syntaxique. La section 2.3 décrit comment modifier le comportement par défaut de `noweb` pour structurer notre document à l'aide des directives LaTeX `\input` et `\include`, tout en conservant le support du référencement croisé des blocs de codes apporté par `noweb`. Enfin la section 2.4 présente le système de génération automatique utilisé pour générer le présent document.

2.1 Installer noweb

Le projet `noweb` n'étant plus en développement actif, plusieurs distributions Linux ne proposent plus de paquet permettant de l'installer. Cette section présente une procédure pour installer `noweb` directement

à partir de ses sources sur un système GNU/Linux sur lequel est notamment installé le shell `GNU Bash`, le compilateur `GCC` et le gestionnaire de version `git`.

Cette procédure se résume à une liste de commandes `Bash`. Bien que cherchant à être le plus générique possible au regard des pré-requis sus-cités, cette proposition n'a aucune prétention à être opérationnelle sur tous les systèmes GNU/Linux. Si des problèmes surviennent, nous invitons le lecteur à se référer à la documentation d'installation distribuée avec les codes sources de `noweb` (`https://raw.githubusercontent.com/nrnrnr/noweb/master/src/INSTALL`).

La procédure d'installation prend la forme suivante :

8a ⟨*install_noweb.bash* 8a⟩≡
```
#!/bin/bash
⟨GNU All-Permissive license 84⟩

set -e
⟨clone noweb repository 8b⟩
⟨setup noweb installation 8c⟩
⟨build noweb 8e⟩
⟨deploy noweb 9b⟩
⟨register noweb install 9c⟩
```

Nous allons donc tout d'abord récupérer le dépôt `git` officiel de `noweb` à l'aide de la commande `git clone`.

8b ⟨*clone noweb repository* 8b⟩≡
```
git clone https://github.com/nrnrnr/noweb
```

À son invocation, cette commande va télécharger les sources du projet `noweb` et les stocker par défaut dans le sous répertoire `noweb` du répertoire courant. Ce dépôt contient de nombreux fichiers et sous répertoires sur lesquels nous ne nous attarderons pas. Le seul sous répertoire qui va nous intéresser ici est le sous répertoire `src` qui contient entre autres les sources des outils `noweb` ainsi que leur documentation. Nous allons donc tout d'abord nous placer dans ce répertoire.

8c ⟨*setup noweb installation* 8c⟩≡
```
cd noweb/src/
```

Nous allons maintenant modifier la date des sources du projet `noweb` afin qu'elles soient considérées comme ayant été mises à jour. Nous forçons ainsi le système de compilation à appliquer le traitement associé auxdites sources comme par exemple générer les différents outils `noweb` et leur documentation. Pour cela nous invoquons simplement la cible `boot` du Makefile courant.

8d ⟨*setup noweb installation* 8c⟩+≡
```
make boot
```

Nous pouvons désormais invoquer la commande `make` pour compiler le projet `noweb` à partir de ses sources.

8e ⟨*build noweb* 8e⟩≡
```
make all
```

Parmi les programmes générés par la commande précédente, certains d'entre eux sont des scripts `Bash` s'appuyant entre autre sur le programme `AWK`. Plusieurs mises en oeuvre de ce programme existent, portant chacune leur propre nom. Par défaut, `noweb` assume que la version disponible sur le système est la version `New AWK` dont la ligne de commande se nomme `nawk`. Or, il est vraisemblable que cette mise en oeuvre ne soit pas installée par défaut sur un système GNU/Linux qui lui préférera sa variante GNU, à savoir `gawk`. En d'autres termes, certains des scripts générés ne fonctionneront pas. Pour palier ce problème, nous allons utiliser le script `awkname` du projet `noweb` qui permet justement de déterminer sur quelle version d'`AWK` les outils `noweb` doivent s'appuyer.

9a ⟨*build noweb* 8e⟩+≡
```
./awkname gawk
```

Nous pouvons désormais installer `noweb` sur le système. Par défaut, `noweb` installe ses binaires, sa documentation et ses bibliothèques dans respectivement les répertoires, `/usr/local/noweb`, `/usr/local/noweb/man` et `/usr/local/noweb/lib`. Les styles LaTeX sont quant à eux déployés dans le répertoire `/usr/local/tex/inputs`. Ces répertoires, pour être créés ou modifiés, nécessitent les droits administrateur sur le système. C'est pourquoi nous utilisons la commande `sudo` pour déployer les fichiers `noweb` dans les répertoires cibles.

9b ⟨*deploy noweb* 9b⟩≡
```
sudo make install
```

Ne reste plus qu'à enregistrer l'installation de `noweb` dans notre shell `Bash`, c'est à dire rendre accessible dans le shell les commandes, pages de manuel et styles LaTeX déployés précédemment. Pour ce faire, nous venons enrichir les variables d'environnement `PATH`, `MANPATH` et `TEXINPUTS` avec les répertoires d'installation de `noweb`. Afin de ne pas avoir à redéfinir ces variables d'environnement dans chaque instance de `Bash`, nous les modifions directement dans le fichier `.bashrc`.

9c ⟨*register noweb install* 9c⟩≡
```
echo "export PATH+=:/usr/local/noweb" >> $HOME/.bashrc
echo "export MANPATH+=:/usrc/local/noweb/man" >> $HOME/.bashrc
echo "export TEXINPUTS+=:/usr/local/tex/inputs" >> $HOME/.bashrc
```

2.2 noweb et vim

Un fichier `noweb` n'étant rien de plus qu'un fichier texte mélangeant LaTeX et code, n'importe quel éditeur peut être utilisé pour rédiger un `noweb`. Se pose cependant la question du support de la coloration syntaxique dans l'éditeur choisi, ce dernier devant identifier les parties LaTeX des parties de codes et utiliser la bonne coloration syntaxique dans chacune des parties.

Pour l'éditeur de texte `vim` cette fonctionnalité peut être ajoutée simplement en utilisant le fichier de syntaxe `noweb.vim` disponible sur le site officiel de `vim` (`https://www.vim.org/scripts/script.php?script_id=3038`). Pour ce faire, il faut d'abord copier ce fichier dans le répertoire `$HOME/.vim/syntax`.

9d ⟨*install the noweb.vim syntax script* 9d⟩≡
```
mkdir -p $HOME/.vim/syntax/
wget https://www.vim.org/scripts/download_script.php?src_id=13268 -O $HOME/.vim/syntax/noweb.vim
```

Nous pouvons alors modifier le fichier `.vimrc` pour configurer l'utilisation de ce script. Trois variables contrôlent son comportement :

- `noweb_language` : détermine le langage qui est utilisé dans les blocs de code ;
- `noweb_backend` : désigne le langage utilisé dans le texte ;
- `noweb_fold_code` : contrôle si les blocs de code doivent être automatiquement “repliés”.

Nous pouvons par exemple ajouter les lignes suivantes au fichier `.vimrc` :

10a ⟨*$HOME/.vimrc* 10a⟩≡
```
let noweb_language="c"
let noweb_backend="tex"
let noweb_fold_code=0
```

Reste alors à configurer `vim` de telle sorte que les fichiers avec une extension donnée (par exemple `*.nw`) soient reconnus comme des fichiers `noweb` devant automatiquement utiliser la coloration syntaxique définie dans le fichier `noweb.vim`. Pour cela, et conformément à la documentation de `vim` (`https://vimhelp.org/vim_faq.txt.html#faq-26.8`), nous définissons un fichier `filetype.vim` comme suit :

10b ⟨*$HOME/.vim/filetype.vim* 10b⟩≡
```
 " my filetype file
if exists("did_load_filetypes")
    finish
endif
augroup filetypedetect
    au! BufRead,BufNewFile *.nw setfiletype noweb
augroup END
```

Ne reste plus alors qu'à s'assurer que la coloration syntaxique est activée dans `vim`

10c ⟨*$HOME/.vimrc* 10a⟩+≡
```
syntax on
```

2.3 Références croisées entre fichiers `noweb`

`noweb` permet d'appréhender l'écriture d'un programme comme un acte d'écriture à part entière avec LaTeX comme langage de description. Pour rédiger son programme, l'auteur peut donc s'appuyer pleinement sur les fonctionnalités de LaTeX ainsi que sur ses bonnes pratiques d'utilisation. Cependant, alors qu'il est courant de fractionner son document LaTeX en plusieurs sous fichiers, `noweb` n'interdit pas à priori cette pratique mais la soumet à plusieurs contraintes ; la principale étant que `noweb` ne supporte pas les références croisées entre les différents fichiers. Un contournement consiste à soumettre plusieurs fichiers `noweb` aux commandes `notangle` et `noweave`, qui extraient respectivement le code source et le LaTeX des fichiers `noweb`. Cette fonctionnalité autorise les références croisées entre les différents fichiers en argument puisque ces derniers sont alors vus comme *un seul* fichier résultant de leur concaténation. Cependant, là où `notangle` permet d'extraire des morceaux de code spécifiques pour les stocker dans des fichiers propres, `noweave` ne produit en sortie qu'un seul fichier LaTeX ; chacune des parties le composant ne peut dès lors plus être incluse séparément à l'aide des commandes LaTeX `\input` ou `\include`.

L'objectif de cette section est de présenter une procédure autorisant et les références croisées entre fichiers `noweb` et leur composition dans les documents LaTeX à l'aide des directives d'inclusion.

2.3.1 Aperçu de `noweave`

Nous cherchons donc à modifier le comportement par défaut de `noweave`. Cette commande n'est en définitive qu'un script `Bash` qui chaîne l'invocation de différents programmes. Pour déterminer ce qu'exécute effectivement la commande `noweave`, on peut utiliser l'option `-v`. Par exemple, l'invocation de la commande `noweave -v -index -delay A.nw B.nw > doc.tex` nous donne les informations suivantes :

10d ⟨*noweave -v -index -delay A.nw B.nw > doc.tex* 10d⟩≡
```
/usr/local/noweb/lib/markup A.nw B.nw \
/usr/local/noweb/lib/finduses | \
/usr/local/noweb/lib/noidx -delay | \
/usr/local/noweb/lib/totex -delay
```

On remarque que `noweave` chaîne ici l'exécution de 4 programmes. Le programme `markup` est d'abord utilisé pour concaténer les fichiers `noweb` en entrée et convertir le résultat dans un format intermédiaire sur lequel les étapes suivantes vont s'appuyer. Le programme `finduses` détermine alors les identifiants utilisés. Puis `noidx` résout les références croisées et l'indexation au sein du document avant de passer la main au programme `totex` qui convertit enfin le résultat en fichier LaTeX.

2.3.2 Vers un nouveau *backend* à `noweave`

Nous allons ici modifier la dernière étape de `noweave` à l'aide du script `Bash` suivant que nous définissons dans le sous répertoire `build` de notre projet :

11a ⟨*build/split_scope.bash* 11a⟩≡
```
#!/bin/bash
⟨GNU All-Permissive license 84⟩

set -e
⟨set the working directory of split_scope.bash 11b⟩
⟨find the noweb lib directory 12c⟩
⟨save the data submitted on standard input in a temporary file 11c⟩
⟨split the weaved data into multiple files 11d⟩
⟨for each resulting file convert it in LaTeX 12a⟩
```

En lieu et place de convertir en un seul fichier LaTeX le résultat des étapes précédentes, nous découpons ce résultat en sous fichiers, où chaque fichier correspond au résultat complet de la commande `noweave` (référencement croisé entre fichiers y compris) mais ici appliquée sur *un* fichier soumis en entrée de la commande `noweave`. Pour ce faire nous nous appuyons sur les données en sortie de la commande `noidx` que nous sauvegardons tout d'abord dans le fichier intermédiaire `weave.all`. Au préalable, nous définissons comme répertoire de travail le répertoire dans lequel notre script s'exécute, et ce afin de s'assurer que le fichier ainsi généré se trouve dans le même répertoire que notre script.

11b ⟨*set the working directory of* `split_scope.bash` 11b⟩≡
```
cd $(dirname $0)
```

11c ⟨*save the data submitted on standard input in a temporary file* 11c⟩≡
```
cat - > weave.all
```
Définit:
`weave.all`, utilisé dans le morceau 11.

Les données issues de `noidx` sont formatées tel un simple fichier texte où chaque ligne contient une seule information. Parmi ces lignes, celles qui nous intéressent sont celles commençant par `@file` puisqu'elles marquent le début des données correspondant à un des fichiers initiaux. Nous utilisons la commande `csplit` pour découper le fichier `weave.all` à chacune de ces lignes et nous sauvons chaque fichier résultat dans un fichier intermédiaire nommé `weave<N>` où `<N>` est un entier compris entre 0 et le nombre de fichiers en sortie.

11d ⟨*split the weaved data into multiple files* 11d⟩≡
```
count=$(cat weave.all | grep -e '^@file ' | wc -l)
if [ $count -le 1 ]; then
  csplit -s -f weave -n 1 weave.all %^@file\ %
else
  csplit -s -f weave -n 1 weave.all %^@file\ % /^@file\ / {$(expr $count - 2)}
fi
```
Définit:
`count`, utilisé dans le morceau 12a.
Utilise `weave.all` 11c.

À noter que Nous utilisons ici la commande `csplit` telle qu'elle est définie dans le standard POSIX.1-2004. Dans cette version, `csplit` demande de connaître à priori le nombre de fichiers à extraire. Pour cela nous comptons simplement le nombre de lignes dans le fichier `weave.all` commençant par `@file`. Nous ajustons alors l'invocation de la commande `csplit` au regard du nombre de fichiers à extraire. Nous invitons le lecteur à se référer à la documentation de `csplit` pour plus d'informations sur cette commande.

Nous pouvons désormais convertir chaque fichier ainsi extrait en fichier LaTeX :

12a ⟨*for each resulting file convert it in LaTeX* 12a⟩≡
```
for((i=0; i<$count; ++i)); do
  ⟨define the filename of the LaTeX to generate 12b⟩
  ⟨convert the file weave$i to LaTeX 12d⟩
done
```
Utilise `count` 11d.

Pour pouvoir être inclus naturellement au sein d'un autre document, le LaTeX généré devra porter le même nom que le fichier `noweb` dont il est issu. En plus du simple mot clef `@file`, les lignes marquant le début d'un des fichiers soumis à `noweave` contiennent également le nom du fichier en question. Nous lisons donc la première ligne de chacun des fichiers pour extraire le nom du fichier `noweb` correspondant, et ainsi pouvoir définir le nom du fichier LaTeX qui lui est associé :

12b ⟨*define the filename of the LaTeX to generate* 12b⟩≡
```
tex=$(head -1 "weave$i" | sed -n 's/@file\ \(.*\).nw/\1.tex/p')
```

Nous sommes alors en mesure de convertir chaque fichier en LaTeX. Pour ce faire nous utilisons le programme `totex` distribué avec `noweb`. Or, ce dernier n'est pas accessible en tant que commande `noweb` ; `noweb` l'assimilant à une "bibliothèque". Nous pouvons cependant retrouver simplement son chemin d'accès en lisant le contenu de la variable `LIB` défini dans le script `noweave`.

12c ⟨*find the noweb lib directory* 12c⟩≡
```
NOLIB=$(cat $(which noweave) | sed -n 's/^LIB=\(.*\)$/\1/1p')
```

Nous appelons enfin `totex` pour convertir les fichiers en LaTeX que nous sauvegardons dans le sous répertoire `tex` de notre projet :

12d ⟨*convert the file weave$i to LaTeX* 12d⟩≡
```
cat "weave$i" | "$NOLIB/totex" -delay | cpif ../tex/$(basename $tex)
```

On notera ici l'utilisation de la commande `cpif` qui permet de n'écrire le fichier LaTeX généré que si ce dernier est différent de celui présent sur disque. Ceci permet au système de compilation de ne pas à avoir à retraiter ce fichier lorsqu'aucun changement n'y a été apporté.

2.3.3 Remplacer le *backend* de `noweave`

Nous pouvons maintenant remplacer la dernière étape de la commande `noweave` par le script `Bash` `split_scope.bash` rédigé dans la section 2.3.2. Dès lors, la commande `noweave` produira un fichier LaTeX par fichier `noweb` qui lui sera soumis en entrée en lieu et place d'un seul fichier LaTeX. Ces fichiers porteront le même nom que les fichiers `noweb` auxquels ils correspondent à l'exception de l'extension qui ne sera plus `.nw` mais `.tex`.

Pour ce faire, `noweave` propose l'option `-backend` qui permet justement de définir quel programme doit être exécuté pour générer la sortie de `noweave` :

12e ⟨*noweave -v -index -delay -backend "bash split_scope.bash" A.nw B.nw* 12e⟩≡
```
/usr/local/noweb/lib/markup A.nw B.nw \
/usr/local/noweb/lib/finduses | \
/usr/local/noweb/lib/noidx -delay | \
bash split_scope.bash -delay
```

En sortie de la commande précédent, nous obtenons donc 2 fichiers, `A.tex` et `B.tex`, entièrement traités par la commande `noweave`, avec support des références croisées entre fichiers, et qui peuvent être naturellement inclus dans n'importe quel document LaTeX.

2.4 Générer le document

Cette partie introduit le système de génération automatique que nous utilisons pour générer les différents fichiers issues des sources de ce document. Et en l'occurence nous utilisons ici `GNU Make` pour piloter l'ensemble du processus : de l'extraction des fichiers LaTeX et des codes sources à l'aide des outils `noweb`, jusqu'à la compilation du programme développé dans le chapitre 3 ou la génération du `pdf` du présent document. Ce Makefile est rédigé de manière conventionnelle et n'est donc pas décrit dans ce programme lettré. La raison principale est que s'il devait être écrit ici, il ne pourrait être utilisé qu'après avoir été extrait des sources du `noweb`. Or, son premier rôle est justement d'extraire ce type de contenu ; soit un cas typique du paradoxe de l'œuf et de la poule. Nous invitons donc le lecteur à se référer directement aux sources du Makefile distribué avec ce document pour - au besoin - étudier, modifier ou étendre son comportement.

Pour générer le présent document, nous invoquons tout simplement la cible `pdf` du Makefile an exécutant la commande suivante à la racine de notre projet :

```
~/meso-litpt $ make pdf
```

À l'issue de cette commande, nous obtenons le document `litpt.pdf` que nous pouvons consulter à l'aide, par exemple, du lecteur `pdf` libre `M`μ`PDF`

```
~/meso-litpt $ mupdf build/litpt.pdf
```

2.4.1 Configurer le système de génération automatique

Pour paramétrer la génération de notre `pdf`, nous utilisons un fichier `config.mk` qui décrit un ensemble de variables contrôlant le comportement de notre système de génération automatique. Ce fichier définit notamment le répertoire dans lequel est écrit ce document ainsi que les options utilisées pour le générer. Le lecteur peut donc se contenter d'éditer ce fichier de configuration pour contrôler le processus de génération, et ce sans avoir à se soucier plus avant du Makefile du projet.

13a ⟨*config.mk* 13a⟩≡
```
⟨licensing formatted with # comments 83b⟩

⟨configure the build directory 13b⟩

⟨configure the generation of the pdf 14⟩
⟨configure the compilation of the program 17b⟩
```

On notera que bien que rédigé de manière lettré, le code précédent ne sert qu'à décrire le fichier de configuration et n'impacte donc en rien son contenu. Dit autrement, tout comme le Makefile qu'il configure, le fichier `config.mk` effectivement utilisé n'est pas issu de cet écrit mais est directement rédigé par le programmeur de manière conventionnelle. Ce fichier peut donc différer du fichier décrit ici qui ne sert en définitive que de support d'explication. Nous invitons donc le lecteur à se référer au fichier `config.mk` distribué avec les sources de cet écrit pour étudier ou modifier la configuration effectivement utilisée par le Makefile du projet.

Nous décidons d'utiliser le sous répertoire `build` pour stocker le `pdf` du document. Nous commençons donc par définir la variable `BUILD_DIR` comme suit :

13b ⟨*configure the build directory* 13b⟩≡
```
BUILD_DIR=build
```

Nous configurons alors l'extraction des codes sources et des fichiers LaTeX opérée par `noweb`. Nous renvoyons ici le lecteur aux pages de manuel des programmes `noweave` et `notangle` pour plus d'informations sur les options de ces programmes.

14 ⟨*configure the generation of the pdf* 14⟩≡
```
NOTANGLE_OPTS = # Nothing
NOWEAVE_OPTS = -index -delay -v
```

Reste alors à configurer la compilation du programme développé dans le chapitre 3. Nous décrivons cette étape dans la section 3.1.2 consacrée à la compilation de notre programme.

Chapitre 3

Développer avec le `Star-Engine` un moteur de rendu par Monte-Carlo

Nous rédigeons dans ce chapitre un programme en ligne de commande qui calcule l'image d'une scène tri-dimensionnelle à l'aide d'un algorithme Monte-Carlo de suivi de chemins. L'algorithme que nous proposons est non biaisé, c'est à dire qu'il résout exactement le modèle physique simulé sans introduire de biais numériques ou statistiques. Le principal objectif de ce programme est d'illustrer l'utilisation du `Star-Engine` dans la mise en œuvre complète d'un code de simulation numérique interagissant avec des données géométriques complexes. Nous présentons donc ici non seulement le code source dudit programme mais également son système de génération automatique.

Le `Star-Engine` est un environnement de développement écrit en `C` standard [Ker88] qui propose un ensemble de bibliothèques sous licence libre [GPL, CeC], facilitant le développement de codes de simulation numérique par Monte-Carlo. Il est en quelque sorte une boîte à outils où chaque bibliothèque répond à une question spécifique ; le programmeur venant utiliser les seules bibliothèques nécessaires à ses développements. Il propose entre autres la génération de suites de nombres aléatoires statistiquement indépendantes en environnement de calcul parallèle, la gestion de données géométriques et volumiques et leur accès via lancer de rayons, ou encore l'échantillonnage de fonctions de diffusion pour une surface ou dans un volume. Dans le présent programme, nous nous concentrons donc sur la mise en œuvre de l'algorithme de rendu en tant que tel et sur le code permettant de le paramétrer ; laissant au `Star-Engine` les problématiques de lancer de rayons dans une scène 3D quelconque, ou la génération de nombres aléatoires.

Le programme que nous développons se veut à la fois complet et simple. Complet dans le sens où le lecteur dispose *in fine* d'un véritable moteur de rendu qui calcule l'image d'une scène dont il peut définir la géométrie, le point de vue et les propriétés physiques. Complet également concernant sa mise en œuvre qui cherche à répondre aux attendus d'un code de simulation numérique à la fois performant et rigoureux. Simple enfin au regard des fonctionnalités proposées et ce afin de centrer le discours au maximum sur cette mise en œuvre et son articulation avec le `Star-Engine`, et non sur le catalogue de ses possibilités.

En entrée, notre programme attend une géométrie décrite dans le format de données OBJ. Dans ce format, nous ne tenons compte que des seuls maillages indexés : les surfaces paramétriques, lignes et autres types de primitives géométriques sont ici simplement ignorées ; de même que l'éventuel fichier de matériaux associé à l'OBJ qui est là aussi laissé de côté. Nous considérons que la scène est composée d'un seul matériau purement diffusif dont la réflectance monochromatique est définie par l'utilisateur pour l'ensemble de la géométrie. Par ailleurs, aucun milieu participant ne vient diffuser les chemins lumineux entre deux surfaces. Enfin, la seule source lumineuse de la scène est la voûte céleste (hémisphère supérieur de la scène) pour laquelle nous utilisons un modèle d'éclairement uniforme. En sortie, notre programme

écrit une image au format *Portable GrayMap format* (PGM) qui peut dès lors être affichée directement par une logiciel de visualisation d'images.

Nous commençons, dans ce chapitre, par décrire comment installer le `Star-Engine` ainsi que la procédure permettant de compiler et d'exécuter le présent programme (section 3.1). Dans le section 3.2, nous rédigeons ensuite son système de compilation avant de mettre en œuvre dans la section 3.3 le système d'analyse des arguments soumis sur la ligne de commande. Nous décrivons alors les interfaces de programmation de la caméra (section 3.4) et de l'image (section 3.5) puis nous initialisons l'application en tant que telle (section 3.6). Dans la section 3.7 nous présentons et mettons en œuvre l'algorithme de rendu avant d'écrire l'image résultante en sortie. Enfin nous rédigeons la fonction principale qui pilote l'exécution de notre programme (section 3.8).

3.1 Compiler et exécuter le programme

L'applicatif rédigé dans ce chapitre est un moteur de rendu écrit en `C` qui, au-delà des dépendances standards propres au langage, utilise les bibliothèques proposées par l'environnement de développement `Star-Engine`. Cette partie décrit la procédure permettant de compiler et d'exécuter ce programme à partir des seules sources dont est issu le présent document. Nous assumons que la machine hôte utilise un système GNU/Linux sur lequel sont installés le compilateur *GNU Compiler Collection* (`GCC`) dans sa version 4.9.2 ou supérieure, le système de génération automatique `GNU Make` ainsi que le système de compilation `CMake`. Nous considérons enfin que le shell utilisé est le `GNU Bash`.

3.1.1 Installer le `Star-Engine`

Avant de compiler le code source décrit dans les chapitres suivants, nous devons installer le `Star-Engine` sur le système hôte. Notre programme s'appuie sur la dernière version du `Star-Engine` qui, à l'heure où ces lignes sont écrites, est la version 0.8. Nous choisissons de l'installer à la racine de notre répertoire de travail, à partir de son archive pré-compilée distribuée sur le site de |Méso|Star> (`https://www.meso-star.com/projects/star-engine/star-engine.html`).

16a ⟨*install the* ***Star-Engine*** 16a⟩≡
```
⟨download the Star-Engine archive 16b⟩
⟨verify the archive integrity 16c⟩
⟨extract the archive 17a⟩
```

16b ⟨*download the* ***Star-Engine*** *archive* 16b⟩≡
```
wget https://www.meso-star.com/projects/star-engine/downloads/Star-Engine-0.8.0-GNU-Linux64.tar.gz \
  -O $HOME/Star-Engine-0.8.0-GNU-Linux64.tar.gz"
```

Nous vérifions alors l'intégrité de l'archive ainsi téléchargée à l'aide de sa signature numérique formatée selon le standard OpenPGP [CDF$^+$]. Pour cela, nous utilisons l'outil *GNU Privacy Guard* (`GPG`) qui propose une mise en œuvre libre de ce standard.

16c ⟨*verify the archive integrity* 16c⟩≡
```
wget https://www.meso-star.com/projects/star-engine/downloads/Star-Engine-0.8.0-GNU-Linux64.tar.gz.sig \
  -O $HOME/Star-Engine-0.8.0-GNU-Linux64.tar.gz.sig"
gpg2 --verify $HOME/Star-Engine-0.8.0-GNU-Linux64.tar.gz.sig
```

Nous assumons ici que nous avons déjà enregistré et certifié la clé du signataire de l'archive dans notre trousseau de clés PGP. Cet à priori reste pour le moins optimiste et il est plus que vraisemblable que l'on ne dispose pas de cette clé. Ceci dit, la procédure complète décrivant l'ajout d'une clé manquante ainsi que sa certification sort du cadre de ce document. Nous renvoyons donc le lecteur vers le bref aperçu de ces étapes proposé sur le site de |Méso|Star> (`https://www.meso-star.com/projects/misc/pgp_signatures.html`) et vers la documentation de *GNU Privacy Guard* pour une description plus complète (`https://www.gnupg.org/gph/en/manual/c235.html`). Bien que cette étape puisse paraître optionnelle,

nous insistons sur le fait que vérifier la signature d'une archive permet non seulement de valider qu'elle n'a pas été corrompue mais aussi d'avérer l'identité de son distributeur. Nous encourageons donc vivement le lecteur à bien vérifier la signature de l'archive qu'il souhaite installer.

Reste alors à décompresser l'archive préalablement vérifiée pour finaliser l'installation. Nous choisissons ici de l'installer à la racine de notre répertoire personnel.

17a ⟨*extract the archive* 17a⟩≡
```
tar -xzvf $HOME/Star-Engine-0.8.0-GNU-Linux64.tar.gz -C $HOME
```

3.1.2 Compiler les codes source

Nous rappelons que le programme que nous développons est rédigé selon le paradigme de la programmation lettrée. Pour générer son exécutable, nous devons donc tout d'abord extraire ses sources du présent document avant de les compiler à proprement parler. Le système de génération automatique de cet écrit (section 2.4) propose la directive `bin` qui automatise cette procédure de compilation, une procédure que l'utilisateur configure à travers le fichier de configuration `config.mk` présenté dans la section 2.4.1. Nous venons ici enrichir ce fichier avec des variables contrôlant la compilation du programme. Nous rappelons que ce fichier de configuration est utilisé comme support d'explication et n'est pas utilisé en l'état par le système de génération automatique ; ce dernier s'appuie sur un fichier de configuration directement rédigé en dur par le programmeur.

17b ⟨*configure the compilation of the program* 17b⟩≡
⟨*configure the* ***Star-Engine*** *install path* 17c⟩
⟨*configure the build type* 17d⟩

Nous commençons par définir le répertoire d'installation du `Star-Engine`. Dans la section précédente (section 3.1.1), nous avons décidé de l'installer à la racine de notre répertoire personnel ; nous définissons donc la variable `STAR_ENGINE_PATH` en conséquence.

17c ⟨*configure the* ***Star-Engine*** *install path* 17c⟩≡
```
STAR_ENGINE_PATH=$(HOME)/Star-Engine-0.8.0-GNU-Linux64
```

Nous configurons alors le type de binaire que nous souhaitons générer. Nous avons ici le choix entre deux types de compilation : `Debug` ou `Release`. En mode `Debug`, le compilateur va, entre autres, associer des données supplémentaires au binaire qu'il génère afin de faciliter le suivi de son exécution, et ainsi aider à l'identification d'erreurs de programmation à l'aide d'un outil de déboggage comme `GNU Debugger`. En mode `Release`, le compilateur va au contraire chercher à générer un binaire optimisé au sens de son temps d'exécution. Le choix du type de compilation dépend donc de l'objectif recherché par le programmeur : débogger son programme ou simplement l'exécuter. Ici nous choisissons de favoriser les performances d'exécution en utilisant le mode `Release`.

17d ⟨*configure the build type* 17d⟩≡
```
BUILD_TYPE=Release
```

Une fois le système de compilation correctement configuré, nous pouvons générer le binaire de notre programme en utilisant la cible `bin` du Makefile associé à notre écrit. Il suffit pour cela d'invoquer la commande suivante à la racine du projet :

```
~/meso-litpt $ make bin
```

3.1.3 Éxecuter le programme

Nous allons désormais exécuter le programme compilé dans la section précédente et dont le code source est décrit dans les section suivantes. Le binaire `pt` en résultant (section 3.2) se situe dans le répertoire de sortie `build` défini par la variable `BUILD_DIR` du fichier de configuration `config.mk` (section 2.4.1). Pour pouvoir l'exécuter, nous devons au préalable enregistrer l'installation du `Star-Engine` dans le shell courant, et ce afin qu'au lancement du programme, celui ci puisse charger en mémoire les bibliothèques dynamiques du `Star-Engine` sur lequel il s'appuie. Nous allons pour cela utiliser le fichier de configuration du `Star-Engine` pour venir enrichir les variables d'environnement du shell courant telles les variables `LD_LIBRARY_PATH` ou `PATH`.

18a ⟨*run_pt.sh* 18a⟩≡
```
⟨register the Star-Engine installation against the current shell 18b⟩
⟨use pt to draw an image 18c⟩
⟨display the rendered image 18d⟩
```

18b ⟨*register the Star-Engine installation against the current shell* 18b⟩≡
```
source $HOME/Star-Engine-0.8.0-GNU-Linux64/etc/star-engine.profile
```

Nous pouvons désormais exécuter notre programme pour calculer une image d'une scène 3D. Dans l'exemple ci-après, nous utilisons un fichier OBJ qui décrit la géométrie de l'*atrium* du palais Sponza située à Dubrovnik en Croatie. Ce fichier est publié sous licence *Creative Commons Non-Commercial 3.0* [CC-] qui n'est pas une licence rigoureusement libre dans le sens où elle interdit toute distribution commerciale. Une copie de ce fichier, compréssée au format `.xz`, est disponible avec les sources de ce document dans le sous-répertoire `etc`. Ce fichier est automatiquement décompressé lors de la compilation des sources du programme. Dans la commande ci-dessous, nous pouvons donc directement utiliser l'OBJ ainsi décompressé. À noter que l'invocation de cette commande lance le rendu de l'image à proprement parler ; un rendu qui, en fonction de la machine sur lequel il est exécuté, peut prendre plusieurs dizaines de minutes.

18c ⟨*use* pt *to draw an image* 18c⟩≡
```
./build/pt -p 8,4,0 -t -72,4,-1 -u 0,1,0 -F 70 \
  -d 800x600 -n 1024 -r 0.8 -i ./etc/sponza.obj -o sponza.pgm
```

Les différentes options passées au programme `pt` sont décrites dans la section 3.3. Un bref descriptif est néanmoins disponible en passant l'option `-h` à la commande `pt`. L'image que nous venons de calculer est écrite dans le fichier `sponza.pgm` que nous pouvons enfin afficher avec un logiciel de visualisation d'images tel que `feh` ou `display`. La figure 3.1 illustre l'image que nous obtenons.

18d ⟨*display the rendered image* 18d⟩≡
```
display sponza.pgm
```

3.2 Système de compilation

Pour contrôler la compilation de notre application nous nous basons sur le système de compilation `CMake` qui permet de décrire le processus de compilation à l'aide de fichiers de configurations. En ce sens, `CMake` est sensiblement identique au système de génération automatique `GNU Make`. Par rapport à ce dernier il propose cependant une syntaxe de plus haut niveau simplifiant l'écriture et la lecture des fichiers de configuration. De plus, et même si ici cette propriété ne nous intéresse pas en premier lieu, `CMake` est multi-plateforme : ses fichiers sont utilisés pour générer des fichiers de configuration *natifs* à une plateforme où un environnement de développement donné. `CMake` se présente donc comme un méta système de compilation à partir duquel plusieurs systèmes de compilation, et donc plusieurs plateformes, peuvent être adressés. Dans notre cas, nous utiliserons `CMake` pour générer un Makefile à destination de `GNU Make`. Nous soulignons que ce fichier est automatiquement généré par la cible `bin` du Makefile principal de notre projet (section 2.4), et nous renvoyons le lecteur vers ledit Makefile pour plus d'information sur cette procédure.

FIGURE 3.1 – Illustration d'une image rendue par le présent programme. La géométrie de la scène réprésente l'*atrium* du palais Sponza situé à Dubrovnik en Croatie.

3.2.1 Fichier de configuration

Notre application est suffisamment simple pour ne nécessiter qu'un seul fichier de configuration dans lequel l'ensemble du processus de compilation est décrit. Ceci étant, indépendamment du nombre de fichiers `CMake` à écrire, nous prenons par habitude de les regrouper dans le sous répertoire `cmake` du projet que nous développons, et ce afin de séparer les sources du système de compilation des sources du programme à compiler. Dans le système `CMake` un fichier de configuration se nomme `CMakeLists.txt`. Nous ajoutons donc dans le sous répertoire `cmake` le fichier suivant :

20a ⟨*cmake/CMakeLists.txt* 20a⟩≡
```
⟨licensing formatted with # comments 83b⟩

⟨configure CMake project 20b⟩
⟨setup project dependencies 21a⟩
⟨configure the target 22b⟩
```

Comme tout fichier source, nous commençons par décrire le ou les détenteurs des droits patrimoniaux du code qu'il contient et la licence qui le régit. Cette section est décrite dans l'annexe A.

Nous configurons alors notre programme vis à vis de `CMake`. Nous fixons tout d'abord la version minimum de `CMake` que nous attendons, ici la version 2.8. Par cette directive, nous disons à `CMake` que nous souhaitons pouvoir utiliser l'ensemble de la syntaxe, des modules et des fonctionnalités de `CMake` définis dans cette version. Si la version de `CMake` installée sur le système est incompatible avec la version souhaitée, une erreur sera notifiée par `CMake` qui ne sera pas alors en mesure de poursuivre son traitement. À noter qu'ici nous restons très conservateurs au regard de la version de `CMake` souhaitée puisqu'à l'heure où ce document est écrit, la dernière version de `CMake` est la version 3.14. Par ce choix, nous élargissons la compatibilité de notre projet aux systèmes plus anciens sans que cela nuise particulièrement à l'écriture de notre fichier de configuration qui ne nécessite pas de version plus récente.

20b ⟨*configure CMake project* 20b⟩≡
```
cmake_minimum_required(VERSION 2.8)
```

Nous définissons ensuite le nom de notre projet `CMake` et le langage de programmation que nous utilisons, et par conséquent quel type de compilateur est nécessaire (ici un compilateur `C`).

20c ⟨*configure CMake project* 20b⟩+≡
```
project(path-tracer)
enable_language(C)
```

3.2.2 Gestion des dépendances

Une application s'appuie généralement sur des bibliothèques externes pour réaliser son traitement. Ces bibliothèques s'apparentent à des briques logicielles qui peuvent être réutilisées d'un applicatif à l'autre. Dans notre cas, nous utilisons plusieurs bibliothèques proposées par le `Star-Engine` pour notamment lancer des rayons dans une scène 3D ou encore générer des nombres aléatoires.

Le système de compilation doit vérifier qu'une version compatible de ces dépendances est bien présente sur le système afin d'assurer que lors de sa compilation, notre programme pourra bien compter sur ces dernières. Pour ce faire, nous nous appuyons sur la notion de paquet `CMake` qui automatise un certain nombre de traitements. Dans le cadre d'une dépendance externe, le paquet correspondant vérifie la présence de cette dépendance sur le système. Le cas échéant, il définit alors un ensemble de variables telles

que la bibliothèque du paquet et le chemin d'accès vers ses fichiers d'en tête. L'appelant peut alors utiliser cette dépendance dans son projet.

21a ⟨*setup project dependencies* 21a⟩≡
```
⟨check for CMake packages 54d⟩
include_directories(
  ⟨list of paths where to look for the dependency headers 54e⟩
)
```

RCMake

Au delà des bibliothèques externes que nous ajouterons comme dépendances au fil du développement, nous souhaitons utiliser le projet RCMake, proposé par le Star-Engine, qui va nous aider à configurer le système de compilation. Pour ce faire, nous recherchons son paquet pour vérifier qu'il est installé sur le système au moins dans sa version 0.3. La directive REQUIRED signifie qu'en son absence le projet ne pourra pas se compiler.

21b ⟨*check for CMake packages* 54d⟩≡
```
find_package(RCMake 0.3 REQUIRED)
```

RCMake propose un ensemble de fichiers et scripts qui enrichissent les fonctionnalités initiales de CMake. Une fois RCMake trouvé, son paquet déclare la variable RCMAKE_SOURCE_DIR qui définit le répertoire dans lequel il est installé. Parmi ces fichiers, celui qui nous intéresse en premier lieu est le module rcmake.cmake, qui, à son inclusion, détectera notamment le compilateur utilisé et définira alors un jeu d'options de compilation pour notre applicatif. Cependant, bien que la directive include existe en CMake, cette dernière recherche les fichiers dans une liste de répertoires par défaut, liste qui n'a vraisemblablement aucune raison de contenir le répertoire où RCMake se trouve. Nous venons donc enrichir la variable CMAKE_MODULE_PATH avec le répertoire d'installation de RCMake avant d'inclure le fichier souhaité. CMake utilisera le contenu de cette variable pour rechercher le fichier à inclure s'il est introuvable dans les répertoires par défaut.

21c ⟨*setup project dependencies* 21a⟩+≡
```
set(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} ${RCMAKE_SOURCE_DIR})
include(rcmake)
```

RSys

Nous ajoutons également la dépendance à la bibliothèque RSys, elle aussi distribuée avec l'environnement de développement Star-Engine. Cette bibliothèque définit notamment des macros venant enrichir le fonctionnalités du seul langage C, ces nouvelles fonctionnalités étant généralement proposées par des extensions propres à chaque compilateur ou intégrées dans les normes plus récentes du langage. L'utilisation de RSys permet donc de s'abstraire du système hôte tout en restant sur une norme C conservatrice (en l'occurrence le C90 [ANS90]) seule garantie à même d'assurer la compatibilité de notre code sur une large gamme de systèmes, tout GNU/Linux fussent-ils.

Au delà des seules considérations de compatibilité, RSys met également en œuvre la notion d'allocateurs mémoire ou de compteurs de références ainsi que plusieurs type de conteneurs (tableau dynamique, table de hachage, liste chaînée, *etc.*). Il définit aussi plusieurs fonctionnalités système de bas niveau (exclusion mutuelle, opérations atomiques, *etc.*) et propose le support d'opérations d'algèbre linéaire sur des vecteurs ou des matrices carrées de 2 à 4 dimensions. RSys se présente donc comme une brique logicielle particulièrement pratique que nous utiliserons tout au long de nos développements. Nous ajoutons dès lors cette dépendance à notre projet.

21d ⟨*check for CMake packages* 54d⟩+≡
```
find_package(RSys 0.6 REQUIRED)
```

21e ⟨*list of paths where to look for the dependency headers* 54e⟩≡
```
${RSys_INCLUDE_DIR}
```

22a ⟨*list of used libraries* 54f⟩≡
```
RSys
```

3.2.3 Configuration de l'application

Nous définissons désormais une variable `CMake` dans laquelle nous listons les fichiers sources de notre application, à savoir les fichiers `*.c` et `*.h`

22b ⟨*configure the target* 22b⟩≡
```
set(PATH_TRACER_FILES
  ⟨list of source files 46c⟩
)
```

Ces fichiers source sont rédigés dans les parties suivantes et seront donc ajoutés à cette variable au fil de cet écrit. Cependant, `CMake` attend un chemin d'accès *absolu* vers les fichiers sources qu'il doit traiter. Pour éviter de devoir renseigner ce chemin absolu pour chaque fichier ajouté à la liste, nous post-traitons celle ci en ajoutant à chacun de ses éléments le chemin absolu du répertoire dans lequel il se trouve. Par convention nous décidons de stocker tous les fichiers source dans le sous répertoire `src` de notre projet. Et dans la section 3.2.1, nous avons stocké notre fichier de configuration dans le sous répertoire `cmake`. Le répertoire qui stocke les sources de notre application est donc `${PROJECT_SOURCE_DIR}/../src`, avec `PROJECT_SOURCE_DIR` la variable `CMake` définissant le chemin d'accès absolu vers notre fichier de configuration. Reste alors à ajouter ce répertoire à chacun des fichiers listés dans la variable `PATH_TRACER_FILES`. Pour ce faire, nous utilisons la fonction `rcmake_prepend_path` proposée par le module `RCMake`.

22c ⟨*configure the target* 22b⟩+≡
```
rcmake_prepend_path(PATH_TRACER_FILES ${PROJECT_SOURCE_DIR}/../src)
```

Nous définissons alors notre application : un programme exécutable de nom `pt` dont le code source est contenu dans les fichiers listés dans `PATH_TRACER_FILES`.

22d ⟨*configure the target* 22b⟩+≡
```
add_executable(pt ${PATH_TRACER_FILES})
```

Nous lions enfin cet exécutable à la liste des bibliothèques dont il dépend. Cette liste sera enrichie dans les parties suivantes, au cours du développement de notre application.

22e ⟨*configure the target* 22b⟩+≡
```
target_link_libraries(pt
  m # Math library
  ⟨list of used libraries 54f⟩
)
```

3.3 Arguments du programme

Cette partie décrit les structures de données et fonctions utilisées pour analyser et stocker les arguments soumis en entrée de la ligne de commande. Ces arguments sont utilisés pour paramétrer et contrôler le comportement de notre programme en définissant entre autres, la scène à rendre, le point de vue à partir duquel la scène est visualisée ou encore la définition de l'image ainsi rendue.

Nous décrivons donc *l'interface de programmation* des arguments du programme. Une interface de programmation est un ensemble de structures de données et de fonctions fournies à l'appelant pour programmer un applicatif spécifique. En `C` une interface de programmation est exposée dans un ou plusieurs fichiers d'en-tête et sa mise en œuvre est généralement définie dans un ou plusieurs fichiers sources. Ci-après, nous décrivons l'interface de programmation des arguments du programme dans le fichier d'en-tête `pt_args.h`, interface que nous mettons en œuvre dans le fichier source `pt_args.c`.

23a ⟨*src/pt_args.h* 23a⟩≡
```
⟨licensing 83a⟩

#ifndef PT_ARGS_H
#define PT_ARGS_H

⟨pt_args.h inclusions 23e⟩
⟨pt_args.h data types 23d⟩
⟨pt_args.h functions declarations 24a⟩

#endif /* PT_ARGS_H */
```

23b ⟨*src/pt_args.c* 23b⟩≡
```
⟨licensing 83a⟩

⟨pt_args.c features definitions 25a⟩

#include "pt_args.h"
⟨pt_args.c inclusions 25b⟩
⟨pt_args.c helper functions 26f⟩
⟨pt_args.c functions definitions 24b⟩
```

Ces 2 fichiers sont alors ajoutés à la liste des fichiers source à compiler.

23c ⟨*list of source files* 46c⟩≡
```
pt_args.h
pt_args.c
```

3.3.1 Structure de données et fonction d'initialisation

Dans le fichier d'en-tête, nous commençons par définir la structure qui stocke les arguments de la ligne de commande. Nous définissons également la constante `PT_ARGS_DEFAULT` qui représente les valeurs par défaut desdits arguments : si un argument n'est pas explicitement paramétré par l'utilisateur, nous prenons comme valeur du paramètre la valeur définie dans cette constante.

23d ⟨`pt_args.h` *data types* 23d⟩≡
```
struct pt_args {
  ⟨list of program arguments 26a⟩
};
static const struct pt_args PT_ARGS_DEFAULT = {
  ⟨default value of the program arguments 26b⟩
};
```

Pour initialiser cette structure nous utilisons les options soumises en entrée du programme. Comme tout programme `C` en ligne de commande, ces options sont listées dans un tableau de chaînes de caractères. Nous passons donc ce tableau en entrée de la fonction qui se charge d'initialiser les arguments à partir des options de la ligne de commande.

23e ⟨`pt_args.h` *inclusions* 23e⟩≡
```
#include <rsys/rsys.h>
```

24a ⟨pt_args.h *functions declarations* 24a⟩≡
```
extern LOCAL_SYM res_T
pt_args_init
  (struct pt_args* args,
   int argc, /* Number of arguments */
   char** argv); /* List of arguments */
```

Nous ne souhaitons pas rendre accessibles les fonctions de cette interface de programmation à l'extérieur du programme. Nous déclarons donc chacune de ces fonctions comme étant locales au programme à l'aide de la macro `LOCAL_SYM`, définie par la bibliothèque `RSys` dans son fichier d'en tête `rsys/rsys.h`. Nous utilisons également le type `res_T`, lui aussi défini dans le même fichier d'en-tête, pour notifier si l'exécution de notre fonction d'initialisation s'est bien passée. Le cas échéant, la fonction retourne la constante `RES_OK` ; toute autre valeur retournée signifiant qu'une erreur s'est produite.

Nous définissons cette fonction d'initialisation comme suit :

24b ⟨pt_args.c *functions definitions* 24b⟩≡
```
res_T
pt_args_init(struct pt_args* args, int argc, char** argv)
{
  ⟨pt_args_init local variables 25c⟩
  res_T res = RES_OK;
  ASSERT(args && argc && argv); /* Pre-condition */

  *args = PT_ARGS_DEFAULT; /* Setup default argument values */

  ⟨parse command line arguments 25d⟩

exit:
  return res;
error:
  ⟨rollback the pt_args_init process 24d⟩
  goto exit;
}
```

On notera qu'en cas d'erreur, nous invoquons le bloc `error` qui se charge de restaurer l'état d'exécution du programme tel qu'il était avant l'appel de cette fonction. Dans notre cas, cette "restauration" consiste à libérer les éventuelles allocations mémoire opérées sur la variable `args`. Pour ce faire, nous appelons la fonction `pt_args_release` que nous déclarons comme fonction d'interface et que nous définirons plus tard (section 3.3.9)

24c ⟨pt_args.h *functions declarations* 24a⟩+≡
```
extern LOCAL_SYM void
pt_args_release
  (struct pt_args* args);
```

24d ⟨*rollback the* pt_args_init *process* 24d⟩≡
```
pt_args_release(args);
```

Reste alors à analyser les options soumises en entrée du programme au travers du tableau de chaînes de caractères `argv`, et initialiser les membres de la variable `args` en conséquence. Les sections suivantes décrivent ces différentes étapes.

3.3.2 Aperçu des options du programme

Notre programme propose plusieurs paramètres permettant de contrôler son exécution. Chaque paramètre se présente sous la forme d'une option de ligne de commande, composée d'une lettre unique représentant le paramètre à configurer, précédé du caractère `'-'`, et suivi éventuellement d'une chaîne de caractères décrivant la valeur de l'option en question ; certaines options n'ont pas de valeur car ne servant qu'à activer un comportement particulier. Les différentes options proposées par notre programme sont les suivantes :

- l'option `-d` contrôle la définition de l'image à rendre ;
- l'option `-n` détermine le nombre de réalisations Monte-Carlo utilisées pour évaluer la luminance d'un pixel de l'image ;
- Les options `-p`, `-t` et `-u` définissent, respectivement, la position, le point de visée et l'axe vertical de la camera. En d'autre termes elles contrôlent le point de vue ;
- l'option `-i` définit le nom du fichier en entrée qui décrit les données de la scène à rendre ;
- l'option `-o` définit le nom du fichier dans lequel l'image calculée est sauvegardée ;
- l'option `-F` contrôle l'angle d'ouverture de notre caméra (en degrés) ;
- l'option `-f` permet de forcer l'écriture de l'image même si un fichier portant le même nom existe déjà ;
- l'option `-r` détermine la réflectance de la scène ;
- enfin, l'option `-h` affiche une brève documentation de notre programme.

Pour analyser les arguments en entrée de la commande nous utilisons la fonction POSIX `getopt` qui itère sur ses options et retourne à chaque itération la lettre correspondante à l'option courante, ou -1 si toutes les options ont déjà été analysées. L'éventuelle valeur d'une option est quant à elle stockée dans la variable globale `optarg`. Les options légitimes au regard de notre application sont listées dans une chaîne de caractères soumise en dernier argument de `getopt` : les options simples sont listées par la lettre les caractérisant et les options attendant une valeur font suivre leur lettre du caractère `':'`. Dans notre cas, seules les options `-f` et `-h` n'attendent aucune valeur particulière.

25a ⟨`pt_args.c` *features definitions* 25a⟩≡
```
#define _POSIX_C_SOURCE 2 /* getopt support */
```

25b ⟨`pt_args.c` *inclusions* 25b⟩≡
```
#include <getopt.h>
```

25c ⟨`pt_args_init` *local variables* 25c⟩≡
```
int opt;
```

25d ⟨*parse command line arguments* 25d⟩≡
```
while((opt = getopt(argc, argv, "d:i:n:p:t:u:o:F:fr:h")) != -1) {
  switch(opt) {
    ⟨parse options 26d⟩
    default: res = RES_BAD_ARG; break; /* Unexpected option */
  }
  if(res != RES_OK) {
    if(optarg) {
      fprintf(stderr, "%s: invalid option argument '%s' -- '%c'\n",
        argv[0], optarg, opt);
    }
    goto error;
  }
}
```

3.3.3 Options de l'image

Nous commençons par analyser les options qui contrôlent l'image à rendre. Nous ajoutons tout d'abord une structure anonyme `image` dans la liste des arguments. Cette structure contient la définition de l'image

et le nombre de réalisations Monte-Carlo utilisées pour évaluer la luminance de chaque pixel. Nous n'oublions pas de fixer la valeur par défaut de ces 2 arguments à savoir une définition de 320 par 240 pixels et un seul échantillon par pixel.

26a ⟨*list of program arguments* 26a⟩≡
```
struct {
  unsigned definition[2];
  unsigned spp; /* #samples per pixel */
} image;
```

26b ⟨*default value of the program arguments* 26b⟩≡
```
{
  {320, 240},/* Image definition */
  1 /* #samples per pixel */
},
```

Nous pouvons désormais changer ces valeurs par défaut avec les options de la ligne de commande. Nous commençons par traiter l'option `-n` qui contrôle le nombre de réalisations par pixel. Nous utilisons la fonction de la bibliothèque `RSys`, `cstr_to_uint`, définie dans le fichier d'en-tête `rsys/cstr.h`, pour convertir la chaîne de caractères tapée par l'utilisateur en entier non signé. Cette conversion peut échouer si, par exemple, la chaîne de caractères en entrée ne représente pas un entier non signé valide. Si la conversion est un succès, nous vérifions alors la validité de l'argument au regard de l'attendu du programme, à savoir que ce nombre ne peut pas être nul.

26c ⟨`pt_args.c` *inclusions* 25b⟩+≡
```
#include <rsys/cstr.h>
```

26d ⟨*parse options* 26d⟩≡
```
case 'n':
  res = cstr_to_uint(optarg, &args->image.spp);
  if(res == RES_OK && args->image.spp == 0) res = RES_BAD_ARG;
  break;
```

Nous poursuivons avec l'option `-d` qui contrôle la définition de l'image. Pour ce faire nous utilisons une fonction intermédiaire, `parse_definition`, qui analyse l'option et vérifie sa validité.

26e ⟨*parse options* 26d⟩+≡
```
case 'd':
  res = parse_definition(optarg, args->image.definition);
  break;
```

Par convention, on impose à l'utilisateur de définir la définition d'image par une chaîne de caractères composée de deux entiers non signés séparés par le caractère `'x'` ; le premier et le second entier représentant le nombre de pixels de l'image selon, respectivement, son axe horizontal et vertical. Nous utilisons la fonction `cstr_to_list_uint` qui convertit une chaîne de caractères en une liste d'entiers. Cette fonction est elle aussi proposée par la bibliothèque `RSys`. Elle prend en arguments la chaîne de caractères à traiter ainsi que le caractère utilisé comme séparateur des éléments de la liste ; ici le caractère `'x'`. Elle retourne alors le tableau d'entiers convertis ainsi que le nombre total d'entiers détectés dans la chaîne de caractères. Le dernier argument fixe le nombre maximum d'entiers qui peuvent être stockés dans le tableau en sortie (ici, 2 entiers).

26f ⟨`pt_args.c` *helper functions* 26f⟩≡
```
static res_T
parse_definition(const char* string, unsigned definition[2])
{
  size_t nintegers;
  res_T res = RES_OK;
```

```
  ASSERT(string && definition);

  res = cstr_to_list_uint(string, 'x', definition, &nintegers, 2);
  if(res != RES_OK) /* Error during conversion */
    return res;
  if(nintegers != 2) /* Too many integers in the submitted string */
    return RES_BAD_ARG;
  if(definition[0] > 16384 /* Definition is too large */
  || definition[1] > 16384)
    return RES_BAD_ARG;

  return RES_OK;
}
```

3.3.4 Options de caméra

Nous définissons une caméra par quatre paramètres : sa position, son point de visée, son axe vertical et son angle d'ouverture. Par défaut, la caméra est positionnée à l'origine du repère, regarde le long de l'axe Y, a pour axe vertical l'axe Z et possède un angle d'ouverture de 70 degrés.

27a ⟨*list of program arguments* 26a⟩+≡

```
struct {
  double pos[3]; /* Position */
  double tgt[3]; /* Targeted point */
  double up[3]; /* Vertical axis */
  double field_of_view; /* Field of view in degrees */
} camera;
```

27b ⟨*default value of the program arguments* 26b⟩+≡

```
{
  {0, 0, 0}, /* Camera position */
  {0, 1, 0}, /* Camera target */
  {0, 0, 1}, /* Camera up vector */
  70.0 /* Camera field of view in degrees */
},
```

Hormis l'angle d'ouverture, tous ces arguments sont des vecteurs de réels à 3 dimensions. Par conséquent, l'analyse des options les configurant est strictement la même. Nous utilisons la fonction de `RSys`, `cstr_to_list_double`, qui, à l'instar de la fonction `cstr_to_list_uint` utilisée pour analyser la définition d'image (section 3.3.3), convertit une chaîne de caractères en une liste, mais cette fois une liste de réels en double précision en lieu et place d'une liste d'entiers. Par convention, on impose que chaque réel soit séparé du précédent par une virgule. Si la conversion ne retourne pas d'erreur, nous nous assurons alors que la dimension du vecteur est bien de 3. Nous factorisons l'ensemble de cette analyse dans une fonction intermédiaire que nous invoquons pour analyser les options qui caractérisent la position, le point de visée et l'axe vertical de la caméra.

27c ⟨`pt_args.c` *helper functions* 26f⟩+≡

```
static res_T
parse_real3(const char* string, double reals[3])
{
  size_t nreals;
  res_T res = RES_OK;
  ASSERT(string && reals);

  res = cstr_to_list_double(string, ',', reals, &nreals, 3);
  if(res == RES_OK && nreals != 3) res = RES_BAD_ARG;

  return res;
}
```

28a ⟨*parse options* 26d⟩+≡
```
case 'p': res = parse_real3(optarg, args->camera.pos); break;
case 't': res = parse_real3(optarg, args->camera.tgt); break;
case 'u': res = parse_real3(optarg, args->camera.up); break;
```

Nous ne vérifions pas ici la validité de la caméra ainsi définie ; un utilisateur peut soumettre 3 vecteurs de réels sans que ceux ci puissent représenter une caméra valide. C'est notamment le cas si sa position et son point de visée sont confondus. Nous laissons la vérification de la consistances de ces paramètres au code qui crée une caméra (section 3.4.1)

Reste alors à analyser l'option définissant l'angle d'ouverture de la caméra. Là aussi nous laissons la fonction de création d'une caméra vérifier la validité de cet angle d'ouverture, seule la conversion de cette option en un réel valide est ici vérifiée.

28b ⟨*parse options* 26d⟩+≡
```
case 'F': res = cstr_to_double(optarg, &args->camera.field_of_view); break;
```

3.3.5 Option du matériau de la scène

Notre scène n'est composée que d'un seul matériau diffus dont la réflectance est configurée par l'utilisateur via l'option `-r`. Nous ajoutons donc une variable aux arguments de notre application pour définir cette réflectance qui, par défault, est initialisée à 1.

28c ⟨*list of program arguments* 26a⟩+≡
```
double reflectivity;
```

28d ⟨*default value of the program arguments* 26b⟩+≡
```
1, /* Reflectivity */
```

Puis nous analysons la valeur de l'option `-r` et validons que non seulement elle représente bien un réel mais que ce réel définit une réflectance valide, c'est à dire comprise entre $[0, 1]$.

28e ⟨*parse options* 26d⟩+≡
```
case 'r':
  res = cstr_to_double(optarg, &args->reflectivity);
  if(res == RES_OK && (args->reflectivity>1 || args->reflectivity <0)) {
    res = RES_BAD_ARG;
  }
  break;
```

3.3.6 Option du fichier en entrée

Les données de la scène sont contenues dans le fichier défini par l'option `-i`. Nous ajoutons à la liste des arguments une chaîne de caractères qui stocke le nom de ce fichier. Par défaut, ce nom de fichier est non défini. Dans ce cas, nous considérerons alors que les données de la scène seront envoyées sur l'entrée standard de la ligne de commande.

28f ⟨*list of program arguments* 26a⟩+≡
```
const char* input_filename;
```

28g ⟨*default value of the program arguments* 26b⟩+≡
```
NULL, /* Input filename */
```

L'analyse de cette option se limite à sauvegarder simplement la valeur courante d'`optarg` définie par la fonction `getopt`. À noter que nous n'avons pas besoin de dupliquer la chaîne de caractères pointée par `optarg` qui en définitive ne fait que pointer un des arguments de `argv`, argument qui ne sera pas modifié au cours de l'exécution du programme et que l'on peut donc pointer directement.

29a ⟨*parse options* 26d⟩+≡
```
case 'i': args->input_filename = optarg; break;
```

3.3.7 Options du fichier en sortie

Le nom du fichier dans lequel l'image est stockée est renseigné par l'option `-o`. Nous définissons dans la structure `struct pt_args` une chaîne de caractères stockant ce nom de fichier qui par défaut n'est pas renseigné et pointe sur `NULL`. Dans ce cas, l'image sera écrite sur la sortie standard ce qui autorisera l'appelant à re-diriger l'image dans un fichier via le shell, ou de la chaîner à l'entrée d'une autre application à l'aide d'un *pipe* UNIX.

29b ⟨*list of program arguments* 26a⟩+≡
```
const char* output_filename;
```

29c ⟨*default value of the program arguments* 26b⟩+≡
```
NULL, /* Output filename */
```

Tout comme le nom du fichier d'entrée (section 3.3.6), l'analyse de cette option consiste à copier le pointeur `optarg`.

29d ⟨*parse options* 26d⟩+≡
```
case 'o': args->output_filename = optarg; break;
```

Nous traitons maintenant l'option permettant de forcer la ré-écriture d'un fichier. Par défaut, nous interdisons de ré-écrire un fichier déjà existant sur disque et ce afin de prévenir qu'une image, ou tout autre fichier portant le même nom, ne soit écrasé par erreur. Nous autorisons la ré-écriture d'un fichier existant seulement si l'option `-f` est explicitement définie.

29e ⟨*list of program arguments* 26a⟩+≡
```
int force_overwriting;
```

29f ⟨*default value of the program arguments* 26b⟩+≡
```
0, /* Force overwriting */
```

29g ⟨*parse options* 26d⟩+≡
```
case 'f': args->force_overwriting = 1; break;
```

3.3.8 Aide de la commande

Enfin nous analysons l'option permettant d'afficher une aide de notre programme. Si l'option `-h` est définie, nous invoquons une fonction intermédiaire qui affiche sur la sortie standard une brève description du programme et de ses options. Nous passons en paramètre de cette fonction le nom de la ligne de commande, nom qui est défini comme premier argument de la ligne commande. Nous notifions alors qu'aucun autre traitement n'est attendu et que par conséquent nous souhaitons quitter l'application. Pour ce faire, nous ajoutons l'option `quit` à la liste des arguments, option qui par défaut n'est pas activée. Nous quittons alors la fonction d'initialisation des arguments puisqu'il est dès lors inutile de poursuivre leur analyse ; l'option `quit` notifiant que nous allons stopper l'exécution du programme.

29h ⟨*list of program arguments* 26a⟩+≡
```
int quit;
```

29i ⟨*default value of the program arguments* 26b⟩+≡
```
0, /* Quit */
```

30a ⟨*parse options* 26d⟩+≡
```
case 'h':
  print_help(argv[0]);
  args->quit = 1;
  goto exit;
```

Dans la fonction `print_help`, nous affichons l'aide de la commande suivant la convention usuelle pour ce type d'application. Nous commençons d'abord par donner son synopsis avant de décrire brièvement ce qu'elle réalise. Puis nous listons ses options. Enfin nous affichons le propriétaire des droits patrimoniaux du code et la licence qui le régit.

30b ⟨`pt_args.c` *helper functions* 26f⟩+≡
```
static void
print_help(const char* cmd_name)
{
  ASSERT(cmd_name);
  ⟨print command synopsis 30c⟩
  ⟨print program description 30d⟩
  ⟨list program options 30e⟩
  ⟨print copyright and program license 31a⟩
}
```

Les options de notre commande sont toutes facultatives. Nous le notifions dans le synopsis en utilisant la norme POSIX utilisée pour ce ce type d'application [Gro18] : ces options sont affichées entre crochets. À noter que nous utilisons la variable `cmd_name` pour afficher le nom de la commande, évitant ainsi à cette fonction de devoir connaître à priori ce nom.

30c ⟨*print command synopsis* 30c⟩≡
```
printf("Usage: %s [OPTION] ...\n", cmd_name);
```

Nous affichons ensuite un bref descriptif de notre programme, en l'occurrence une application qui calcule une image par suivi de chemins.

30d ⟨*print program description* 30d⟩≡
```
printf("Render a scene with a Monte-Carlo backward path-tracer\n\n");
```

Nous listons alors les options de la ligne de commande par ordre alphabétique. Chaque option est accompagnée d'un bref descriptif et sa valeur par défaut est affichée en utilisant le contenu de la constante `PT_ARGS_DEFAULT`.

30e ⟨*list program options* 30e⟩≡
```
printf(
  "  -d WIDTHxHEIGHT  image definition. Default value is %ux%u.\n",
  PT_ARGS_DEFAULT.image.definition[0],
  PT_ARGS_DEFAULT.image.definition[1]);
printf(
  "  -F               define the camera field of view (in degrees). Its default\n"
  "                   value is %g degrees.\n",
  PT_ARGS_DEFAULT.camera.field_of_view);
printf(
  "  -f               overwrite the OUTPUT file if it already exists.\n");
printf(
  "  -h               display this help and exit.\n");
printf(
  "  -i INPUT         file of the scene to render. If not defined, scene data are\n"
  "                   read from standard input.\n");
printf(
```

```
    "  -n NREALISATIONS number of per pixel realisations.\n");
  printf(
    "  -o OUTPUT        file where the rendered image is written. If not defined,\n"
    "                   the image is written on standard output.\n");
  printf(
    "  -p X,Y,Z         camera position. Its default value is [%g, %g, %g].\n",
    PT_ARGS_DEFAULT.camera.pos[0],
    PT_ARGS_DEFAULT.camera.pos[1],
    PT_ARGS_DEFAULT.camera.pos[2]);
  printf(
    "  -r REFLECTIVITY  scene reflectivity. Its default value is %g.\n",
    PT_ARGS_DEFAULT.reflectivity);
  printf(
    "  -t X,Y,Z         camera target. Its default value is [%g, %g, %g].\n",
    PT_ARGS_DEFAULT.camera.tgt[0],
    PT_ARGS_DEFAULT.camera.tgt[1],
    PT_ARGS_DEFAULT.camera.tgt[2]);
  printf(
    "  -u X,Y,Z         camera up vector. Its default value is [%g, %g, %g].\n",
    PT_ARGS_DEFAULT.camera.up[0],
    PT_ARGS_DEFAULT.camera.up[1],
    PT_ARGS_DEFAULT.camera.up[2]);
  printf("\n");
```

Enfin, nous affichons que |Meso|Star> est le propriétaire patrimonial du code avant de notifier à l'utilisateur la license sous laquelle est distribué le programme, en l'occurrence la *GNU General Public License* dans sa version 3 [GPL]. Comme pour le synopsis, nous utilisons le paramètre `cmd_name` pour afficher le nom du programme sans l'avoir fixé à priori.

31a ⟨*print copyright and program license* 31a⟩≡

```
  printf(
    "%s (C) 2019 |Meso|Star>. This is free software released under the GNU GPL\n"
    "license, version 3 or later. You are free to change or redistribute it under\n"
    "certain conditions <http://gnu.org/licenses/gpl.html>.\n",
    cmd_name);
```

3.3.9 Libérer les arguments

Aucune donnée listée dans la structure des arguments ne nécessite de traitement particulier après avoir été initialisée. La fonction de libération des arguments ne réalise donc aucune action. Bien qu'elle paraisse dès lors inutile, nous conservons cette fonction afin d'automatiser chez l'appelant le réflexe qui lui dicte qu'une fois initialisée une variable doit être libérée. Cette stricte symétrie lui évite ainsi de se demander quand et pourquoi ce ne serait pas le cas, facilitant par la même occasion la maintenance du code lorsque au cours de son évolution la fonction de libération en vient à réaliser effectivement une action.

31b ⟨`pt_args.c` *functions definitions* 24b⟩+≡

```
  void
  pt_args_release(struct pt_args* args)
  {
    ASSERT(args);
    (void)args; /* Avoid the "unused variable" warning */
  }
```

3.4 Caméra

Dans cette section nous mettons en œuvre une caméra de type sténopé, c'est à dire une caméra sans effet de profondeur de champ. Une caméra décrit le point de vue à partir duquel l'image de la scène est

rendue. Elle se définit par une position, une orientation, un angle d'ouverture et un rapport de projection entre la largeur et la hauteur de l'image. La définition de l'image à rendre reste cependant un attribut de l'image en tant que telle ; une même caméra peut donc être utilisée pour rendre des images de différentes définitions.

Dans le fichier d'en-tête `pt_camera.h` nous déclarons l'interface de programmation de la caméra, c'est à dire les structures de données et fonctions que l'appelant peut utiliser pour la manipuler. Cette interface est alors mise en œuvre dans un fichier source séparé.

32a ⟨*src/pt_camera.h* 32a⟩≡
```
⟨licensing 83a⟩

#ifndef PT_CAMERA_H
#define PT_CAMERA_H

⟨pt_camera.h inclusions 32f⟩
⟨pt_camera.h data types 32d⟩
⟨pt_camera.h functions declarations 33b⟩

#endif /* PT_CAMERA_H */
```

32b ⟨*src/pt_camera.c* 32b⟩≡
```
⟨licensing 83a⟩

#include "pt_camera.h"
⟨pt_camera.c inclusions 34a⟩
⟨pt_camera.c data types 32e⟩
⟨pt_camera.c helper functions 39f⟩
⟨pt_camera.c functions definitions 33c⟩
```

32c ⟨*list of source files* 46c⟩≡
```
pt_camera.h
pt_camera.c
```

3.4.1 Créer une caméra

Nous commençons tout d'abord par déclarer la structure de données qui représente une camera. Par conception, nous n'autorisons la manipulation d'une variable caméra qu'à travers ses fonctions d'interface. Nous pouvons dès lors cacher la représentation mémoire de ce type à l'appelant et effectuer une *déclaration avancée* de cette structure ; sa définition à proprement parler se trouvant dans le fichier `C`.

32d ⟨`pt_camera.h` *data types* 32d⟩≡
```
struct pt_camera; /* Forward declaration of the camera data type */
```

32e ⟨`pt_camera.c` *data types* 32e⟩≡
```
struct pt_camera {
  ⟨camera point of view members 35g⟩
  ⟨camera miscellaneous members 34d⟩
};
```

En masquant dans le fichier d'en tête la représentation mémoire de la structure caméra, nous interdisons la création d'une variable de ce type directement sur la pile. Pour créer une caméra, l'appelant doit passer par sa fonction de création, fonction proposée par notre interface de programmation et qui se charge d'allouer et d'initialiser l'espace mémoire de la caméra avant de retourner à l'utilisateur un pointeur vers cet espace mémoire nouvellement créé.

32f ⟨`pt_camera.h` *inclusions* 32f⟩≡
```
#include <rsys/rsys.h>
```

33a ⟨pt_camera.h *data types* 32d⟩+≡
```
struct pt;
```

33b ⟨pt_camera.h *functions declarations* 33b⟩≡
```
extern LOCAL_SYM res_T
pt_camera_create
  (struct pt* pt,
   const double position[3],
   const double target[3],
   const double up[3],
   const double image_ratio, /* Width / Height */
   const double field_of_view, /* Vertical field of view in radians */
   struct pt_camera** camera); /* Output camera */
```

Dans notre mise en œuvre, nous choisissons de définir le point de vue de la caméra à sa création, évitant ainsi de devoir l'initialiser avec un point de vue par défaut qui devrait être modifié par la suite avec le point de vue souhaité. Par conséquent, en plus d'un pointeur vers la variable représentant notre programme (`pt`), nous passons en entrée de cette fonction les paramètres permettant de calculer le point de vue de la caméra, à savoir : sa position, son point de visée, son axe vertical, le rapport entre la largeur et la hauteur de l'image et l'angle d'ouverture vertical (en radians). Enfin, comme dernier argument, nous passons l'adresse du pointeur dans lequel la caméra ainsi créée est retournée.

À noter que nous effectuons une déclaration avancée du type structuré `struct pt` signifiant là aussi que ce type existe mais que sa représentation est définie ailleurs (section 3.6). Cette déclaration autorise l'utilisation du type `struct pt*` comme type d'argument de la fonction de création.

Par ailleurs, l'interface de programmation que nous définissons n'a pas vocation à être accessible à l'extérieur du programme ; nous déclarons donc l'ensemble de ses fonctions comme étant locales à l'applicatif à l'aide de la macro `LOCAL_SYM` définie dans `rsys/rsys.h`. Enfin, nous utilisons le type `res_T`, défini dans le même fichier d'en-tête, pour notifier en retour de fonction si une erreur s'est produite pendant son exécution.

La mise en œuvre de cette fonction de création se trouve dans le fichier `C` et se structure comme suit :

33c ⟨pt_camera.c *functions definitions* 33c⟩≡
```
res_T
pt_camera_create
  (struct pt* pt,
   const double position[3],
   const double target[3],
   const double up[3],
   const double image_ratio,
   const double field_of_view,
   struct pt_camera** camera)
{
  ⟨pt_camera_create local variables 34e⟩
  res_T res = RES_OK;

  ⟨check the pt_camera_create input parameters 34b⟩
  ⟨create the camera 34f⟩
  ⟨setup the camera point of view 36a⟩

exit:
  ⟨setup the pt_camera_create output 35a⟩
  return res;
error:
  ⟨rollback the pt_camera_create process 35f⟩
```

```
    goto exit;
  }
```

Avant de créer et d'initialiser la caméra, nous vérifions tout d'abord que les paramètres d'entrée de la fonction sont valides. Dans le cas contraire nous notifions le problème rencontré dans le journal d'évènements de l'application (section 3.6.3) avant de retourner une erreur.

34a ⟨pt_camera.c *inclusions* 34a⟩≡
```
  #include "pt.h"
  #include <rsys/math.h> /* Define the PI constant */
```

34b ⟨*check the* pt_camera_create *input parameters* 34b⟩≡
```
  if(!pt || !position || !target || !up || !camera) {
    logger_print(&pt->logger, LOG_ERROR,
      "%s: invalid NULL argument\n", FUNC_NAME);
    res = RES_BAD_ARG;
    goto error;
  }
  if(field_of_view <= 0 || field_of_view >= PI) {
    logger_print(&pt->logger, LOG_ERROR,
      "%s: invalid vertical field of view ‘%g’\n", FUNC_NAME, field_of_view);
    res = RES_BAD_ARG;
    goto error;
  }
  if(image_ratio <= 0) {
    logger_print(&pt->logger, LOG_ERROR,
      "%s: invalid image ratio ‘%g’\n", FUNC_NAME, field_of_view);
    res = RES_BAD_ARG;
    goto error;
  }
```

Nous allouons désormais l'espace mémoire de notre caméra. Pour ce faire, nous déclarons la variable intermédiaire `cam` qui pointe vers la caméra que nous souhaitons créer. À sa définition, cette caméra n'a pas encore d'espace mémoire alloué et pointe donc sur `NULL`. Nous utilisons alors l'allocateur de l'application (section 3.6.2) pour allouer l'espace mémoire de la structure caméra. Nous vérifions que l'allocation s'est bien passée afin de retourner une erreur mémoire dans le cas contraire. Le cas échéant, nous conservons le pointeur vers la variable représentant notre application et ce afin d'avoir notamment accès à son journal d'évènements ou son allocateur, allocateur que nous devrons réutiliser pour libérer l'espace mémoire que nous venons d'allouer (section 3.4.4). Nous affectons enfin à l'adresse pointée par la variable de sortie `camera` l'espace mémoire que nous avons alloué.

34c ⟨pt_camera.c *inclusions* 34a⟩+≡
```
  #include <rsys/mem_allocator.h>
```

34d ⟨*camera miscellaneous members* 34d⟩≡
```
  struct pt* pt;
```

34e ⟨pt_camera_create *local variables* 34e⟩≡
```
  struct pt_camera* cam = NULL;
```

34f ⟨*create the camera* 34f⟩≡
```
  cam = MEM_CALLOC(&pt->allocator, 1, sizeof(*cam));
  if(!cam) {
    logger_print(&pt->logger, LOG_ERROR,
      "%s: could not allocate the camera data structure\n", FUNC_NAME);
    res = RES_MEM_ERR;
    goto error;
  }
  cam->pt = pt;
  ⟨init the camera reference counter to 1 35d⟩
```

35a ⟨*setup the* `pt_camera_create` *output* 35a⟩≡
```
if(camera) *camera = cam;
```

Nous gérons l'espace mémoire alloué pour une caméra à l'aide d'un compteur de références. Pour cela, nous nous appuyons sur la bibliothèque `RSys` et son fichier d'en-tête `rsys/ref_count.h` qui déclare l'interface de programmation d'un compteur de références. Dans cette mise en œuvre, une variable de type `ref_T` doit être ajoutée à notre type pour stocker combien de fois une variable de type caméra est effectivement référencée. Nous initialisons ce compteur dès que l'espace mémoire de la caméra est alloué et ce afin de pouvoir directement gérer sa durée de vie en se basant sur ce mécanisme.

35b ⟨`pt_camera.c` *inclusions* 34a⟩+≡
```
#include <rsys/ref_count.h>
```

35c ⟨*camera miscellaneous members* 34d⟩+≡
```
ref_T ref;
```

35d ⟨*init the camera reference counter to 1* 35d⟩≡
```
ref_init(&cam->ref);
```

En retour de la fonction `ref_init` le compteur de la variable `cam` est égal à 1 : dit autrement nous attribuons automatique une référence sur celle-ci à l'appelant. Nous déclarons alors deux nouvelles fonctions d'interface, `pt_camera_ref_get` et `pt_camera_ref_put` lui permettant, respectivement, d'incrémenter et de décrémenter le nombre de références qu'il détient. Quand la valeur du compteur de références atteint 0, l'espace mémoire de la caméra est alors silencieusement libéré. Le mise en œuvre de ces deux fonctions est défini ultérieurement (section 3.4.4).

35e ⟨`pt_camera.h` *functions declarations* 33b⟩+≡
```
extern LOCAL_SYM void
pt_camera_ref_get
  (struct pt_camera* camera);

extern LOCAL_SYM void
pt_camera_ref_put
  (struct pt_camera* camera);
```

Afin d'éviter toute fuite mémoire, nous devons désormais traiter le cas où une erreur serait détectée dans la fonction de création et ce après que la caméra ait été allouée avec succès. Dans ce cas, nous devons libérer l'espace mémoire alloué avant le retour de la fonction ; toute erreur invalidant les traitements opérés jusqu'alors. Pour cela, nous décrémentons simplement le compteur de références à l'aide de la fonction d'interface `pt_camera_ref_put` notifiant que nous relâchons la référence qui nous a été attribuée lors de l'appel précédent à `ref_init`.

35f ⟨*rollback the* `pt_camera_create` *process* 35f⟩≡
```
if(cam != NULL) {
  pt_camera_ref_put(cam);
  cam = NULL;
}
```

3.4.2 Calcul du point de vue

Nous représentons le point de vue de notre caméra à l'aide d'un repère orthogonal dans lequel l'axe Y représente l'axe vertical et l'axe Z son axe de visée.

35g ⟨*camera point of view members* 35g⟩≡
```
double axis_x[3];
double axis_y[3];
double axis_z[3];
double position[3]; /* frame origin */
```

Le calcul de ce point de vue se décompose en deux étapes. Dans un premier temps nous calculons le repère orthonormé de la caméra. Puis, dans un second temps, nous déformons ce repère en accord avec le rapport d'image souhaité par l'utilisateur et l'angle d'ouverture de la caméra.

36a ⟨*setup the camera point of view* 36a⟩≡
```
⟨compute the orthonormal camera frame 36c⟩
⟨adjust the camera frame wrt the image ratio and the field of view 37c⟩
```

Dans la fonction de création nous commençons donc par calculer un repère orthonormé représentant le point de vue de la caméra. Nous utilisons pour cela les fonctions proposées par la bibliothèque `RSys`, dans le fichier d'en-tête `rsys/double3.h`, qui manipulent des vecteurs à 3 dimensions. Parmi ces fonctions, nous utilisons notamment la fonction `d3_normalize` qui normalise un vecteur et retourne sa longueur initiale. Nous vérifions cette longueur pour s'assurer que les paramètres fournis en entrée de la fonction permettent bien de définir une caméra valide. Par exemple une position et un point de visée confondus définissent un vecteur Z null. En cas d'erreur de cette nature, nous affichons un message dans le journal d'évènements de l'application avant de retourner une erreur.

36b ⟨`pt_camera.c` *inclusions* 34a⟩+≡
```
#include <rsys/double3.h>
```

36c ⟨*compute the orthonormal camera frame* 36c⟩≡
```
d3_set(cam->position, position);
d3_sub(cam->axis_z, target, position);
d3_cross(cam->axis_x, cam->axis_z, up);
d3_cross(cam->axis_y, cam->axis_z, cam->axis_x);

if(d3_normalize(cam->axis_x, cam->axis_x) <= 0
|| d3_normalize(cam->axis_y, cam->axis_y) <= 0
|| d3_normalize(cam->axis_z, cam->axis_z) <= 0) {
  logger_print(&pt->logger, LOG_ERROR,
    "%s: invalid camera point of view:\n"
    "\tposition = [%g, %g, %g]\n"
    "\ttarget   = [%g, %g, %g]\n"
    "\tup       = [%g, %g, %g]\n",
    FUNC_NAME,
    position[0], position[1], position[2],
    target[0], target[1], target[2],
    up[0], up[1], up[2]);
  res = RES_BAD_ARG;
  goto error;
}
```

Dans le repère orthonormal que nous venons de calculer, l'axe Y ne pointe pas vers le demi espace orthogonal au vecteur vertical `up`, mais vers son opposé. En d'autres termes, il est orienté vers le bas du plan image. L'objectif ici est de correspondre au mieux avec la structuration mémoire de l'image dont il est à priori entendu que les lignes soient ordonnées de haut en bas (section 3.5). Sans cette inversion, l'image serait donc inversée en Y et demanderait alors à être ré-inversée lors de son écriture en sortie. En inversant l'axe Y de la caméra nous cloisonnons cette question de changement de repère à la seule caméra, évitant ainsi à l'utilisateur de devoir se poser la question.

Nous allons désormais déformer le repère que nous venons de construire en fonction de l'angle d'ouverture de la caméra et du rapport entre la largeur et la hauteur de l'image. Nous cherchons ici à déformer la pyramide de vue de notre caméra, c'est à dire la pyramide dont le sommet est défini par la position de la caméra et dont la base est orthogonale à l'axe Z de celle ci. Après déformation, l'angle au sommet de la pyramide autour de l'axe X doit être égal à l'angle d'ouverture de la caméra, et le rapport des longueurs de la base de la pyramide en X et en Y doit quant à lui être identique au rapport entre la largeur et

la hauteur de l'image à rendre. Pour cela, nous choisissons de fixer la longueur de l'axe vertical à 1. En d'autres termes, l'axe Y restera normé, seuls les axes X et Z seront déformés. Ces axes correspondent respectivement à la largeur de la base et la hauteur de la pyramide de vue. À partir de l'angle d'ouverture de la caméra nous calculons cette hauteur `h` que nous utilisons comme norme de l'axe Z. L'axe X est quant à lui simplement mis à l'échelle par le rapport entre la largeur et la hauteur de l'image à rendre.

37a ⟨`pt_camera.c` *inclusions* 34a⟩+≡
```
#include <math.h> /* tan definition */
```

37b ⟨`pt_camera_create` *local variables* 34e⟩+≡
```
double h;
```

37c ⟨*adjust the camera frame wrt the image ratio and the field of view* 37c⟩≡
```
h = 1.0/* Half length of the pyramid base along Y */
  / tan(field_of_view*0.5);
d3_muld(cam->axis_z, cam->axis_z, h);
d3_muld(cam->axis_x, cam->axis_x, image_ratio);
```

Le repère que nous venons de calculer est en définitive une transformation permettant de passer une coordonnée définie dans le plan image de la caméra, en une position dans le repère absolu de la scène. Soit p une position dans le repère caméra, d'aucun peut transformer p dans le repère absolu par la transformation affine suivante :

$$p' = O + p \times \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \tag{3.1}$$

Avec O la position de la caméra, c'est à dire `cam->position` dans notre code `C`, et X, Y, et Z les axes du repère caméra que nous venons de calculer, soit dans notre code les variables `cam->axis_x`, `cam->axis_y` et `cam->axis_z`. Dans le repère caméra, les coordonnées $\{-1, -1, 1\}$ et $\{1, 1, 1\}$ représentent donc, respectivement, le coin supérieur gauche et le coin inférieur droit de l'image. Dans la section suivante (section 3.4.3) nous utilisons cette transformation pour générer des rayons issus de la caméra.

Enfin, nous conservons comme variable membre de la caméra le vecteur `up` utilisé pour construire son repère. Dans l'algorithme de rendu que nous mettons en œuvre, nous avons besoin de déterminer où se trouve la voûte céleste qui est en l'occurrence la seule source lumineuse de la scène. Et pour cela nous utilisons ce vecteur `up` dont nous admettons qu'il pointe vers le ciel. Nous ajoutons dès lors cette variable à la structure caméra ainsi qu'une fonction d'accès à celle-ci afin de la rendre accessible à l'appelant.

37d ⟨*camera point of view members* 35g⟩+≡
```
double up[3];
```

37e ⟨*compute the orthonormal camera frame* 36c⟩+≡
```
d3_set(cam->up, up);
```

37f ⟨`pt_camera.h` *functions declarations* 33b⟩+≡
```
extern LOCAL_SYM void
pt_camera_get_up
  (const struct pt_camera* camera,
   double up[3]);
```

37g ⟨`pt_camera.c` *functions definitions* 33c⟩+≡
```
void
pt_camera_get_up(const struct pt_camera* camera, double up[3])
{
  ASSERT(camera && up);
  d3_set(up, camera->up);
}
```

3.4.3 Générer un rayon caméra

L'algorithme de rendu que nous souhaitons mettre en œuvre est un algorithme de suivi de chemins partant de la caméra et remontant vers les sources ; dit autrement nous traçons des chemins lumineux à rebours. Ces chemins partent donc de la caméra avec pour origine le sommet de la pyramide de vue, et vise la base de celle ci. La fonction suivante se charge de générer ce type de rayons.

38a ⟨pt_camera.h *functions declarations* 33b⟩+≡
```
extern LOCAL_SYM void
pt_camera_ray
  (const struct pt_camera* camera,
   const double sample[2],
   double ray_org[3],
   double ray_dir[3]);
```

Cette fonction prend en entrée 2 arguments : la caméra que nous souhaitons utiliser et une position 2D sur l'image. Les coordonnées de cette position sont comprises entre $[0, 1)^2$; le point $\{0, 0\}$ correspondant au coin supérieur gauche de l'image. En sortie de fonction l'utilisateur obtient l'origine du rayon généré ainsi que sa direction normalisée. La mise en oeuvre de cette fonction est la suivante :

38b ⟨pt_camera.c *functions definitions* 33c⟩+≡
```
void
pt_camera_ray
  (const struct pt_camera* camera,
   const double sample[2],
   double ray_org[3],
   double ray_dir[3])
{
  ⟨pt_camera_ray local variables 38d⟩
  ⟨check pt_camera_ray pre-conditions 38c⟩
  ⟨scale and bias sample in [−1, 1)^2 38e⟩
  ⟨transform the adjusted sample in a world space direction 39b⟩
  ⟨normalize the ray direction and setup its origin 39c⟩
}
```

Nous nous assurons d'abord que les paramètres d'entrée vérifient les pré-conditions de la fonction.

38c ⟨check pt_camera_ray *pre-conditions* 38c⟩≡
```
ASSERT(camera && sample && ray_org && ray_dir);
ASSERT(sample[0] >= 0 && sample[0] < 1);
ASSERT(sample[1] >= 0 && sample[1] < 1);
```

Puis nous utilisons le repère de la caméra pour transformer la position sur l'image (paramètre `sample`) en une direction dans le repère absolu. Pour ce faire, nous commençons par transformer les coordonnées 2D de cette position dans un repère dont l'origine se situe au centre de l'image, et où les frontières inférieures et supérieures de l'image se trouve en -1 et 1 le long des axes X et Y.

38d ⟨pt_camera_ray *local variables* 38d⟩≡
```
double sample_adjusted[2];
```

38e ⟨scale and bias sample in $[-1, 1)^2$ 38e⟩≡
```
sample_adjusted[0] = sample[0] * 2 - 1;
sample_adjusted[1] = sample[1] * 2 - 1;
```

Nous pouvons alors transformer cette position en une direction dans le repère absolu en la transformant avec l'orientation de la caméra (équation 3.1). Nous rappelons que le repère de la caméra a été construit afin de rendre compte de l'angle d'ouverture de la caméra et du rapport d'aspect de l'image à rendre (section 3.4.2). Cette seule transformation suffit donc pour calculer la direction de notre rayon au regard de l'ensemble des paramètres de la caméra.

39a ⟨`pt_camera_ray` *local variables* 38d⟩+≡
```
double x[3], y[3];
```

39b ⟨*transform the adjusted sample in a world space direction* 39b⟩≡
```
d3_muld(x, camera->axis_x, sample_adjusted[0]);
d3_muld(y, camera->axis_y, sample_adjusted[1]);
d3_add(ray_dir, x, y);
d3_add(ray_dir, ray_dir, camera->axis_z);
```

Reste alors à normaliser la direction que l'on vient de calculer et à copier la position de la caméra comme origine du rayon.

39c ⟨*normalize the ray direction and setup its origin* 39c⟩≡
```
d3_normalize(ray_dir, ray_dir);
d3_set(ray_org, camera->position);
```

3.4.4 Gestion du compteur de références

Jusqu'alors nous n'avons déclaré que les fonctions d'interfaces autorisant l'utilisateur à incrémenter et décrémenter le compteur de référence d'une caméra (section 3.4.1). Cette section décrit leur mise en œuvre. Nous commençons par la fonction qui se charge d'incrémenter le compteur de référence. Ici, nous nous contentons d'appeler la fonction proposée par `RSys` pour incrémenter le compteur.

39d ⟨`pt_camera.c` *functions definitions* 33c⟩+≡
```
void
pt_camera_ref_get(struct pt_camera* camera)
{
  ASSERT(camera);
  ref_get(&camera->ref);
}
```

La fonction `pt_camera_ref_put` est sensiblement identique à la fonction précédente. La principale différence est que l'interface de programmation du compteur de référence attend un paramètre d'entrée supplémentaire, à savoir un pointeur de fonction qui sera invoqué pour libérer l'espace mémoire occupé par la caméra si son compteur de référence est nul.

39e ⟨`pt_camera.c` *functions definitions* 33c⟩+≡
```
void
pt_camera_ref_put(struct pt_camera* camera)
{
  ASSERT(camera);
  ref_put(&camera->ref, camera_release);
}
```

Nous passons donc à la fonction `ref_put` la fonction `camera_release` suivante :

39f ⟨`pt_camera.c` *helper functions* 39f⟩≡
```
static void
camera_release(ref_T* reference)
{
  ⟨camera_release local variables 40a⟩
  ASSERT(reference);
```

```
  ⟨retrieve the pointer toward the corresponding camera 40b⟩
  ⟨free the camera 40c⟩
}
```

On notera que cette fonction a comme seul paramètre d'entrée un pointeur vers le compteur de référence de notre variable caméra et non un pointeur vers la variable elle même. Nous rappelons que nous utilisons ici les compteurs de références tels qu'ils sont mis en œuvre dans la bibliothèque `RSys`, une mise en œuvre qui, par construction, se veut indépendante d'une quelconque structure de données. Le profil du pointeur de fonction attendu par la fonction `ref_put` ne peut donc faire mention qu'au seul type connu par l'interface de programmation des compteurs de référence, en l'occurrence un pointeur vers une variable de type `ref_T`.

Nous pouvons cependant retrouver un pointeur vers la caméra qui contient ce compteur de référence. Nous connaissons la représentation mémoire de la structure caméra et pouvons par conséquent déterminer le nombre d'octets entre le début de la structure et ce compteur. En soustrayant ce nombre d'octets ou pointeur passé en entrée de la fonction nous obtenons alors un pointeur vers la caméra qui lui est associée. Pour réaliser cette opération, nous utilisons le macro `CONTAINER_OF` de la bibliothèque `RSys`, qui prend comme paramètre d'entrée le pointeur considéré, la structure vers laquelle on souhaite pointer, et le nom du membre dans cette structure auquel le pointeur fait référence.

40a ⟨`camera_release` *local variables* 40a⟩≡

```
struct pt_camera* camera;
```

40b ⟨*retrieve the pointer toward the corresponding camera* 40b⟩≡

```
camera = CONTAINER_OF(reference, struct pt_camera, ref);
```

Nous pouvons alors libérer l'espace mémoire occupé par la caméra en utilisant l'allocateur employé précédemment pour l'allouer (section 3.4.1).

40c ⟨*free the camera* 40c⟩≡

```
MEM_RM(&camera->pt->allocator, camera);
```

3.5 Image

Nous développons ici l'interface de programmation pour gérer l'espace mémoire de l'image que nous allons rendre. Une image est un tableau à deux dimensions dont la taille en X et en Y est contrôlée par la *définition* de l'image. Dans notre programme, chaque élément de l'image (appelé *pixel*) stocke une estimation de la luminance captée par l'angle solide défini par le point de vue de la caméra et le pixel considéré.

Dans le fichier d'en-tête `pt_image.h` nous déclarons cette interface de programmation que nous mettons en œuvre dans un fichier `C` séparé.

40d ⟨*src/pt_image.h* 40d⟩≡

```
⟨licensing 83a⟩

#ifndef PT_IMAGE_H
#define PT_IMAGE_H

⟨pt_image.h inclusions 41e⟩
⟨pt_image.h data types 41c⟩
⟨pt_image.h functions declarations 41g⟩

#endif /* PT_IMAGE_H */
```

41a ⟨*src/pt_image.c* 41a⟩≡
⟨*licensing* 83a⟩

```
#include "pt_image.h"
```
⟨pt_image.c *inclusions* 42b⟩
⟨pt_image.c *data types* 41d⟩
⟨pt_image.c *helper functions* 45c⟩
⟨pt_image.c *functions definitions* 42a⟩

41b ⟨*list of source files* 46c⟩≡
```
pt_image.h
pt_image.c
```

3.5.1 Créer une image

Tout comme pour une caméra (section 3.4.1), nous choisissons de n'autoriser la manipulation d'une image qu'à travers ses fonctions d'interfaces. Nous effectuons donc une déclaration avancée de son type structuré que nous définissons dans son fichier source.

41c ⟨pt_image.h *data types* 41c⟩≡
```
struct pt_image;
```

41d ⟨pt_image.c *data types* 41d⟩≡
```
struct pt_image {
```
⟨*image memory layout members* 43f⟩
⟨*image miscellaneous members* 42e⟩
```
};
```

Là aussi, tout comme pour la caméra, nous gérons la durée de vie d'une image par un compteur de références. Nous déclarons donc non seulement la fonction de création d'une image mais aussi les deux fonctions permettant d'incrémenter et de décrémenter son compteur de références.

41e ⟨pt_image.h *inclusions* 41e⟩≡
```
#include <rsys/rsys.h> /* Define the LOCAL_SYM macro and the res_T type */
```

41f ⟨pt_image.h *data types* 41c⟩+≡
```
struct pt; /* Forward declaration */
```

41g ⟨pt_image.h *functions declarations* 41g⟩≡
```
extern LOCAL_SYM res_T
pt_image_create
  (struct pt* pt,
   const size_t width,
   const size_t height,
   struct pt_image** image);

extern LOCAL_SYM void
pt_image_ref_get
  (struct pt_image* image);

extern LOCAL_SYM void
pt_image_ref_put
  (struct pt_image* image);
```

Le profil des fonctions qui gèrent le compteur de références d'une image suit, sans surprise, la même logique que leur pendant pour une caméra. La fonction de création, quant à elle, prend comme paramètres d'entrée un pointeur vers la variable représentant notre application (`pt`) et deux entiers supplémentaires caractérisant la définition de l'image (`width` et `height`). En dernier argument, nous lui passons l'adresse du pointeur que l'appelant souhaite utiliser pour référencer l'image ainsi créée. Cette fonction retourne enfin son état d'exécution à l'aide d'une variable de type `res_T`.

La définition de cette fonction est développée ci-après :

42a ⟨pt_image.c *functions definitions* 42a⟩≡
```
res_T
pt_image_create
  (struct pt* pt,
  const size_t width,
  const size_t height,
  struct pt_image** image)
{
  ⟨pt_image_create local variables 43a⟩
  res_T res = RES_OK;

  ⟨check the pt_image_create input parameters 42c⟩
  ⟨create the image variable 43b⟩
  ⟨setup the image buffer 44a⟩

exit:
  ⟨setup the pt_image_create output 43c⟩
  return res;
error:
  ⟨rollback the pt_image_create process 43d⟩
  goto exit;
}
```

Nous commençons par vérifier que les paramètres en entrée de la fonction sont valides (pointeurs non nuls, définition différente de zéro) avant, dans le cas contraire, d'afficher un message d'erreur dans le journal d'évènements de l'application et de retourner une erreur.

42b ⟨pt_image.c *inclusions* 42b⟩≡
```
#include "pt.h"
```

42c ⟨*check the* pt_image_create *input parameters* 42c⟩≡
```
if(!pt || !image) {
  logger_print(&pt->logger, LOG_ERROR,
    "%s: invalid NULL argument\n", FUNC_NAME);
  res = RES_BAD_ARG;
  goto error;
}
if(!width || !height) {
  logger_print(&pt->logger, LOG_ERROR,
    "%s: invalid image définition '%lux%lu'\n",
    FUNC_NAME, (unsigned long)width, (unsigned long)height);
  res = RES_BAD_ARG;
  goto error;
}
```

Nous allouons alors l'espace mémoire de la variable représentant l'image à créer que nous référençons à l'aide de la variable locale `img`. Comme à l'accoutumée, nous vérifions que l'espace mémoire a bien été alloué et retournons une erreur si ce n'est pas le cas. Une fois allouée, nous initialisons le compteur de référence de l'image à 1 et stockons un pointeur vers l'application afin notamment de conserver un accès à son allocateur que nous devrons utiliser pour libérer cet espace mémoire.

42d ⟨pt_image.c *inclusions* 42b⟩+≡
```
#include <rsys/mem_allocator.h>
#include <rsys/ref_count.h>
```

42e ⟨*image miscellaneous members* 42e⟩≡
```
struct pt* pt;
ref_T ref;
```

43a ⟨pt_image_create *local variables* 43a⟩≡
```
struct pt_image* img = NULL;
```

43b ⟨*create the image variable* 43b⟩≡
```
img = MEM_CALLOC(&pt->allocator, 1, sizeof(*img));
if(!img) {
  logger_print(&pt->logger, LOG_ERROR,
    "%s: could not allocate the image data structure\n", FUNC_NAME);
  res = RES_MEM_ERR;
  goto error;
}
img->pt = pt;
ref_init(&img->ref);
```

En sortie de fonction nous affectons l'espace alloué à l'adresse pointée par la variable `image`. Et en cas d'erreur nous libérons cet espace mémoire en décrémentant son compteur de référence.

43c ⟨*setup the* `pt_image_create` *output* 43c⟩≡
```
if(image) *image = img;
```

43d ⟨*rollback the* `pt_image_create` *process* 43d⟩≡
```
if(img != NULL) {
  pt_image_ref_put(img);
  img = NULL;
}
```

3.5.2 Création du tampon de l'image

En chaque pixel de l'image nous stockons la somme des poids Monte-Carlo et la somme de ces poids élevés au carrés, utilisés pour estimer la luminance du pixel. Nous stockons également le nombre de poids effectivement accumulés dans ces deux sommes. Ces données nous permettront de calculer l'espérance de la luminance au pixel, sa variance et son écart type. Nous ajoutons la type `struct pt_pixel` comme un type de l'interface de programmation d'une image ; l'utilisateur devant être en mesure de lire et d'écrire un pixel de l'image, la représentation mémoire d'un pixel doit être connue de celui-ci. Nous définissons également la constante `PT_PIXEL_NULL` qui représente la valeur du pixel avant tout traitement.

43e ⟨`pt_image.h` *data types* 41c⟩+≡
```
struct pt_pixel {
  double sum; /* Sum of Monte-Carlo weights */
  double sum2; /* Sum of squared Monte-Carlo weights */
  size_t nweights; /* Numober of accumulated weights */
};
static const struct pt_pixel PT_PIXEL_NULL = {0, 0, 0};
```

Nous choisissons d'organiser les pixels de l'image ligne par ligne dans un tampon continu en mémoire ; soit N la définition de l'image en X, les N premiers pixels correspondent aux N pixels de la première ligne, les N suivant à ceux de la seconde ligne et ainsi de suite. Les lignes de l'image sont quant à elles stockées de haut en bas, la première ligne correspond donc à la ligne supérieur de l'image. À la création de l'image, nous allouons donc ce tampon mémoire en fonction de la définition de l'image et de la taille mémoire d'un pixel. Nous vérifions, comme toujours, que l'allocation a bien eu lieu avant de retourner une erreur mémoire dans le cas contraire.

43f ⟨*image memory layout members* 43f⟩≡
```
struct pt_pixel* pixels;
```

44a ⟨*setup the image buffer* 44a⟩≡
```
img->pixels = MEM_CALLOC(&pt->allocator, width*height, sizeof(*img->pixels));
if(!img->pixels) {
  logger_print(&pt->logger, LOG_ERROR,
    "%s: could not allocate the image buffer\n", FUNC_NAME);
  res = RES_MEM_ERR;
  goto error;
}
```

Nous stockons enfin la définition de l'image que nous créons et ajoutons deux fonctions permettant à l'appelant d'interroger cette définition.

44b ⟨*image memory layout members* 43f⟩+≡
```
size_t width;
size_t height;
```

44c ⟨*setup the image buffer* 44a⟩+≡
```
img->width = width;
img->height = height;
```

44d ⟨pt_image.h *functions declarations* 41g⟩+≡
```
extern LOCAL_SYM size_t
pt_image_get_width
  (const struct pt_image* image);

extern LOCAL_SYM size_t
pt_image_get_height
  (const struct pt_image* image);
```

44e ⟨pt_image.c *functions definitions* 42a⟩+≡
```
size_t
pt_image_get_width(const struct pt_image* image)
{
  ASSERT(image);
  return image->width;
}

size_t
pt_image_get_height(const struct pt_image* image)
{
  ASSERT(image);
  return image->height;
}
```

3.5.3 Accéder à un pixel de l'image

Dans cette section nous développons la fonction d'interface permettant à l'appelant d'accéder à un pixel de l'image pour qu'il soit en mesure de lire ou d'écrire son contenu. Cette fonction prend comme paramètres d'entrée l'image considérée ainsi que les indices du pixel en X et en Y. En sortie, elle retourne un pointeur vers le pixel ainsi indexé.

44f ⟨pt_image.h *functions declarations* 41g⟩+≡
```
extern LOCAL_SYM struct pt_pixel*
pt_image_at
  (struct pt_image* image,
   const size_t x,
   const size_t y);
```

Cette fonction donne à l'utilisateur un accès aux pixels de l'image tout en lui masquant son organisation mémoire. La mise en œuvre de cette fonction consiste uniquement à évaluer le nombre de pixels entre le début du tampon de l'image et le pixel auquel on souhaite accéder, avant d'indexer le tampon de l'image avec cet écart. Au préalable nous nous assurons que les indices du pixel sont valides au regard de la définition de l'image à laquelle on souhaite accéder.

45a ⟨pt_image.c *functions definitions* 42a⟩+≡
```
struct pt_pixel*
pt_image_at(struct pt_image* image, const size_t x,  const size_t y)
{
  ASSERT(image && x < image->width && y < image->height);
  return &image->pixels[y*image->width + x];
}
```

3.5.4 Libérer l'espace mémoire d'une image

La durée de vie d'une image est gérée à l'aide d'un compteur de références. La mise en oeuvre des fonctions incrémentant et décrémentant ce compteur suit la même logique que leur mise en oeuvre dans le cas d'une caméra. Nous invitons donc le lecteur à se référer à la section 3.4.4 pour plus de détails sur leur code.

45b ⟨pt_image.c *functions definitions* 42a⟩+≡
```
void
pt_image_ref_get(struct pt_image* image)
{
  ASSERT(image);
  ref_get(&image->ref);
}

void
pt_image_ref_put(struct pt_image* image)
{
  ASSERT(image);
  ref_put(&image->ref, image_release);
}
```

La seule différence notable dans le code ci-dessus concerne la fonction à appeler pour libérer l'espace mémoire de l'image lorsque son compteur de référence atteint zéro. Ici, nous utilisons la fonction `image_release` suivante :

45c ⟨pt_image.c *helper functions* 45c⟩≡
```
static void
image_release(ref_T* reference)
{
  struct pt_image* image;
  ASSERT(reference);
  image = CONTAINER_OF(reference, struct pt_image, ref);
  if(image->pixels) MEM_RM(&image->pt->allocator, image->pixels);
  MEM_RM(&image->pt->allocator, image);
}
```

Dans cette fonction, nous utilisons tout d'abord la macro `CONTAINER_OF` pour retrouver l'adresse de l'image à laquelle la référence passée en entrée correspond (section 3.4.4). Nous libérons ensuite l'espace mémoire du tampon de l'image si ce dernier est effectivement alloué avant de libérer l'espace mémoire de l'image en tant que telle.

3.6 L'application

Dans notre programme, l'application est représentée par un type structuré listant les variables communes à une même exécution, comme l'allocateur mémoire utilisé pour allouer dynamiquement des blocs mémoires ou la scène que nous souhaitons rendre. Selon toute évidence, notre programme contiendra donc qu'*une seule* variable de type "application" définissant une configuration et un état particulier d'exécution. Cette variable sera initialisée dès le lancement du programme, se basant sur les arguments passés à la ligne de commande pour se configurer, et sera libérée à sa toute fin. En définitive, la variable "application" matérialise le programme et agrège en ses membres un état et une configuration particulière qui en son absence auraient vraisemblablement pris la forme de variables globales au programme.

Dans cette section nous développons l'interface de programmation de l'application. Cette interface décrit le type structuré la représentant et les fonctions s'y rapportant. Comme à l'accoutumée, les types et fonctions directement accessibles à l'appelant sont listés dans un fichier d'en-tête et leur mise en œuvre est définie dans un fichier source séparé.

46a ⟨*src/pt.h* 46a⟩≡
```
⟨licensing 83a⟩

#ifndef PT_H
#define PT_H

⟨pt.h inclusions 47a⟩
⟨pt.h forward declarations 47b⟩
⟨pt.h data types 46d⟩
⟨pt.h functions declarations 47c⟩

#endif /* PT_H */
```

46b ⟨*src/pt.c* 46b⟩≡
```
⟨licensing 83a⟩

⟨pt.c features definitions 53b⟩

#include "pt.h"
⟨pt.c inclusions 47d⟩
⟨pt.c helper functions declarations 58a⟩
⟨pt.c helper functions definitions 49c⟩
⟨pt.c functions definitions 47e⟩
```

46c ⟨*list of source files* 46c⟩≡
```
pt.h
pt.c
```

3.6.1 Structure de données et fonction d'initialisation

Le type structuré représentant l'application est directement défini dans le fichier d'en-tête. Par là même nous rendons l'ensemble de ses variables accessibles à l'appelant. Nous nommons ce type structuré `pt`, acronyme de *Path-Tracer*, qui est en l'occurrence le préfixe utilisé par les types et les fonctions des interfaces de programmation de l'ensemble du programme.

46d ⟨`pt.h` *data types* 46d⟩≡
```
struct pt {
  ⟨struct pt members 48d⟩
};
```

Pour configurer notre application, nous utilisons les arguments de la ligne de commande. Nous passons donc en entrée de sa fonction d'initialisation un pointeur vers les arguments de la ligne de commande matérialisés par une variable de type `struct pt_args` (section 3.3).

47a ⟨`pt.h` *inclusions* 47a⟩≡
```
#include <rsys/rsys.h>
```

47b ⟨`pt.h` *forward declarations* 47b⟩≡
```
struct pt_args;
```

47c ⟨`pt.h` *functions declarations* 47c⟩≡
```
extern LOCAL_SYM res_T
pt_init
  (const struct pt_args* args,
   struct pt* pt);
```

Le type structuré de la variable `args` est défini dans un fichier d'en-tête séparé que nous devrions vraisemblablement inclure pour pouvoir l'utiliser. Cependant, une bonne pratique de programmation consiste à éviter, autant que faire se peut, d'inclure des fichiers d'en-tête dans d'autres fichiers d'en-tête ; la raison principale étant liée au temps de compilation. En limitant le nombre de fichiers inclus, d'aucun limite le temps passé par le pré-processeur à traiter ces différentes inclusions. Et ce gain peut être d'autant plus significatif pour un fichier d'en-tête qui a lui même vocation à être inclus dans d'autres fichiers. Or, dans le profil de la fonction que nous souhaitons déclarer, nous n'utilisons pas directement le type `struc pt_args` mais un pointeur vers ce type ; nous n'avons donc pas besoin d'en connaître la représentation mémoire. Par conséquent, nous pouvons effectuer une déclaration avancée de ce type à la place d'inclure le fichier d'en-tête le définissant, ce qui signifie au compilateur que ce type existe mais que sa représentation mémoire est définie ailleurs.

Dans la mise en œuvre de la fonction d'initialisation, nous nous appuyons sur le contenu de la variable `args` pour configurer l'ensemble de l'application. Ici, nous devons donc connaître la représentation mémoire de cette variable pour pouvoir accéder à ses membres, une représentation mémoire définie dans le fichier d'en-tête `pt_args.h` que nous incluons alors dans le fichier `C`.

47d ⟨`pt.c` *inclusions* 47d⟩≡
```
#include "pt_args.h"
```

47e ⟨`pt.c` *functions definitions* 47e⟩≡
```
res_T
pt_init(const struct pt_args* args, struct pt* pt)
{
  ⟨pt_init local variables 50e⟩
  res_T res = RES_OK;

  *pt = PT_NULL; /* Default state of the pt variable */

  ⟨setup the allocator 48e⟩
  ⟨setup the logger 49b⟩
  ⟨setup the camera 50f⟩
  ⟨setup the image 51d⟩
  ⟨setup the output stream 52a⟩
  ⟨setup the scene geometry 54c⟩
  ⟨setup the scene material 59g⟩

  ⟨release pt_init temporary data 55i⟩

exit:
  return res;
error:
```

```
    ⟨release pt_init temporary data 55i⟩
    ⟨rollback the pt_init process 48b⟩
    goto exit;
  }
```

En cas d'erreur, nous appelons la fonction `pt_release` (section 3.6.11) pour annuler les traitements effectués jusqu'alors et ce *après* avoir relâché les variables locales qui le nécessitent. Ces variables peuvent en effet s'appuyer sur des membres de l'application que l'on cherche justement à invalider. Nettoyer l'application avant de libérer ces variables locales pourrait invalider des membres de l'application justement utilisés pour libérer ces variables.

48a ⟨pt.h *functions declarations* 47c⟩+≡

```
extern LOCAL_SYM void
pt_release
  (struct pt* pt);
```

48b ⟨*rollback the* pt_init *process* 48b⟩≡

```
pt_release(pt);
```

3.6.2 Configuration de l'allocateur mémoire

Pour initialiser notre application, nous commençons par configurer l'allocateur que nous souhaitons utiliser. Nous ajoutons donc un allocateur comme variable membre de l'application. Le type de cette variable est défini par la bibliothèque `RSys` dans le fichier d'en-tête `rsys/mem_allocator.h` que nous incluons.

48c ⟨pt.h *inclusions* 47a⟩+≡

```
#include <rsys/mem_allocator.h>
```

48d ⟨struct pt *members* 48d⟩≡

```
struct mem_allocator allocator;
```

`RSys` propose plusieurs types d'allocateurs mémoire, chacun d'eux étant caractérisé par sa politique d'allocation et la manière dont il enregistre les allocations. Ici, nous ne souhaitons pas définir de politique d'allocation particulière : nous utilisons donc l'allocateur par défaut, semblable dans son comportement aux fonctions d'allocation proposées par la bibliothèque `C` standard. Cet allocateur nous est donc essentiellement utile pour enregistrer les allocations que nous effectuons et pouvoir vérifier ainsi qu'aucune fuite mémoire n'est à déplorer à l'échelle de l'allocateur.

48e ⟨*setup the allocator* 48e⟩≡

```
res = mem_init_regular_allocator(&pt->allocator);
if(res != RES_OK) {
  fprintf(stderr, "%s: could not initialise the memory allocator\n", FUNC_NAME);
  goto error;
}
```

À noter qu'en cas d'erreur lors de l'initialisation de l'allocateur, nous affichons un message d'erreur directement sur la sortie standard des erreurs à la place d'utiliser le journal d'évènements de l'application, ce dernier n'étant toujours pas configuré.

3.6.3 Configuration du journal d'évènements

Dans cette partie nous configurons le journal d'évènements. Nous commençons par ajouter un journal d'évènements comme variable membre de l'application. Nous utilisons ici un journal d'évènements tel qu'il est défini par la bibliothèque `RSys`.

48f ⟨pt.h *inclusions* 47a⟩+≡

```
#include <rsys/logger.h>
```

49a ⟨`struct pt` *members* 48d⟩+≡
```
struct logger logger;
```

La configuration de ce journal consiste à l'initialiser avant de lui associer une fonction d'écriture pour chaque type de message qu'il propose.

49b ⟨*setup the logger* 49b⟩≡
```
res = logger_init(&pt->allocator, &pt->logger);
if(res != RES_OK) {
  fprintf(stderr, "%s: could not initialise the logger\n", FUNC_NAME);
  goto error;
}
```
⟨*setup the logger streams* 50a⟩

Le journal d'évènements mis en œuvre dans la bibliothèque `RSys` propose trois type de messages : les messages d'information, les messages d'erreur, et les messages d'avertissement. Les fonctions utilisées pour écrire chaque type de message sont laissées à la charge de l'appelant qui peut dès lors décider ce qu'il en fait : il peut sauvegarder ces messages dans des fichiers, les envoyer sur le réseau, les afficher sur les sorties standards ou encore simplement les ignorer. Ici, nous décidons d'afficher ces messages sur les sorties standards. Pour cela nous définissons trois fonctions intermédiaires, une par type de message, qui affichent le message envoyé au journal d'évènements sur une sortie standard. Le premier paramètre d'entrée de ces fonctions est le message à afficher et le second paramètre est un pointeur vers des données utilisateur. Ici nous n'utilisons pas de données utilisateur particulières et nous ignorons donc simplement ce dernier paramètre.

49c ⟨`pt.c` *helper functions definitions* 49c⟩≡
```
static void
print_out(const char* msg, void* ctx)
{
  (void)ctx; /* Avoid the "unused variable" warning */
  fprintf(stderr, "\x1b[1m\x1b[32m>\x1b[0m %s", msg);
}

static void
print_err(const char* msg, void* ctx)
{
  (void)ctx; /* Avoid the "unused variable" warning */
  fprintf(stderr, "\x1b[31merror:\x1b[0m %s", msg);
}

static void
print_warn(const char* msg, void* ctx)
{
  (void)ctx; /* Avoid the "unused variable" warning */
  fprintf(stderr, "\x1b[33mwarning:\x1b[0m %s", msg);
}
```

Dans les fonctions que nous venons de définir, nous affichons les messages envoyés au journal d'évènements précédés d'un texte nous aidant à identifier visuellement le type de message affiché. Pour les messages d'information, nous écrivons le caractère '>' en vert et en gras. Les messages d'erreur et d'avertissement sont quant à eux précédés des textes "error :" et "warning :" affichés, respectivement, en rouge et en jaune. Pour contrôler le style et la couleur de ces préfixes nous utilisons les caractères d'échappement ANSI directement dans le message à afficher. Enfin, on notera qu'indépendamment du type de message, nous l'affichons quoiqu'il en soit sur la sortie standard des erreurs, là où on pourrait au moins attendre l'utilisation de la sortie standard pour les messages d'information. Or, nous utilisons déjà cette sortie standard comme "fichier" de sortie possible aux données de notre programme (section 3.6.6). En écrivant l'ensemble

des messages sur la sortie standard des erreurs, nous empêchons ainsi de sauvegarder ces messages dans le fichier de sortie, ce qui aurait pour conséquence d'en corrompre le format.

Nous utilisons enfin l'interface de programmation du journal d'évènements pour définir ces fonctions comme fonctions d'écriture des différents types de messages :

50a ⟨*setup the logger streams* 50a⟩≡
```
logger_set_stream(&pt->logger, LOG_OUTPUT, print_out, NULL);
logger_set_stream(&pt->logger, LOG_ERROR, print_err, NULL);
logger_set_stream(&pt->logger, LOG_WARNING, print_warn, NULL);
```

3.6.4 Configuration de la caméra

Nous créons désormais la caméra utilisée pour le rendu. Nous nous contentons ici d'appeler la fonction de création d'une caméra (section 3.4.1) en se basant sur les arguments de la ligne de commande pour la paramétrer.

Pour cela, nous ajoutons tout d'abord une variable caméra à la structure `pt`, variable qui représentera la caméra utilisée pour le rendu. Cette variable est de type opaque, c'est à dire d'un type dont la représentation mémoire est occultée à l'appelant. Ce dernier ne peut donc pas définir une variable de ce type directement sur la pile et doit utiliser les fonctions proposées par son interface de programmation pour allouer dynamiquement et initialiser son espace mémoire. C'est pourquoi la variable caméra est un *pointeur* vers un type structuré et non directement une variable de ce type. Nous pouvons donc effectuer une déclaration avancée du type `struct pt_camera` et ainsi pouvoir ajouter une variable membre qui pointe vers une variable de ce type.

50b ⟨`pt.h` *forward declarations* 47b⟩+≡
```
struct pt_camera;
```

50c ⟨`struct pt` *members* 48d⟩+≡
```
struct pt_camera* camera;
```

Dans la fonction d'initialisation, nous créons et configurons notre caméra en se basant directement sur la plupart des arguments de la ligne de commande matérialisant ladite caméra. Seul son angle d'ouverture et le rapport d'aspect de son image nécessitent un traitement particulier. Pour l'angle d'ouverture, nous le renseignons en degrés sur la ligne de commande là où la caméra attend des radians. Nous utilisons donc la macro `MDEG2RAD` définit dans `rsys/math.h` pour le convertir en radians. Pour le rapport d'aspect du plan image, nous émettons l'hypothèse raisonnable que l'utilisateur souhaite avoir des pixels carrés et nous utilisons alors la définition de l'image pour calculer ce rapport.

50d ⟨`pt.c` *inclusions* 47d⟩+≡
```
#include "pt_camera.h"
```

50e ⟨`pt_init` *local variables* 50e⟩≡
```
double image_ratio;
```

50f ⟨*setup the camera* 50f⟩≡
```
image_ratio = (double)args->image.definition[0] / (double)args->image.definition[1];
res = pt_camera_create(pt, args->camera.pos, args->camera.tgt,
  args->camera.up, image_ratio, MDEG2RAD(args->camera.field_of_view),
  &pt->camera);
if(res != RES_OK) goto error;
```

Ici, en cas d'erreur, nous n'affichons pas de message particulier, la fonction `pt_camera_create` notifiant déjà à l'utilisateur le type d'erreur qu'elle a rencontré.

3.6.5 Configuration de l'image

Nous représentons l'image à rendre à l'aide d'une variable de type `struct pt_image` dont l'interface de programmation est développée dans la section 3.5. Nous commençons par ajouter une image comme variable membre de notre application. Comme pour la variable caméra, le type de cette variable est opaque et nous ne déclarons donc pas l'image comme une variable de ce type mais comme un pointeur vers une variable de ce type.

51a ⟨`pt.h` *forward declarations* 47b⟩+≡
```
struct pt_image;
```

51b ⟨`struct pt` *members* 48d⟩+≡
```
struct pt_image* image;
```

Dans la fonction d'initialisation, nous créons alors simplement cette image en utilisant la résolution définie par les arguments de la ligne de commande.

51c ⟨`pt.c` *inclusions* 47d⟩+≡
```
#include "pt_image.h"
```

51d ⟨*setup the image* 51d⟩≡
```
res = pt_image_create(pt, args->image.definition[0], args->image.definition[1],
  &pt->image);
if(res != RES_OK) goto error;
```

Là aussi nous n'affichons pas de message d'erreur en cas de problème, la fonction de création de l'image affichant déjà le type d'erreur rencontré dans le journal d'évènements de l'application.

Reste alors à initialiser le nombre de réalisations par pixel souhaité par l'utilisateur. Dans les arguments de la ligne de commande, ce paramètre est une variable de l'image alors que l'interface de programmation d'une image ne représente que l'organisation mémoire de ses pixels. Nous ajoutons donc à la structure de l'application une nouvelle variable membre qui stocke ce nombre de réalisations passé en argument.

51e ⟨`struct pt` *members* 48d⟩+≡
```
size_t spp; /* # Samples per pixel */
```

51f ⟨*setup the image* 51d⟩+≡
```
pt->spp = args->image.spp;
```

3.6.6 Configuration du fichier en sortie

Nous allons désormais configurer le fichier dans lequel les données de l'image calculée seront stockées. Pour ce faire, nous ajoutons à la liste des membres de la structure "application" une variable qui pointe vers le fichier en sortie. On rappelle qu'en `C`, un fichier s'apparente aussi bien à un fichier disque qu'à une zone mémoire ou une sortie standard.

51g ⟨`pt.h` *inclusions* 47a⟩+≡
```
#include <stdio.h>
```

51h ⟨`struct pt` *members* 48d⟩+≡
```
FILE* output;
```

L'utilisateur du programme peut décider d'écrire l'image rendue directement sur la sortie standard ou dans un fichier sur disque. Ce choix est ici déterminé par la valeur du pointeur `output_filename` de la variable `args`. Un pointeur `NULL` signifie que l'utilisateur n'a pas renseigné le nom du fichier en sortie : l'image sera alors écrite sur la sortie standard. Dans le cas contraire, la chaîne de caractères pointée par `output_filename` est utilisée comme nom du fichier en sortie.

52a ⟨*setup the output stream* 52a⟩≡
```
if(!args->output_filename) {
  pt->output = stdout;
} else {
  ⟨open the output file 52b⟩
}
```

Nous avons vu lors de l'analyse des arguments (section 3.3.7) que nous interdisons à priori l'écriture dans un fichier déjà présent sur disque, et ce afin d'éviter d'écraser par erreur son contenu. Pour forcer l'écriture d'un tel fichier, l'utilisateur doit l'expliciter sur la ligne de commande à l'aide de l'option '-f' qui une fois analysée, définit le membre `force_overwriting` de la variable `args` à 1. L'ouverture du fichier en sortie est donc conditionnée par sa présence sur le disque et la valeur de cette variable membre. Ouvrir ce fichier ne se limite donc pas à un simple appel à la fonction `fopen`. Nous décidons par conséquent de mettre en œuvre cette ouverture conditionnée dans la fonction décrite ci-après.

52b ⟨*open the output file* 52b⟩≡
```
res = open_output_stream
  (pt, args->output_filename, args->force_overwriting, &pt->output);
if(res != RES_OK) goto error;
```

52c ⟨`pt.c` *helper functions definitions* 49c⟩+≡
```
static res_T
open_output_stream
  (struct pt* pt,
   const char* filename,
   const int force_overwriting,
   FILE** out_fp)
{
  ⟨open_output_stream local variables 52d⟩
  res_T res = RES_OK;
  ASSERT(pt && filename && out_fp);

  if(force_overwriting) {
    ⟨open the output file even though it already exists 53a⟩
  } else {
    ⟨open the output file only if it does not exist 53e⟩
  }

exit:
  ⟨setup open_output_stream output 54a⟩
  return res;
error:
  ⟨rollback the open_output_stream process 54b⟩
  goto exit;
}
```

L'ouverture forcée du fichier reste le cas le plus simple. Nous ouvrons simplement le fichier en écriture avec la fonction `fopen` et vérifions que le pointeur de fichier retourné est valide avant de renvoyer une erreur dans le cas contraire

52d ⟨open_output_stream *local variables* 52d⟩≡
```
FILE* fp = NULL;
```

53a ⟨*open the output file even though it already exists* 53a⟩≡
```
fp = fopen(filename, "w");
if(!fp) {
  logger_print(&pt->logger, LOG_ERROR,
    "%s: could not open the output file ‘%s’\n", FUNC_NAME, filename);
  res = RES_IO_ERR; /* Notify an Input/Output error */
  goto error;
}
```

L'ouverture conditionnée du fichier demande plus d'attention. Une approche naïve consisterait d'abord à interroger la présence du fichier sur disque et en son absence d'ouvrir ensuite ce dernier comme dans le cas précédent. Or, rien n'assure qu'un fichier portant le même nom ne sera pas créé par un processus tiers entre l'étape de vérification et l'ouverture effective du fichier. Nous écraserions alors le contenu du fichier ainsi créé, ce que nous cherchons justement à éviter. Pour une mise en œuvre rigoureuse nous utilisons la fonction `open` de la bibliothèque `C` standard, plus fine dans sa paramétrisation que la fonction `fopen`, et qui permet justement d'opérer une ouverture conditionnée de fichier. Pour ce faire nous lui passons, entre autres, les options `O_CREAT` et `O_EXCL` qui demandent à la fonction de notifier le numéro d'erreur `EEXIST` si le fichier à ouvrir existe déjà. Nous soulignons que cette fonction retourne un *descripteur de fichier* ; nous utilisons alors la fonction `fdopen`, définie dans la norme POSIX 2001, pour convertir ce descripteur en pointeur de fichier, comme attendu en sortie de fonction.

53b ⟨`pt.c` *features definitions* 53b⟩≡
```
#define _POSIX_C_SOURCE 200112L /* Provide fdopen support */
```

53c ⟨`pt.c` *inclusions* 47d⟩+≡
```
#include <errno.h>
#include <fcntl.h> /* Declare the open function */
#include <unistd.h> /* Declare the close function */
#include <sys/stat.h> /* Define the S_IRUSR & S_IWUSR constants */
```

53d ⟨`open_output_stream` *local variables* 52d⟩+≡
```
int fd = -1; /* File descriptor */
```

53e ⟨*open the output file only if it does not exist* 53e⟩≡
```
fd = open(filename, O_CREAT|O_EXCL|O_TRUNC|O_WRONLY, S_IRUSR|S_IWUSR);
if(fd < 0) { /* An error occurs */
  if(errno == EEXIST) {
    logger_print(&pt->logger, LOG_ERROR,
      "%s: the output file ‘%s’ already exists\n", FUNC_NAME, filename);
    res = RES_IO_ERR;
    goto error;
  } else {
    logger_print(&pt->logger, LOG_ERROR,
      "%s: unexpected error while opening the output file ‘%s’\n",
      FUNC_NAME, filename);
    res = RES_IO_ERR;
    goto error;
  }
}

fp = fdopen(fd, "w");
if(!fp) {
  logger_print(&pt->logger, LOG_ERROR,
    "%s: could not open the output file ‘%s’\n", FUNC_NAME, filename);
  res = RES_IO_ERR;
  goto error;
}
```

Il ne nous reste donc plus qu'à affecter la valeur du pointeur de fichier local à l'adresse pointée par la variable en sortie, et à écrire le traitement à effectuer en cas d'erreur, à savoir fermer le pointeur ou descripteur de fichier éventuellement ouvert dans cette fonction.

54a ⟨*setup* `open_output_stream` *output* 54a⟩≡
```
*out_fp = fp;
```

54b ⟨*rollback the* `open_output_stream` *process* 54b⟩≡
```
if(fp) {
  fclose(fp);
  fp = NULL;
} else {
  close(fd);
}
```

3.6.7 Chargement et configuration de la géométrie

Nous choisissons de décrire la géométrie de la scène à l'aide du format de fichier OBJ. Dans cette partie, nous présentons la procédure qui charge les données géométriques à partir d'un tel fichier et les structures en mémoire pour l'étape de rendu.

54c ⟨*setup the scene geometry* 54c⟩≡
⟨*setup* `Star-3D` 55b⟩
⟨*load the geometry data from the OBJ file* 55h⟩
⟨*create a* `Star-3D` *scene* 56c⟩
⟨*register the loaded geometries against the* `Star-3D` *scene* 56f⟩
⟨*create the* `Star-3D` *scene view for the ray-tracing* 57c⟩

`Star-3D`

La mise en données que nous souhaitons opérer doit permettre l'intersection efficace d'un rayon avec la géométrie de la scène. En effet, l'algorithme de rendu que nous allons mettre en oeuvre s'appuie sur le lancer de rayons pour suivre la trajectoire des photons dans la scène. Au sein de l'environnement `Star-Engine`, la bibliothèque `Star-3D` offre notamment cette fonctionnalité : elle partionne la géométrie dans l'espace afin d'accélérer la traversée des rayons dans la scène et tester leur intersection uniquement avec les géométries les avoisinant. Nous utilisons donc la bibliothèque `Star-3D` pour représenter en mémoire la géométrie de notre scène.

54d ⟨*check for* `CMake` *packages* 54d⟩≡
```
find_package(Star3D 0.6 REQUIRED)
```

54e ⟨*list of paths where to look for the dependency headers* 54e⟩≡
```
${Star3D_INCLUDE_DIR}
```

54f ⟨*list of used libraries* 54f⟩≡
```
Star3D
```

Nous ajoutons à notre application une variable membre qui pointe vers l'instance de la bibliothèque `Star-3D` que nous utilisons. Comme d'habitude, nous effectuons une déclaration avancée du type pointé afin d'éviter d'inclure le fichier d'en-tête de la bibliothèque `Star-3D` dans le fichier courant.

54g ⟨`pt.h` *forward declarations* 47b⟩+≡
```
struct s3d_device;
```

54h ⟨`struct pt` *members* 48d⟩+≡
```
struct s3d_device* s3d;
```

Dans la fonction d'initialisation de l'application, nous utilisons l'interface de programmation de `Star-3D` pour créer et initialiser l'espace mémoire pointé par cette variable membre. À noter que nous passons à la fonction de création de `Star-3D`, un pointeur vers le journal d'évènements de l'application pour que ce dernier soit utilisé par `Star-3D` pour reporter ses messages. De même, nous passons un pointeur vers l'allocateur de notre application afin que les allocations mémoires effectuées dans `Star-3D` utilisent ledit allocateur. Celui ci pourra par conséquent détecter les fuites mémoires non seulement dans notre programme mais aussi au sein de `Star-3D`.

55a ⟨`pt.c` *inclusions* 47d⟩+≡
```
#include <star/s3d.h>
```

55b ⟨*setup* `Star-3D` 55b⟩≡
```
res = s3d_device_create
  (&pt->logger, &pt->allocator, 0/*verbosity level*/, &pt->s3d);
if(res != RES_OK) {
  logger_print(&pt->logger, LOG_ERROR,
    "%s: coult nod create the Star-3D device\n", FUNC_NAME);
  goto error;
}
```

Chargement de l'OBJ

Pour charger les données d'un fichier OBJ et les convertir en géométries `Star-3D`, nous utilisons la bibliothèque `Star-3DAW`, elle aussi distribuée avec le `Star-Engine`. Nous commençons par ajouter cette bibliothèque comme dépendance du projet.

55c ⟨*check for* `CMake` *packages* 54d⟩+≡
```
find_package(Star3DAW 0.3 REQUIRED)
```

55d ⟨*list of paths where to look for the dependency headers* 54e⟩+≡
```
${Star3DAW_INCLUDE_DIR}
```

55e ⟨*list of used libraries* 54f⟩+≡
```
Star3DAW
```

Nous pouvons maintenant utiliser `Star-3DAW` pour charger les données du fichier d'entrée. Dans la même logique que pour `Star-3D`, nous devons d'abord créer une variable qui incarne la bibliothèque `Star-3DAW` que nous utilisons. Cependant, nous n'avons pas besoin de définir cette variable en tant que variable membre de l'application, car le chargement des données géométriques ne concerne que l'étape d'initialisation. Nous nous contentons donc de définir cette variable comme variable locale à la fonction d'initialisation.

55f ⟨`pt.c` *inclusions* 47d⟩+≡
```
#include <star/s3daw.h>
```

55g ⟨`pt_init` *local variables* 50e⟩+≡
```
struct s3daw* s3daw = NULL;
```

55h ⟨*load the geometry data from the OBJ file* 55h⟩≡
```
res = s3daw_create(&pt->logger, &pt->allocator, NULL, NULL, pt->s3d,
  1/*verbosity level*/, &s3daw);
if(res != RES_OK) {
  logger_print(&pt->logger, LOG_ERROR,
    "%s: could not create the Star-3DAW device\n", FUNC_NAME);
  goto error;
}
```

55i ⟨*release* `pt_init` *temporary data* 55i⟩≡
```
if(s3daw) { s3daw_ref_put(s3daw); s3daw = NULL; }
```

Là aussi, tout comme pour Star-3D, nous passons à la fonction de création de Star-3DAW le journal d'évènements et l'allocateur de l'application pour que cette bibliothèque les utilise en lieu et place d'un journal d'évènements et d'un allocateur par défaut. Nous lui passons également le pointeur vers la bibliothèque Star-3D que nous venons de créer et qui sera utilisé par Star-3DAW pour créer les géométries Star-3D à partir des données de l'OBJ.

Il ne nous reste plus alors qu'à charger le fichier OBJ soumis en entrée du programme et convertir la géométrie qu'il décrit en géométrie Star-3D. On rappelle que l'utilisateur peut décider de ne pas renseigner de fichier OBJ à charger. Dans ce cas, la variable membre input_filename de la variable args est nulle et notre programme considère que les données de l'OBJ sont passées via l'entrée standard.

56a ⟨*load the geometry data from the OBJ file* 55h⟩+≡
```
if(args->input_filename) {
  res = s3daw_load(s3daw, args->input_filename);
} else {
  res = s3daw_load_stream(s3daw, stdin);
}
if(res != RES_OK) {
  logger_print(&pt->logger, LOG_ERROR,
    "%s: could not load the OBJ file ‘%s’\n",
     FUNC_NAME, args->input_filename ? args->input_filename : "stdin");
  goto error;
}
```

Définition de la scène

Nous créons désormais la scène Star-3D dans laquelle nous allons ajouter les géométries que nous venons de charger. Cette scène est une variable intermédiaire, donc locale à la fonction ; le lancer de rayons ne s'appuyant pas directement sur cette scène mais sur une *vue* de celle ci que nous définissons plus tard dans cette même section.

56b ⟨pt_init *local variables* 50e⟩+≡
```
struct s3d_scene* scn = NULL;
```

56c ⟨*create a* Star-3D *scene* 56c⟩≡
```
res = s3d_scene_create(pt->s3d, &scn);
if(res != RES_OK) {
  logger_print(&pt->logger, LOG_ERROR,
    "%s: could not create the Star-3D scene\n", FUNC_NAME);
  goto error;
}
```

56d ⟨*release* pt_init *temporary data* 55i⟩+≡
```
if(scn) { s3d_scene_ref_put(scn); scn = NULL; }
```

Nous ajoutons alors à cette scène les géométries chargées par Star-3DAW. Pour ce faire, nous interrogeons cette bibliothèque pour connaître le nombre de géométries Star-3D créées à partir de l'OBJ. Puis nous itérons sur ces dernières pour les attacher à la scène.

56e ⟨pt_init *local variables* 50e⟩+≡
```
size_t nshapes, ishape;
```

56f ⟨*register the loaded geometries against the* Star-3D *scene* 56f⟩≡
```
s3daw_get_shapes_count(s3daw, &nshapes);
FOR_EACH(ishape, 0, nshapes) {
  struct s3d_shape* shape;
  s3daw_get_shape(s3daw, ishape, &shape);

  ⟨set the shape hit filter function 58b⟩
```

```
    res = s3d_scene_attach_shape(scn, shape);
    if(res != RES_OK) {
      logger_print(&pt->logger, LOG_ERROR,
        "%s: could not attach the ground geometry to its Star-3D scene\n",
        FUNC_NAME);
      goto error;
    }
  }
```

On notera qu'avant d'ajouter une géométrie `Star-3D` à la scène, nous lui associons une fonction de filtrage d'impact que nous ne décrivons pas ici : cette notion ainsi que sa mise en œuvre sont présentées dans une section spécifique (section 3.6.8).

Vue de la scène

Nous créons désormais une vue de la scène que nous venons de configurer. Dans `Star-3D`, une vue représente un état de la scène auquel l'appelant peut accéder : une fois la vue créée, toute modification apportée à la scène n'influence pas la vue. À sa création, l'appelant définit quels opérateurs d'accès il souhaite pouvoir utiliser pour interroger la géométrie décrite dans la vue. Parmi ces opérateur, on citera l'échantillonnage uniforme de la géométrie ou encore le lancer de rayons. En fonction des opérateurs renseignés, `Star-3D` construit les structures de données nécessaires auxdits opérateurs comme par exemple la cumulée des aires des surfaces pour l'échantillonnage uniforme ou la structure de partitionnement des géométries pour le lancer de rayon.

Dans notre cas, nous souhaitons pouvoir intersecter un rayon avec la scène. Nous construisons donc une vue avec comme seul opérateur d'accès l'opérateur "lancer de rayons", vue que nous stockons comme variable membre de notre application puisque devant être utilisée dans l'étape de rendu.

57a ⟨`pt.h` *forward declarations* 47b⟩+≡
```
  struct s3d_scene_view;
```

57b ⟨`struct pt` *members* 48d⟩+≡
```
  struct s3d_scene_view* scnview;
```

57c ⟨*create the* `Star-3D` *scene view for the ray-tracing* 57c⟩≡
```
  res = s3d_scene_view_create(scn, S3D_TRACE, &pt->scnview);
  if(res != RES_OK) {
    logger_print(&pt->logger, LOG_ERROR,
      "%s: could not create the view of the Star-3D scene\n", FUNC_NAME);
    goto error;
  }
```

3.6.8 Fonction de filtrage d'impact

Lors de l'étape de définition de la scène décrite dans la section précédente, nous associons à chacune de ses géométries une fonction de filtrage d'impact. Cette fonction est invoquée par `Star-3D` lorsqu'un rayon impacte la géométrie. L'appelant reprend alors la main et peut ainsi réaliser les traitements qu'il souhaite en rapport avec l'impact détecté. En retour de la fonction de filtrage, il notifie à `Star-3D` si cette intersection doit être simplement ignorée (c'est à dire filtrée) ou non.

La fonction de filtrage d'impact est un concept central à `Star-3D`. Il permet à l'utilisateur de définir un ensemble de traitements à réaliser *au moment* de l'impact. Le lancer de rayons peut alors être vu comme un simple itérateur pour lequel le traitement appliqué à chaque itération (chaque intersection) est porté par cette fonction. D'aucun peut alors utiliser cette fonctionnalité pour traiter les surfaces transparentes, ou encore mettre en oeuvre des opérateurs de lancer de rayons ne se contentant plus de retourner le seul premier impact le long d'un rayon [GWA16].

Dans notre programme, nous utilisons une fonction de filtrage pour ignorer l'intersection d'un rayon avec la géométrie d'où il démarre. Ces auto-intersections sont généralement traitées en ignorant les impacts dont la distance par rapport à l'origine du rayon est en dessous d'un certain seuil. Reste alors à déterminer le seuil en question : un seuil trop grand conduit à rater des intersections valides là un seuil trop faible n'assure plus l'absence d'auto-intersection. L'utilisation d'une fonction de filtrage évite ce choix arbitraire du seuil tout en étant bien plus robuste.

58a ⟨`pt.c` *helper functions declarations* 58a⟩≡
```
static int
discard_self_hit
  (const struct s3d_hit* hit,
   const float ray_org[3],
   const float ray_dir[3],
   void* ray_data,
   void* filter_data);
```

58b ⟨*set the shape hit filter function* 58b⟩≡
```
res = s3d_mesh_set_hit_filter_function(shape, discard_self_hit, NULL);
if(res != RES_OK) {
  logger_print(&pt->logger, LOG_ERROR,
    "%s: could not setup the hit filter function\n", FUNC_NAME);
  goto error;
}
```

Nous utilisons la fonction `s3d_mesh_set_hit_filter_function` pour associer à la géométrie `Star-3D` notre fonction de filtrage `discard_self_hit`. Le dernier paramètre de la fonction est un pointeur vers les données utilisateur, associées à la géométrie, que celui ci souhaite passer à la fonction de filtrage via son paramètre d'entrée `filter_data`. Dans notre cas, nous n'avons pas de données de ce type à transférer à la fonction de filtrage ; nous passons donc `NULL` comme dernier argument.

Une fonction de filtrage a comme premier paramètre d'entrée l'impact à traiter. Les deux paramètres suivants décrivent le rayon à l'origine de l'impact. Le quatrième paramètre est un pointeur vers des données de l'appelant passées par celui ci via la fonction de lancer de rayons ; ce sont donc des données associées au rayon lancé. Le dernier paramètre pointe enfin vers des données utilisateur relatives à la géométrie impactée, des données que nous avons décidé d'ignorer.

Dans notre programme, nous traitons des géométries uniquement composées de triangles. Par conséquent, un rayon qui démarre d'un triangle ne peut impacter ce même triangle, sauf dans le cas d'une auto-intersection que nous cherchons justement à filtrer. Notre fonction de filtrage se limite donc à comparer la primitive géométrique impactée par le rayon avec la primitive géométrique de laquelle le rayon démarre, une primitive que nous passons en entrée de la fonction de filtrage via le paramètre `ray_data`. Deux primitives identiques signifient que le rayon intersecte sa primitive de départ et que le présent impact est invalide : il doit donc être filtré. Dans le cas contraire, l'impact est tout simplement considéré comme valide.

58c ⟨`pt.c` *helper functions definitions* 49c⟩+≡
```
int
discard_self_hit
  (const struct s3d_hit* hit,
   const float ray_org[3],
   const float ray_dir[3],
   void* ray_data,
   void* filter_data)
{
  const struct s3d_primitive* prim_from = ray_data;
  ASSERT(hit && ray_data && prim_from);
```

```
  /* Avoid "unused variable" warnings */
  (void)ray_org, (void)ray_dir, (void)filter_data;

  return S3D_PRIMITIVE_EQ(prim_from, &hit->prim);
}
```

3.6.9 Configuration des matériaux

Par simplicité, notre moteur de rendu ne supporte qu'un seul matériau pour l'ensemble de la scène. Celui ci est un matériau monochromatique, purement diffusif, dont la réflectance est définie par l'utilisateur via une option de la ligne de commande (section 3.3.5). Pour représenter ce matériau, nous utilisons la bibliothèque `Star-ScatteringFunction` (`Star-SF`) déployée avec le `Star-Engine`. Cette bibliothèque propose plusieurs fonctions de diffusion surfacique bidirectionnelles dont la fonction de réflexion lambertienne qui nous cherchons à utiliser ici. `Star-SF` expose également des fonctions de diffusion volumique (*aka* des fonctions de phase) que nous n'utiliserons pas : nous supposons l'absence de milieu participant et par conséquent aucun phénomène de diffusion dans le milieu n'est à prendre en compte.

Nous commençons par ajouter `Star-SF` à la liste des dépendances de notre programme.

59a ⟨*check for `CMake` packages* 54d⟩+≡
```
find_package(StarSF 0.6 REQUIRED)
```

59b ⟨*list of paths where to look for the dependency headers* 54e⟩+≡
```
${StarSF_INCLUDE_DIR}
```

59c ⟨*list of used libraries* 54f⟩+≡
```
StarSF
```

Puis nous ajoutons une fonction de diffusion surfacique à la liste des variables membres de l'application.

59d ⟨`pt.h` *forward declarations* 47b⟩+≡
```
struct ssf_bsdf;
```

59e ⟨`struct pt` *members* 48d⟩+≡
```
struct ssf_bsdf* bsdf;
```

Enfin nous créons notre fonction lambertienne que nous configurons avec la réflectance soumise comme argument du programme.

59f ⟨`pt.c` *inclusions* 47d⟩+≡
```
#include <star/ssf.h>
```

59g ⟨*setup the scene material* 59g⟩≡
```
res = ssf_bsdf_create(&pt->allocator, &ssf_lambertian_reflection, &pt->bsdf);
if(res != RES_OK) {
  logger_print(&pt->logger, LOG_ERROR,
    "%s: could not create the lambertian BSDF\n", FUNC_NAME);
    goto error;
}
res = ssf_lambertian_reflection_setup(pt->bsdf, args->reflectivity);
if(res != RES_OK) {
  logger_print(&pt->logger, LOG_ERROR,
    "%s: could not setup the lambertian BSDF\n", FUNC_NAME);
  goto error;
}
```

3.6.10 La constante PT_NULL

Nous définissons dans l'interface de programmation de l'application la constante `PT_NULL`, de type `struct pt`, qui caractérise l'état d'une application non initialisée. Nous utiliserons cette constante pour affecter une valeur "nulle" aux variables membres de l'application avant que celle-ci soit effectivement initialisée. L'objectif est de pouvoir invoquer la fonction `pt_release` indépendemment de l'état d'avancement de l'étape d'initialisation. Dès lors, nous pourrons utiliser cette fonction dans un bloc de gestion d'erreur puisque la variable application, sans être totalement ou du tout intialisée, aura un état nul que nous pourrons tester afin de savoir lesquelles de ses variables membres doivent être libérées.

Pour cette constante, nous souhaitons finalement affecter zéro à l'ensemble de ses membres (`NULL` dans le cas de pointeurs). Or, justement, la norme `C` nous assure que l'ensemble des membres d'une variable statique est initialisé implicitement à zéro ; la seule définition de notre constante suffit donc à l'initialiser comme voulu.

60a ⟨`pt.h` *data types* 46d⟩+≡
```
static const struct pt PT_NULL; /* Implicitly initialised to 0 */
```

Nous ajoutons enfin deux variables membres à la structure de données "application", qui notifient si son allocateur et son journal d'évènements ont été initialisés. Ces derniers sont de types structurés et il est donc difficile de connaître à priori leur état d'initialisation sans interroger leurs variables membres. Nous stockons donc cet état directement dans notre structure.

60b ⟨`struct pt` *members* 48d⟩+≡
```
int allocator_is_init;
int logger_is_init;
```

60c ⟨*setup the allocator* 48e⟩+≡
```
pt->allocator_is_init = 1;
```

60d ⟨*setup the logger* 49b⟩+≡
```
pt->logger_is_init = 1;
```

3.6.11 Libérer l'application

Nous définissons ici la fonction `pt_release` qui relâche les variables membres détenues par l'application et ferme le fichier de sortie en fonction de leur état d'initialisation. Nous vérifions enfin que l'ensemble des blocs mémoire alloués par l'allocateur sont désormais libérés afin d'afficher un message d'erreur dans le cas contraire.

60e ⟨`pt.c` *functions definitions* 47e⟩+≡
```
void
pt_release(struct pt* pt)
{
  ASSERT(pt);

  if(pt->camera) pt_camera_ref_put(pt->camera);
  if(pt->image) pt_image_ref_put(pt->image);
  if(pt->s3d) s3d_device_ref_put(pt->s3d);
  if(pt->scnview) s3d_scene_view_ref_put(pt->scnview);
  if(pt->bsdf) ssf_bsdf_ref_put(pt->bsdf);
  if(pt->output && pt->output != stdout) fclose(pt->output);
  if(pt->logger_is_init) logger_release(&pt->logger);
  if(pt->allocator_is_init) {
    const size_t memsz = MEM_ALLOCATED_SIZE(&pt->allocator);
    if(memsz) { /* Check memory leaks */
      fprintf(stderr, "%s: memory leaks: %lu Bytes\n",
        FUNC_NAME, (unsigned long)memsz);
```

```
    }
    mem_shutdown_regular_allocator(&pt->allocator);
  }
  *pt = PT_NULL;
}
```

3.7 Rendre l'image

Dans cette section nous développons la fonction qui réalise le traitement du programme à proprement parler. Cette fonction fait partie de l'interface de l'application. Elle est invoquée à partir de la fonction principale pour calculer l'image à rendre et la stocker dans le fichier en sortie. Son seul paramètre d'entrée est l'application elle même qui contient en définitive l'ensemble des données qui lui sont nécessaires (caméra, image, géométrie, fichier en sortie, *etc.*).

61a ⟨`pt.h` *functions declarations* 47c⟩≡

```
extern LOCAL_SYM res_T
pt_run
  (struct pt* pt);
```

61b ⟨`pt.c` *functions definitions* 47e⟩≡

```
res_T
pt_run(struct pt* pt)
{
  res_T res = RES_OK;
  ASSERT(pt);

  ⟨draw the image 65a⟩
  ⟨write the image to the output file 76b⟩

exit:
  return res;
error:
  goto exit;
}
```

Dans notre programme, nous supposons l'absence de milieu participant et, par conséquent, qu'un chemin lumineux voyage en ligne droite entre deux surfaces. Le seul matériau la composant est un matériau purement diffusif et, toujours dans un souci de simplicité, la seule source lumineuse de la scène est la voûte céleste que nous assumons être uniforme avec un éclairement de 1. En fixant l'éclairement à 1, nous assurons que la luminance estimée en chaque pixel sera inférieure ou égale à 1, facilitant ainsi sa conversion et son encodage dans un format d'image conventionnel. Et en l'occurrence, nous écrivons en sortie une image au format *Portable Gray Format* (PGM) dans lequel la couleur d'un pixel est stockée sur une seule composante encodée sur 8-bits. Notre programme de rendu étant monochromatique, nous pouvons nous contenter d'une seule composante par pixel en lieu et place d'un format Rouge Vert Bleu plus classique. De plus, ce format de fichier est non seulement très facile à écrire mais est aussi largement supporté par les logiciels de visualisation ou de traitement d'images.

Nous soulignons que le choix d'écrire en sortie ce format d'image ne s'explique que par un souci de simplicité. Ce faisant, nous réduisons non seulement la précision de la valeur au pixel, mais nous ignorons aussi l'écart type de son estimation. De plus, nous ne stockons pas la luminance en tant que telle mais une couleur, traduite dans un espace de colorimétrie spécifique, après éventuellement une étape de mappage tonal et de correction de tons. L'alternative consisterait à sortir directement pour chaque pixel l'estimation de sa luminance ainsi que son écart type. Une image stockée dans un tel format devrait par conséquent être post-traitée pour être convertie dans un format conventionnel avant d'être visualisée. L'avantage évident est qu'un tel format donne un contrôle total à l'utilisateur sur l'image qu'il souhaite afficher. Il

peut par exemple appliquer l'opérateur de mappage tonal qu'il souhaite ou encoder l'image dans l'espace de couleur de son choix. Au delà de cette dimension artistique, il conserve aussi les résultats bruts de la simulation, l'autorisant ainsi à les analyser. Nous encourageons donc le lecteur à mettre en œuvre ce type de format de fichier à titre d'exercice et à développer les outils de post-traitements qu'il souhaite au regard de ces données brutes.

3.7.1 Algorithme de rendu

Notre algorithme de rendu est basé sur le calcul de la luminance moyenne captée par chaque pixel d'une grille, dans l'angle solide correspondant à ce pixel.

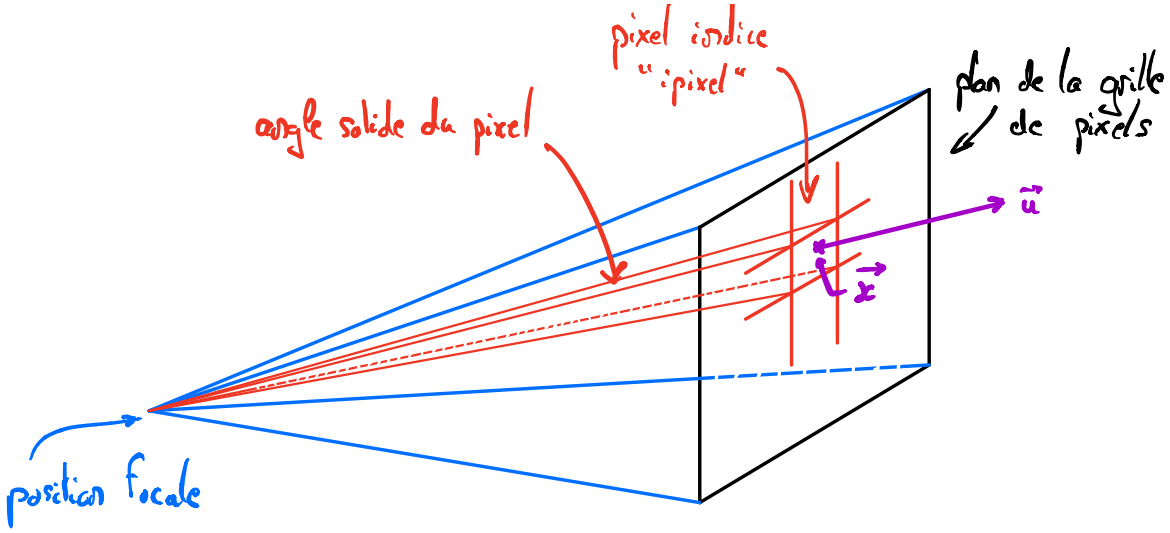

FIGURE 3.2 – Schéma de la caméra sténopé utilisée : la position de départ $\mathbf{x}$ et la direction de départ $\mathbf{u}$ du rayon depuis un pixel donné sont liées.

$$< L > (ipixel) = \int_{\Delta\Omega(ipixel)} \frac{1}{\Omega(ipixel)} d\omega(\mathbf{u}) \Big\{ L(\mathbf{x}, -\mathbf{u}) \Big\} \tag{3.2}$$

La relation 3.2 donne l'expression de la luminance moyenne $< L > (ipixel)$ captée par le pixel d'indice $ipixel$. Cette expression peut se lire directement comme un algorithme de Monte-Carlo consistant à échantillonner de façon uniforme une direction initiale $\mathbf{u}$ dans l'angle solide $\Delta\Omega(ipixel)$ correspondant au pixel. Dans le cas général, il faudrait également intégrer la luminance reçue sur la surface du pixel, mais dans le cas de la caméra sténopé que nous utilisons ici, la position de départ du rayon depuis le pixel et sa direction initiale sont liées via la position focale (voir figure 3.2). Le poids de la réalisation statistique est la luminance $L(\mathbf{x}, -\mathbf{u})$ incidente en $\mathbf{x}$ dans la direction $-\mathbf{u}$. Et pour calculer cette luminance, nous utilisons là aussi un algorithme Monte-Carlo décrit de façon synthétique par la relation suivante :

$$L(\mathbf{x}, -\mathbf{u}) = L(\mathbf{x_{int}}, -\mathbf{u}) \mathcal{H}(\mathbf{u} \cap \mathcal{B}) + L_0 \mathcal{H}(\mathbf{u} \cap \Omega^+) + 0 \mathcal{H}(\mathbf{u} \cap \Omega^-) \tag{3.3}$$

L'équation 3.3 traduit simplement le fait que trois cas de figure peuvent être rencontrés lorsqu'un rayon est émis depuis la position $\mathbf{x}$ dans la direction $\mathbf{u}$ (voir figure 3.3). Dans le premier cas, le rayon intersecte une surface $\mathcal{B}$ à la position $\mathbf{x_{int}}$. Il s'agit alors de calculer la luminance $L(\mathbf{x_{int}}, -\mathbf{u})$ qui se propage en $\mathbf{x_{int}}$ dans la direction $-\mathbf{u}$. Étant donné qu'aucun milieu semi-transparent n'est à prendre en compte, il n'y a pas besoin de représenter l'atténuation de la luminance le long du trajet entre les positions $\mathbf{x_{int}}$ et $\mathbf{x}$. Dans le second cas, le rayon n'intersecte pas de surface mais l'hémisphère supérieur représenté

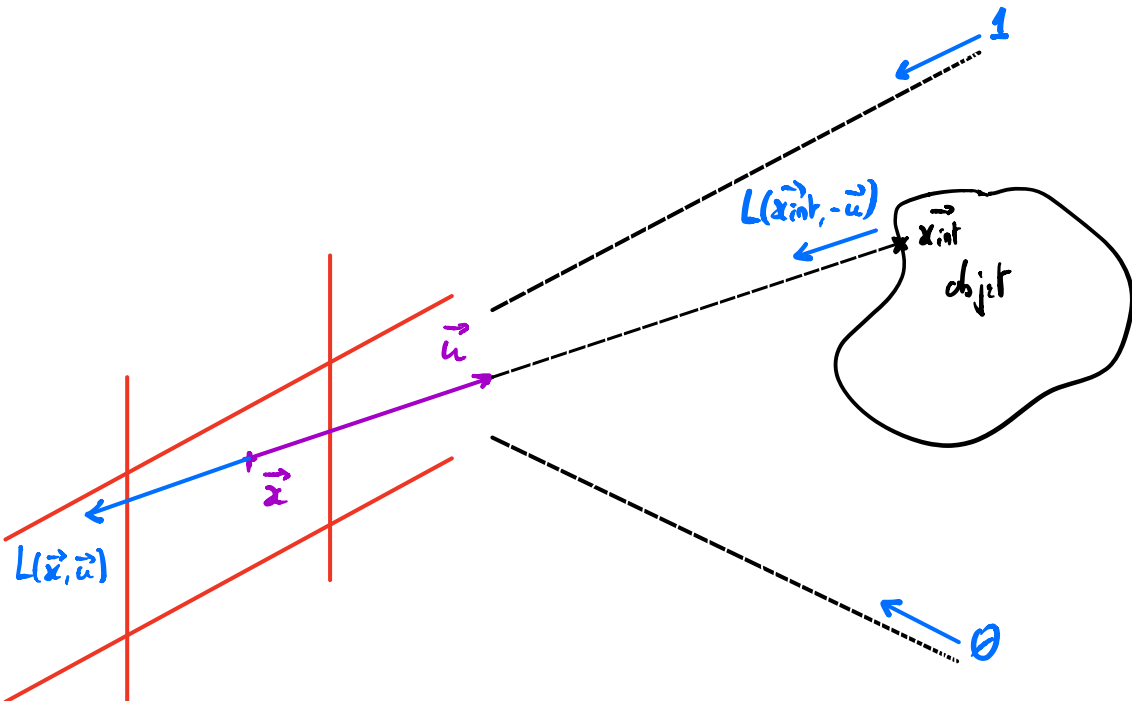

FIGURE 3.3 – La luminance qu'il faut intégrer est $L(\mathbf{x}, -\mathbf{u})$, la luminance se propageant en $\mathbf{x}$ selon la direction $-\mathbf{u}$. Trois cas de figure sont possibles : le rayon $(\mathbf{x},\mathbf{u})$ intersecte une paroi en $\mathbf{x_{int}}$ et $L(\mathbf{x}, -\mathbf{u}) = L(\mathbf{x_{int}}, -\mathbf{u})$; le rayon est perdu dans la voûte céleste, et $L(\mathbf{x}, -\mathbf{u}) = L_0 = 1$; le rayon est perdu dans les abîmes et $L(\mathbf{x}, -\mathbf{u}) = 0$.

par Ω^+, dont la luminance isotrope L_0 vaut 1. Ici, le poids de la réalisation doit être incrémenté par la valeur de L_0 multipliée par la transmissivité du trajet optique. Enfin, dans le troisième cas le rayon n'a pas intersecté de surface, mais n'a pas non plus intersecté la "voûte céleste" : il a donc nécessairement intersecté les "abîmes", c'est à dire l'hémisphère inférieur de la scène représenté par Ω^-. Dès lors, le poids de la réalisation est incrémenté de zéro qui est en l'occurrence la valeur de la luminance pour l'ensemble de cet hémisphère.

Définir l'intersection $\mathbf{x_{int}}$ d'un rayon avec la géométrie de la scène est tout l'enjeu de l'algorithme de suivi de chemins que nous mettons en œuvre dans la partie 3.7.6. Dans cette même partie, nous décrivons également comment nous déterminons où est situé la voûte céleste Ω^+ et donc, par symétrie, les abîmes Ω^-.

Lorsqu'une surface $\mathcal{B}$ est intersectée par un rayon, il s'agit alors d'évaluer la luminance $L(\mathbf{x_{int}}, -\mathbf{u})$ qui, contrairement aux abîmes et à la voûte céleste, n'est pas connue. Soit $\mathbf{n}$ la normale à la surface $\mathcal{B}$ en $\mathbf{x_{int}}$, cette luminance est la somme des contributions, pour toutes les directions incidentes $\mathbf{u}'$ sur l'hémisphère supérieur $2\pi^+$ défini par $\mathbf{n}$, de la puissance incidente en $\mathbf{x_{int}}$ dans la direction $\mathbf{u}'$, et réfléchie dans la direction $\mathbf{u}$:

$$L(\mathbf{x_{int}}, -\mathbf{u}) = \rho \int_{2\pi^+} BRDF(\mathbf{u}, \mathbf{u}')|\mathbf{u}' \cdot \mathbf{n}| L(\mathbf{x_{int}}, -\mathbf{u}')d\omega(\mathbf{u}') \tag{3.4}$$

avec ρ la réflectivité totale hémisphérique de la surface en $\mathbf{x_{int}}$, et $BRDF(\mathbf{u}, \mathbf{u}')$ la fonction de densité de réflectivité bidirectionnelle, définie comme la fraction de l'éclairement en $\mathbf{x_{int}}$ selon $\mathbf{u}$ qui est réfléchie dans la direction $\mathbf{u}'$ (voir figure 3.4).

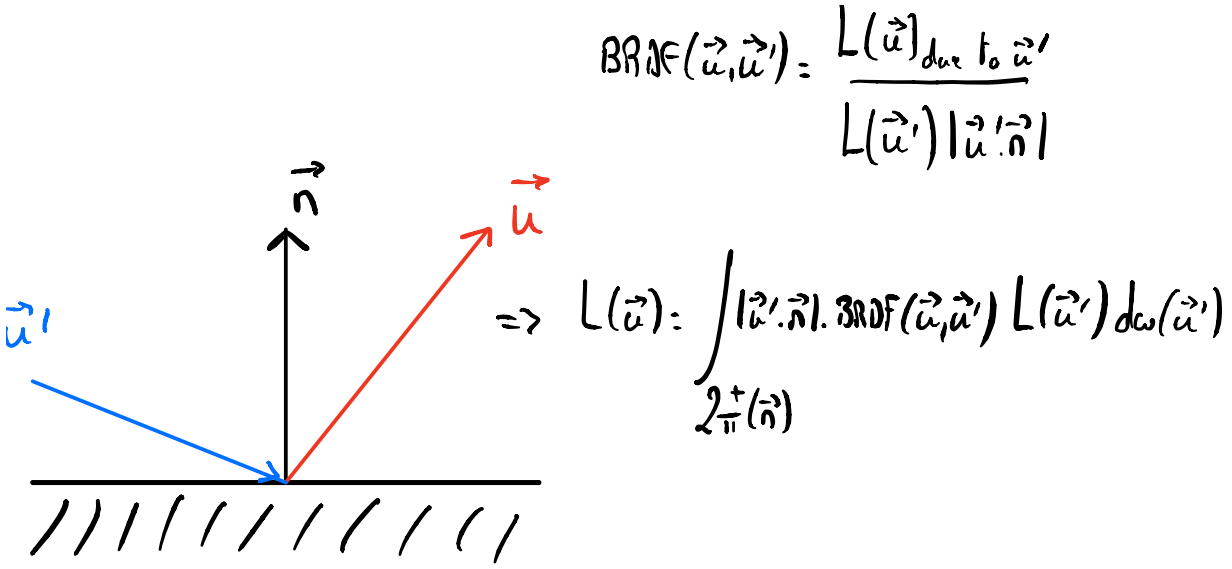

FIGURE 3.4 – Représentation du principe d'échantillonnage d'une direction $\mathbf{u}'$ de propagation après réflexion en paroi et définition de la fonction BRDF($\mathbf{u}$,$\mathbf{u}'$).

La relation 3.4 correspond une nouvelle fois à un algorithme de Monte-Carlo qui consiste à tenir compte de la réflectivité totale de la paroi ρ. Cette réflectivité étant un coefficient positif inférieur à 1, il s'agit en définitive de la probabilité que la collision en $\mathbf{x_{int}}$ soit une réflexion. Pour prendre en compte cette probabilité, d'aucun peut considérer que le trajet optique est quoiqu'il en soit réfléchi en paroi mais que sa transmissivité après rebond est alors pondérée par ρ ; une fraction ρ de la puissance est réfléchie, et sa fraction complémentaire 1-ρ est absorbée par la paroi. L'alternative consiste à ne considérer que seule une proportion ρ des trajets optiques sera réfléchie en $\mathbf{x_{int}}$, la fraction complémentaire 1-ρ correspondant dès lors à une absorption *totale* en paroi. Un simple test de Bernouilli sur la valeur de ρ permet alors de déterminer si le trajet optique doit s'arrêter avec un poid nul ou au contraire doit continuer en conservant sa transmissivité courante.

Indépendamment de la solution retenue pour prendre en compte la réflectivité ρ au point d'impact, dans le cas d'un rebond en paroi nous échantillonnons une nouvelle direction de propagation $\mathbf{u}'$ dans l'hémisphère $2\pi^+$ défini par $\mathbf{n}$. Cet échantillonnage doit être assuré selon la fonction de densité de probabilité $p(\mathbf{u},\mathbf{u}') = |\mathbf{u}'\cdot\mathbf{n}|BRDF(\mathbf{u},\mathbf{u}')$ (par exemple, $p(\mathbf{u},\mathbf{u}') = \frac{|\mathbf{u}'\cdot\mathbf{n}|}{\pi}$ pour une surface diffuse). Le poids de la réalisation est alors incrémenté par la luminance $L(\mathbf{x_{int}},-\mathbf{u}')$ se propageant en $\mathbf{x_{int}}$ dans la direction $-\mathbf{u}'$ multiplié par la transmissivité du trajet : l'algorithme boucle alors à la relation 3.3 pour ces nouvelles valeurs de $\mathbf{x}$ et $\mathbf{u}$.

La subtilité mentionnée précédemment, concernant la façon dont la réflectivité totale ρ peut être considérée, conduit à deux branches algorithmiques. Notre algorithme commence par utiliser la première variante, qui consiste à multiplier la transmissivité du trajet par ρ puis à échantillonner une nouvelle direction de propagation $\mathbf{u}'$. Cette variante présente l'avantage de ne pas arrêter les trajectoires optiques de façon systématique dans le cas où par exemple la réflectivité totale est proche de zéro. Une fois la transmissivité du trajet inférieure à un certain seuil, nous décidons alors de basculer sur la variante "roulette russe" qui consiste à continuer ou arrêter la trajectoire au prorata, respectivement, des probabilités ρ et 1-ρ. Ce basculement vers la "roulette russe" permet d'assurer qu'une trajectoire optique aura une probabilité d'être arrêtée dans le cas où, par exemple, elle serait piégée dans une cavité réfléchissante avec une probabilité très faible de rencontrer la voûte céleste. Si seulement la première variante algorithmique devait être utilisée, ce genre de trajectoires pourrait être réfléchi un très grand nombre de fois avant de pouvoir quitter la cavité dans laquelle il se trouve.

Un dernier cas de figure reste problématique : celui où la position de la caméra est située à l'intérieur d'une cavité fermée. Il est impossible pour un trajet optique de sortir de cette cavité, et donc de rencontrer la voûte céleste : certes, la luminance en toute position et dans toute direction à l'intérieur de cette cavité est nulle, mais l'algorithme de Monte-Carlo n'est pas informé du fait que le trajet est cantonné à une cavité fermée totalement isolée (d'un point de vue radiatif) du reste de la scène. Dans le cas où la réflectivité du matériau est inférieure à 1, la transmisivité du trajet finira par descendre en dessous du seuil permettant de basculer en variante "roulette russe", et il existera donc une probabilité d'arrêter les trajets optiques (au prorata de la probabilité $1-\rho$). Mais dans le cas où la réflectivité du matériau est exactement $\rho = 1$, cette probabilité d'arrêt des trajets est strictement nulle. Il faut donc imaginer un mécanisme spécifique à ce cas de figure. La relation 3.4 peut être, par exemple, ré-écrite sous la forme :

$$L(\mathbf{x_{int}}, -\mathbf{u}) = \frac{1}{2}\rho \int_{2\pi^+} BRDF(\mathbf{u}, \mathbf{u}')|\mathbf{u}' \cdot \mathbf{n}|\Big\{2L(\mathbf{x_{int}}, -\mathbf{u}')\Big\}d\omega(\mathbf{u}') + \frac{1}{2}\Big\{0\Big\} \tag{3.5}$$

Cette écriture permet de trouver une solution dans le cas particulier où les trajets lumineux sont piégés dans une cavité fermée parfaitement réfléchissante : un test de Bernouilli permet alors de décider si, lors d'une réflexion en paroi, le trajet optique doit continuer avec une probabilité 1/2 (et le poids de la réalisation doit être multiplié par 2) ou si le trajet optique peut être arrêté, avec une probabilité 1/2, et un poids nul. De façon à ne pas créer de variance artificielle (multiplication du poids par 2), cette variante algorithmique peut n'être activée, par exemple, que lorsque le nombre de réflexions qu'à connu un trajet optique donné est supérieur à un seuil prédéterminé.

Le poids de la réalisation statistique, qui correspond en définitive à la luminance portée par une trajectoire optique, est donc obtenu comme une somme de contributions calculées lors de chaque collision avec une paroi, avec la voûte céleste ou avec les abîmes. Dans le cas d'une collision avec la voûte céleste ou les abîmes, nous incrémentons le poids avec une valeur correspondant à la transmissivité courante du trajet multiplié par la luminance en bout de celui ci (L_0 pour la voûte céleste, 0 pour les abîmes). Dans le cas d'une collision en paroi soit nous modifions la transmissivité du chemin, soit nous l'arrêtons ou le poursuivons sans changer sa transmissivité et donc sans changer le poids de la réalisation.

3.7.2 Dessiner l'image

Nous développons la fonction qui calcule l'image. Cette fonction itère simplement sur les pixels de l'image et pour chacun d'eux estime sa luminance en utilisant l'algorithme Monte-Carlo décrit dans la section précédente.

65a ⟨*draw the image* 65a⟩≡
```
res = draw_image(pt);
if(res != RES_OK) {
  logger_print(&pt->logger, LOG_ERROR,
    "%s: error during the rendering\n", FUNC_NAME);
  goto error;
}
```

65b ⟨`pt.c` *helper functions definitions* 49c⟩≡
```
static res_T
draw_image(struct pt* pt)
{
  ⟨draw_image local variables 66d⟩
  res_T res = RES_OK;
  ASSERT(pt);

  ⟨define the number of threads to use 67a⟩
  ⟨setup the random number generator for multi threads 68c⟩

  ⟨for each pixel in parallel 67c⟩ {
    ⟨get identifier of the thread that computes this pixel 69c⟩
```

```
    ⟨estimate the radiance of the pixel 69d⟩
  }

exit:
  ⟨release draw_image local variables 69a⟩
  return res;
error:
  goto exit;
}
```

Comme tout algorithme Monte-Carlo, l'estimation de la luminance en chaque pixel se parallélise très bien. Cependant, en lieu et place de paralléliser cette estimation, nous parallélisons le rendu de l'image dans sa totalité, et ce afin d'améliorer l'efficacité de la parallélisation. En effet, en parallélisant le calcul de l'image, les processus de calcul ne sont synchronisés qu'une seule fois, à la fin du rendu, là où une parallélisation à l'échelle du pixel demanderait une synchronisation après l'estimation de chaque pixel. De plus, dans le cadre d'un algorithme de rendu, le nombre de réalisations par pixel est généralement 1 à 3 ordres de grandeur plus bas que le nombre de réalisations utilisé pour une simulation Monte-Carlo plus conventionnelle, et ce non seulement pour limiter le temps de calcul mais aussi car l'œil filtre naturellement l'imprécision numérique par voisinage de pixel. Par conséquent, le calcul d'un seul pixel peut ne pas occuper l'ensemble des processus de calcul, ce qui pour une image devient bien plus improbable ; le nombre de pixels à calculer étant généralement bien plus important que le nombre de réalisations par pixel.

3.7.3 Parallélisation du calcul

Pour paralléliser le rendu de l'image nous utilisons le standard OpenMP [DM98] qui permet de distribuer un calcul sur plusieurs processus légers (ou *thread*) dans une environnement à mémoire partagée. Nous commençons donc par ajouter OpenMP comme une dépendance du projet.

66a ⟨*check for* `CMake` *packages* 54d⟩≡
```
find_package(OpenMP 1.2 REQUIRED)
```

66b ⟨*configure the target* 22b⟩≡
```
set_target_properties(pt PROPERTIES
  COMPILE_FLAGS "${OpenMP_C_FLAGS}"
  LINK_FLAGS "${OpenMP_C_FLAGS}")
```

Cette étape est légèrement différente des précédents ajouts de dépendances. Nous utilisons toujours le système de paquets `CMake` pour vérifier qu'OpenMP est présent sur le système, mais à la place d'enrichir la variable `CMAKE_INCLUDE_DIR` avec le répertoire dans lequel ses fichiers d'en-tête se trouvent, nous venons ajouter un jeu d'options de compilations à notre programme. De même, à la place de lier OpenMP à notre programme à l'aide de la fonction `target_link_libraries`, nous venons là aussi ajouter des options d'édition de liens. Ces différences s'expliquent par le fait que, jusqu'alors, nous ajoutions des dépendances issues du `Star-Engine` là où le paquet OpenMP distribué avec `CMake` repose sur une autre logique qu'ici nous utilisons.

Nous déterminons alors le nombre de processus de calcul à utiliser. Dans notre programme, nous choisissons d'utiliser autant de processus de calcul que de processeurs présents sur la machine hôte. Nous interrogeons donc OpenMP pour déterminer le nombre de processeurs disponibles sur notre système.

66c ⟨`pt.c` *inclusions* 47d⟩≡
```
#include <omp.h>
```

66d ⟨`draw_image` *local variables* 66d⟩≡
```
size_t nthreads;
```
Définit:
`nthreads`, utilisé dans les morceaux 67–69.

67a ⟨*define the number of threads to use* 67a⟩≡
```
nthreads = (size_t)omp_get_num_procs();
```
Utilise `nthreads` 66d.

Nous distribuons alors le rendu par *une* boucle parallèle qui itère sur les pixels de l'image. Il est courant que le parcours exhaustif des pixels d'une image soit mis en œuvre à l'aide de 2 boucles imbriquées, l'une itérant sur les lignes et l'autre sur les colonnes de l'image. Ici, nous utilisons une seule boucle sur l'ensemble de pixels, et ce afin de paralléliser l'estimation des pixels et non le calcul d'une ligne ou d'une colonne de l'image. Un pixel est dès lors identifié par un indentifiant à une dimension qui ne dit à priori rien sur le pixel 2D auquel il correspond. Nous discutons de cette associativité entre l'indifiant 1D et son pixel dans la section 3.7.5.

Pour paralléliser une boucle, OpenMP partitionne le domaine d'itération en tronçons de taille approximativement identique. Chaque sous domaine est alors attribué à un processus de calcul selon une politique de répartition de charge spécifique. Afin d'assurer la reproductibilité du calcul, nous forçons OpenMP à répartir ces sous ensembles de manière *statique* ce qui signifie que pour un nombre de processus fixé, un même pixel sera toujours traité par le même processus. Avec cette politique de distribution de charge, chaque tronçon est alors attribué à un processus de manière circulaire. On notera cependant qu'une image peut être constituée de larges zones homogènes, peu coûteuses à calculer. En découpant l'image en grandes zones continues de pixels, un processus pourrait être amené à traiter seul les pixels de l'image coûteux à évaluer là ou d'autres processus n'auraient au contraire que peu de calcul à réaliser. Afin de mieux répartir la charge de calcul, nous forçons OpenMP à découper l'image en tronçon de 1024 pixels dans le but de distribuer une même zone de l'image sur l'ensemble des processus qui participent à son rendu.

67b ⟨`draw_image` *local variables* 66d⟩+≡
```
int64_t ipixel, npixels;
```

67c ⟨*for each pixel in parallel* 67c⟩≡
```
npixels = (int64_t)
  (pt_image_get_width(pt->image) * pt_image_get_height(pt->image));
omp_set_num_threads((int)nthreads);
#pragma omp parallel for schedule(static, 1024/*chunk size*/)
for(ipixel=0; ipixel < npixels; ++ipixel)
```
Utilise `nthreads` 66d.

3.7.4 Le générateur de nombres aléatoires

Comme tout estimateur Monte-Carlo, notre algorithme de rendu utilise un générateur de nombres aléatoires pour échantillonner différentes fonctions de distribution. `Star-Sampling` (ou `Star-SP`) est une bibliothèque installée avec le `Star-Engine` qui propose, entre autres, plusieurs types de générateurs. Nous ajoutons cette bibliothèque comme dépendance de notre programme afin de pouvoir utiliser un de ces générateurs.

67d ⟨*check for `CMake` packages* 54d⟩+≡
```
find_package(StarSP 0.8 REQUIRED)
```

67e ⟨*list of paths where to look for the dependency headers* 54e⟩≡
```
${StarSP_INCLUDE_DIR}
```

67f ⟨*list of used libraries* 54f⟩≡
```
StarSP
```

La parallélisation du calcul de l'image pose la question de l'indépendance des nombres aléatoires entre les différents processus. En effet, pour assurer une indépendance statistique stricte entre ces processus, nous devons garantir que chacun d'entre eux utilise une séquence de nombres aléatoires qui lui est propre. Pour cela, nous pouvons découper une séquence de nombres aléatoires en sous séquences, attribuées de manière circulaire aux différents processus. `Star-SP` propose ce type de fonctionnalité à travers la notion de générateur mandataire ou générateur *proxy*. Un tel générateur se charge de partitionner une séquence de nombres aléatoires en sous séquences qu'il distribue alors aux différents générateurs dont il est le mandataire ; chaque générateur disposant alors de sa propre séquence.

Nous commençons donc tout d'abord par créer un générateur mandataire. Pour cela, nous invoquons la fonction `ssp_rng_proxy_create` et lui passons en paramètre d'entrée non seulement l'allocateur utilisé pour allouer son espace mémoire mais aussi le type de générateur de nombres aléatoires que l'on souhaite utiliser (ici un Mersenne Twister sur 64-bits) ainsi que le nombre de sous générateurs dont le présent générateur est mandataire. Et en l'occurrence nous en souhaitons autant que de processus concourants au calcul de l'image.

68a ⟨`pt.c` *inclusions* 47d⟩+≡
```
#include <star/ssp.h>
```

68b ⟨`draw_image` *local variables* 66d⟩+≡
```
struct ssp_rng_proxy* rng_proxy = NULL;
```

68c ⟨*setup the random number generator for multi threads* 68c⟩≡
```
res = ssp_rng_proxy_create
  (&pt->allocator, &ssp_rng_mt19937_64, nthreads, &rng_proxy);
if(res != RES_OK) {
  logger_print(&pt->logger, LOG_ERROR,
    "%s: could not create the proxy RNG\n", FUNC_NAME);
  goto error;
}
```
Utilise `nthreads` 66d.

Nous allouons ensuite le tableau dans lequel nous listons les générateurs de chaque processus. Nous demandons alors au générateur mandataire de nous attribuer un générateur par séquence aléatoire unique que nous stockons dans ce tableau.

68d ⟨`draw_image` *local variables* 66d⟩+≡
```
struct ssp_rng** rngs = NULL;
size_t i;
```

68e ⟨*setup the random number generator for multi threads* 68c⟩+≡
```
rngs = MEM_CALLOC(&pt->allocator, nthreads, sizeof(*rngs));
if(!rngs) {
  logger_print(&pt->logger, LOG_ERROR,
    "%s: could not allocate the array of per thread RNG\n", FUNC_NAME);
  res = RES_MEM_ERR;
  goto error;
}

FOR_EACH(i, 0, nthreads) {
  res = ssp_rng_proxy_create_rng(rng_proxy, i, &rngs[i]);
  if(res != RES_OK) {
    logger_print(&pt->logger, LOG_ERROR,
      "%s: could not create the RNG of the thread %lu\n",
      FUNC_NAME, (unsigned long)i);
    goto error;
  }
}
```
Utilise `nthreads` 66d.

En fin de fonction, nous n'oublions pas de libérer l'ensemble de ces générateurs puisque ces derniers sont définis en tant que variables locales à la fonction courante.

69a ⟨*release* `draw_image` *local variables* 69a⟩≡
```
if(rng_proxy) ssp_rng_proxy_ref_put(rng_proxy);
if(rngs) {
  FOR_EACH(i, 0, nthreads) {
    if(rngs[i]) ssp_rng_ref_put(rngs[i]);
  }
  MEM_RM(&pt->allocator, rngs);
}
```
Utilise `nthreads` 66d.

3.7.5 Dessiner un pixel

Nous avons vu précédemment que dans notre programme, la luminance captée par un pixel est estimée par un seul processus et ce à l'aide d'un générateur aléatoire qui lui est propre. Nous mettons en œuvre cette estimation dans une fonction séparée que nous invoquons à chaque pixel.

69b ⟨`pt.c` *helper functions declarations* 58a⟩≡
```
static void
draw_pixel
  (struct pt* pt,
   struct ssp_rng* rng,
   const size_t ipixel);
```

69c ⟨*get identifier of the thread that computes this pixel* 69c⟩≡
```
const int ithread = omp_get_thread_num();
```

69d ⟨*estimate the radiance of the pixel* 69d⟩≡
```
draw_pixel(pt, rngs[ithread], (size_t)ipixel);
```

C'est dans cette fonction que nous réalisons l'intégration Monte-Carlo telle que décrite par la relation 3.2 (section 3.7.1) : à chaque réalisation, nous suivons un chemin lumineux qui passe à travers le pixel et participe donc à la luminance qu'il capte. Pour ce faire, nous échantillonnons une direction dans l'angle solide défini par la position de la caméra et le pixel considéré. Nous évaluons ensuite la luminance le long de cette direction en remontant le chemin vers les sources de la scène. La luminance ainsi déterminée est le poids de notre réalisation que nous ajoutons enfin à l'estimation du pixel.

69e ⟨`pt.c` *helper functions definitions* 49c⟩+≡
```
void
draw_pixel(struct pt* pt, struct ssp_rng* rng, const size_t ipixel)
{
  ⟨draw_pixel local variables 70a⟩
  size_t irealisation;
  ASSERT(pt && rng);

  ⟨compute the pixel coordinates from its 1D index 70b⟩
  ⟨fetch and clean up the pixel data 70d⟩

  FOR_EACH(irealisation, 0, pt->spp/*#samples per pixels*/) {
    ⟨local variables of the realisation 70g⟩

    ⟨sample a direction into the solid angle of the pixel 70h⟩
    ⟨compute the radiance that reaches the pixel along the camera ray 71e⟩
    ⟨add the computed radiance against the pixel accumulators 71f⟩
  }
}
```

Avant l'itération Monte-Carlo en tant que telle, nous déterminons tout d'abord à quel pixel fait référence l'argument `ipixel` passé en entrée de fonction. Pour cela, nous assimilons cet identifiant 1D à un index dans une liste dans laquelle les pixels de l'image seraient ordonnées ligne par ligne.

70a ⟨`draw_pixel` *local variables* 70a⟩≡
```
size_t img_definition[2];
size_t pix_idx[2];
```

70b ⟨*compute the pixel coordinates from its 1D index* 70b⟩≡
```
/* Compute the image space pixel coordinates */
img_definition[0] = pt_image_get_width(pt->image);
img_definition[1] = pt_image_get_height(pt->image);
pix_idx[0] = ipixel % img_definition[0];
pix_idx[1] = ipixel / img_definition[0];
```

Cette associativité est purement arbitraire et d'aucun pourrait utiliser une toute autre stratégie, bien que la présente proposition ait pour mérite de suivre la structuration des pixels en mémoire (section 3.5.2). Nous pourrions par exemple assurer une plus grande cohérence spatiale entre les identifiants en listant les pixels non pas ligne par ligne mais suivant leur code de Morton [Mor66] ou le long de la courbe de Hilbert les parcourant [Hil91].

Une fois les coordonnée 2D du pixel déterminées, nous récupérons un pointeur vers la zone mémoire qui stocke l'estimation de sa luminance et nous l'initialisons à zéro. Nous pouvons dès lors débuter l'intégration Monte-Carlo en tant que telle.

70c ⟨`draw_pixel` *local variables* 70a⟩+≡
```
struct pt_pixel* pixel;
```

70d ⟨*fetch and clean up the pixel data* 70d⟩≡
```
pixel = pt_image_at(pt->image, pix_idx[0], pix_idx[1]);
*pixel = PT_PIXEL_NULL;
```

Au début de chaque réalisation, nous échantillonnons une direction dans l'angle solide défini par la caméra et le pixel. Pour ce faire, nous pré-calculons, avant la boucle principale, la taille du pixel ainsi que sa borne inférieure en X et en Y dans le plan image normalisé. Dans ce repère 2D, la valeur 1 correspond à la taille de l'image dans la dimension considérée. Les deux variables que nous calculons ici définissent en définitive le rectangle englobant du pixel relativement au rectangle de l'image.

70e ⟨`draw_pixel` *local variables* 70a⟩+≡
```
double pix_size[2];
double pix_low[2];
```

70f ⟨*compute the pixel coordinates from its 1D index* 70b⟩+≡
```
/* Lower left corner of the pixel in the normalized image space */
pix_size[0] = 1.0 / (double)img_definition[0];
pix_size[1] = 1.0 / (double)img_definition[1];
pix_low[0] = (double)pix_idx[0] * pix_size[0];
pix_low[1] = (double)pix_idx[1] * pix_size[1];
```

Nous utilisons alors ces deux variables pour échantillonner une position 2D aléatoire au sein du pixel.

70g ⟨*local variables of the realisation* 70g⟩≡
```
double sample[2];
```

70h ⟨*sample a direction into the solid angle of the pixel* 70h⟩≡
```
sample[0] = ssp_rng_canonical(rng)*pix_size[0] + pix_low[0];
sample[1] = ssp_rng_canonical(rng)*pix_size[1] + pix_low[1];
```

On notera que nous générons ici une position aléatoire sur le plan image de la caméra défini sur $[0,1)^2$; la grille de pixels peut être vue comme une stratification de cet espace à échantillonner. Dit autrement, à l'échelle de la caméra, nous échantillonnons uniformément son plan image en échantillonnant uniformément chacune de ses strates. Nous pouvons alors invoquer la fonction `pt_camera_ray`, proposée par l'interface de programmation de la caméra (section 3.4), pour générer un rayon partant de la caméra et qui traverse le plan image à la position 2D que nous venons de calculer. Ce faisant nous échantillons uniformément une direction dans l'angle solide du pixel.

71a ⟨*local variables of the realisation* 70g⟩+≡
```
double ray_org[3];
double ray_dir[3];
```

71b ⟨*sample a direction into the solid angle of the pixel* 70h⟩+≡
```
pt_camera_ray(pt->camera, sample, ray_org, ray_dir);
```

Nous évaluons ensuite la luminance captée par le pixel le long de cette direction en invoquant la fonction `compute_radiance` définie dans la section suivante. En retour de fonction nous obtenons une luminance que nous accumulons au sein du pixel : nous incrémentons la somme de ses poids Monte-Carlo et la somme des ces mêmes poids élevés au carré. Ces deux accumulateurs nous servirons à calculer l'espérance et l'écart type de notre estimation (section 3.7.7).

71c ⟨`pt.c` *helper functions declarations* 58a⟩+≡
```
static double
compute_radiance
  (struct pt* pt,
   struct ssp_rng* rng,
   const double ray_org[3],
   const double ray_dir[3]);
```

71d ⟨*local variables of the realisation* 70g⟩+≡
```
double w; /* Monte-Carlo weight */
```

71e ⟨*compute the radiance that reaches the pixel along the camera ray* 71e⟩≡
```
w = compute_radiance(pt, rng, ray_org, ray_dir);
```

71f ⟨*add the computed radiance against the pixel accumulators* 71f⟩≡
```
pixel->sum += w;
pixel->sum2 += w*w;
pixel->nweights += 1;
```

3.7.6 Suivi de chemins

Nous allons désormais suivre à rebours le chemin lumineux qui contribue au pixel courant. Cet algorithme de suivi de chemins est mis en œuvre dans la fonction `compute_radiance` définie ci-après, fonction qui n'est en définitive que la traduction informatique de l'algorithme décrit dans la relation 3.3 de la section 3.7.1 :

71g ⟨`pt.c` *helper functions definitions* 49c⟩+≡
```
double
compute_radiance
  (struct pt* pt,
   struct ssp_rng* rng,
   const double position[3],
   const double direction[3])
{
  ⟨compute_radiance local variables 72a⟩
  double L = 0; /* Radiance */
  ASSERT(pt && rng && position && direction); /* Check input variables */
```

```
  ⟨define where the skydome is 72b⟩
  ⟨setup the starting path segment 73b⟩
  ⟨trace the path to compute the radiance that it carries 73c⟩

  return L;
}
```

Cette fonction prend comme paramètres d'entrée un pointeur vers l'application, le générateur de nombres aléatoires du processus qui trace ledit chemin, ainsi que la position et la direction de son premier segment. En sortie, elle retourne à l'appelant la luminance portée par le chemin ainsi tracé.

Situer la voûte céleste

Dans notre programme, nous décidons que la seule source lumineuse de la scène est la voûte céleste. Or, nous n'avons pour l'heure aucune donnée qui nous permette de la situer. Nous pourrions fixer arbitrairement sa position le long d'un axe mais rien ne garantit à priori que la scène soit orientée en accord avec cet axe vertical, qui n'est en définitive qu'une convention. À la place, nous pourrions demander à l'utilisateur de renseigner l'axe vertical de la scène à l'aide d'une option sur la ligne de commande. Cependant, il définit déjà un axe vertical, à savoir celui de la caméra (section 3.3.4), et il est vraisemblable que cet axe pointe bien vers le "haut" de la scène. Nous choisissons donc d'utiliser l'axe vertical de la caméra pour déterminer où se situe la voûte céleste.

La voûte que nous cherchons à représenter est un espace semi-infini aligné le long d'un axe de la scène. Dit autrement, il est défini sur $[0, \pm\infty)$ le long de l'axe X, Y ou Z. Nous déterminons parmi ces axes lequel pointe vers la voûte céleste en évaluant le demi espace vers lequel l'axe vertical de la caméra est principalement orienté.

72a ⟨`compute_radiance` *local variables* 72a⟩≡

```
double cam_up[3], cam_up_abs[3];
int skydome_axis;
```

72b ⟨*define where the skydome is* 72b⟩≡

```
pt_camera_get_up(pt->camera, cam_up);
cam_up_abs[0] = fabs(cam_up[0]);
cam_up_abs[1] = fabs(cam_up[1]);
cam_up_abs[2] = fabs(cam_up[2]);
skydome_axis = cam_up_abs[0] > cam_up_abs[1]
  ? (cam_up_abs[0] > cam_up_abs[2] ? 0/*X axis*/ : 2/*Z axis*/)
  : (cam_up_abs[1] > cam_up_abs[2] ? 1/*Y axis*/ : 2/*Z axis*/);
```

On notera que le code précédent ne fait que déterminer l'axe de la voûte céleste mais ne dit rien de son signe. Pour déterminer dans quel demi espace se trouve effectivement la voûte céleste, nous utiliserons le signe du vecteur `cam_up` en invoquant `sign(cam_up[skydome_axis])` : une valeur positive ou négative signifie que le ciel est défini dans, respectivement, le demi espace positif ou négatif le long de cet axe ; Le demi espace complémentaire étant alors assimilé aux "abîmes" dont l'éclairement est nul.

Initialisation & aperçu de la marche aléatoire

Nous déterminons désormais l'état initial de notre marche aléatoire. Nous ajoutons pour cela deux variables locales, `pos` et `dir`, qui représentent respectivement la position et la direction d'un segment du chemin. Et nous initialisons ces variables avec les paramètres d'entrée de la fonction caractérisant le premier segment du chemin. Nous définissons également la variable locale `transmissivity` qui stocke la

fraction d'énergie non atténuée le long du chemin. Pour l'heure, notre chemin n'a rencontré aucune surface et n'a donc pas encore été atténué par une réflexion en paroi ; nous initialisons donc cette variable à 1.

73a ⟨`compute_radiance` *local variables* 72a⟩+≡
```
double pos[3], dir[3];
double transmissivity = 1.0;
```

73b ⟨*setup the starting path segment* 73b⟩≡
```
d3_set(pos, position);
d3_set(dir, direction);
```

Nous écrivons désormais la boucle principale de notre fonction dans laquelle nous traçons le segment du chemin, le faisons rebondir en paroi et ce jusqu'à ce que le chemin parte à l'infini (équation 3.5).

73c ⟨*trace the path to compute the radiance that it carries* 73c⟩≡
```
for(;;) {
  ⟨trace the current path segment 73e⟩

  if(⟨the path reaches infinity 74c⟩) {
    ⟨compute the path radiance and stop the random walk 74d⟩
  }

  ⟨sample a bounce direction 75b⟩
  ⟨update the path transmissivity wrt the bounce direction 75d⟩
  ⟨setup the next path segment 76a⟩
}
```

Tracer un segment du chemin

Pour suivre un chemin dans la scène, nous commençons par tracer le segment courant de notre chemin à l'aide de la bibliothèque `Star-3D`. Dans `Star-3D` un rayon est caractérisé par son origine, sa direction et son étendue : une intersection le long du rayon ne peut avoir lieu qu'à une distance comprise dans l'intervalle défini par cette étendue. Ici, nous souhaitons lancer notre rayon à l'infini ; nous fixons donc l'étendue du rayon à $[0, \infty)$.

73d ⟨`compute_radiance` *local variables* 72a⟩+≡
```
float ray_range[2];
```

73e ⟨*trace the current path segment* 73e⟩≡
```
ray_range[0] = 0;
ray_range[1] = (float)INF;
```

Là où nous utilisons des virgules flottantes à double précision pour représenter l'origine et la direction du segment courant, les rayons de `Star-3D` sont quant à eux définis par des vecteurs flottants à *simple précision*. Nous convertissons donc les paramètres du segment à l'aide de la fonction `f3_set_d3` proposée par `RSys`. À noter que bien que la direction du segment soit normalisée, nous la re-normalisons après conversion afin d'assurer qu'elle est bien normalisée au regard de sa simple précision.

73f ⟨`compute_radiance` *local variables* 72a⟩+≡
```
float ray_org[3], ray_dir[3];
```

73g ⟨*trace the current path segment* 73e⟩+≡
```
f3_set_d3(ray_org, pos);
f3_set_d3(ray_dir, dir);
f3_normalize(ray_dir, ray_dir);
```

Au delà des variables définissant le rayon à tracer, nous passons également comme argument à la fonction de lancer de rayon l'adresse de la primitive géométrique sur laquelle se situe l'origine du rayon courant. Pendant le lancer de rayon, `Star-3D` transfère ce paramètre à la fonction de filtrage définie dans la section 3.6.8. On rappelle que cette fonction nous permet de rejeter (c'est à dire filtrer) une intersection si elle a lieu sur le triangle sur lequel repose l'origine du rayon, évitant ainsi d'intersecter le triangle dont on est censé partir. Initialement, notre rayon démarre de la caméra et n'est donc pas situé sur une quelconque primitive : nous initialisons cette variable à `S3D_PRIMITIVE_NULL`. Nous mettrons à jour sa valeur au fil des intersections du chemin avec la géométrie de la scène.

74a ⟨`compute_radiance` *local variables* 72a⟩+≡
```
struct s3d_primitive prim_from = S3D_PRIMITIVE_NULL;
struct s3d_hit hit = S3D_HIT_NULL;
```

74b ⟨*trace the current path segment* 73e⟩+≡
```
s3d_scene_view_trace_ray
  (pt->scnview, ray_org, ray_dir, ray_range, &prim_from, &hit);
```

Vers l'infini et au-delà

Le résultat de l'intersection du rayon avec la géométrie de la scène est retourné dans la variable `hit`. Cette variable matérialise l'impact du rayon avec une géométrie ou son absence. Après avoir tracé le rayon dans la scène, nous commençons donc par tester l'état de cette variable, à l'aide de la macro `S3D_HIT_NONE`, afin de déterminer si le rayon a intersecté une géométrie ou pas.

74c ⟨*the path reaches infinity* 74c⟩≡
```
S3D_HIT_NONE(&hit)
```

En l'absence d'intersection, nous avons deux possibilités : soit le rayon ainsi tracé pointe vers la voûte céleste, soit il est dirigé vers les abîmes. Dans le premier cas, la luminance en bout de chemin est alors égale à l'éclairement du ciel multiplié par la transmissivité du chemin, c'est à dire la quantité d'énergie non atténuée le long du chemin. Or, nous rappelons que dans notre programme l'éclairement du ciel est uniforme et vaut 1. Par conséquent la luminance en bout du chemin est égale à sa seule transmissivité. À contrario, s'il atteint les abîmes, la luminance du chemin est simplement nulle.

74d ⟨*compute the path radiance and stop the random walk* 74d⟩≡
```
if(sign(dir[skydome_axis]) == sign(cam_up[skydome_axis])) {
  L = transmissivity;
} else {
  L = 0;
}
break; /* Stop the random walk */
```

Rebond en paroi

Un chemin qui intersecte une géométrie va alors rebondir sur celle-ci dans une direction distribuée selon la fonction de diffusion surfacique bidirectionnelle associée à l'impact. Dans notre cas, la scène n'est composée que d'un seul matériau représenté par une fonction de réflexion lambertienne définie par la bibliothèque `Star-SF` (section 3.6.9). Pour générer la prochaine direction du chemin, nous invoquons la fonction `ssf_bsdf_sample` qui prend en argument non seulement cette fonction de diffusion et un générateur aléatoire, mais également la direction entrante et la normale à l'impact. Par convention, `Star-SF` assume que ces deux vecteurs pointent vers *l'extérieur* de la surface. Après avoir normalisé la normale à l'impact, nous nous assurons donc qu'elle est bien orientée selon l'attendu de `Star-SF` et l'inversons dans le cas contraire. Nous inversons également la direction entrante `dir` qui jusqu'alors pointe

vers la surface impactée.

75a ⟨`compute_radiance` *local variables* 72a⟩+≡
```
double dir_next[3];
double dir_temp[3];
double N[3];
double R;
```

75b ⟨*sample a bounce direction* 75b⟩≡
```
d3_set_f3(N, hit.normal);
d3_normalize(N, N);
if(d3_dot(N, dir) > 0) d3_minus(N, N);
d3_minus(dir_temp, dir);
R = ssf_bsdf_sample(pt->bsdf, rng, dir_temp, N, dir_next, NULL, NULL);
```

En sortie de `ssf_bsdf_sample`, nous obtenons la prochaine direction du chemin (`dir_next`) ainsi que la réflectance de la surface impactée en accord avec la direction entrante et la direction échantillonnée. On notera que notre matériau étant purement diffusif, la valeur de la réflectance `R` reste ici indépendante de ces deux directions. Quoiqu'il en soit, indépendamment de ce cas particulier, nous utilisons cette réflectance pour mettre à jour la transmissivité du chemin après rebond. Pour ce faire, nous avons plusieurs possibilités (équation 3.3). Nous pouvons multiplier le poids porté par le chemin par la réflectance en paroi. Nous pouvons également utiliser une roulette russe pour arrêter le chemin au prorata de cette réflectance. L'utilisation d'une roulette russe a pour principal avantage de permettre l'arrêt du chemin avant qu'il n'atteigne l'infini, accélérant dès lors notre algorithme de suivi de chemin. En contrepartie il est vraisemblable qu'elle augmente la variance de notre estimation. Nous choisissons donc de basculer en roulette russe uniquement quand la transmissivité du chemin est "faible" au regard d'un seuil que nous fixons ici arbitrairement à 0.1

Dans la section 3.7.1, nous avons vu que la probabilité qu'un chemin atteigne une source lumineuse peut être quasi ou strictement nulle. C'est notamment le cas si la caméra est positionnée dans une cavité fermée. Dans cette situation, avec une réflectance en paroi inférieure à 1, la roulette russe sur cette même réflectance permet là encore d'arrêter ce type chemin qui porte quoiqu'il en soit une luminance nulle. Cependant, avec une réflectance en paroi de 1, cette roulette russe ne nous est plus d'aucune utilité et les chemins rebondissent alors à l'infini. Pour palier cet écueil, nous ajoutons une nouvelle roulette russe que nous activons quand le nombre de rebonds en paroi dépasse un certain seuil (ici fixé arbitrairement à 1000). Dès lors, nous décidons une fois sur deux soit d'arrêter le chemin concerné, soit de multiplier son poids par 2 (équation 3.5).

75c ⟨`compute_radiance` *local variables* 72a⟩+≡
```
int nbounces = 0;
```

75d ⟨*update the path transmissivity wrt the bounce direction* 75d⟩≡
```
if(transmissivity < 0.1) { /* Russian roulette on surface reflectivity */
  if(ssp_rng_canonical(rng) >= R) break;
} else {
  transmissivity *= R;
  if(nbounces > 1000) { /* Russian roulette to stop long paths */
    if(ssp_rng_canonical(rng) > 0.5) {
      break;
    } else {
      transmissivity *= 2;
    }
  }
}
nbounces++;
```

Reste alors à mettre à jour le segment du chemin après rebond. Nous commençons par positionner l'origine du segment à l'intersection du segment précédent avec la géométrie. Nous initialisons ensuite sa direction avec la direction que l'on vient d'échantillonner en paroi. Enfin, nous affectons à la variable `prim_from` l'identifiant du triangle impacté par le segment précédent, et sur lequel repose l'origine du nouveau segment. Ce triangle sera dès lors ignoré lors du prochain lancer de rayon à l'aide de la fonction de filtrage d'impact, évitant ainsi que le rayon intersecte le triangle dont il est censé partir.

76a ⟨*setup the next path segment* 76a⟩≡
```
pos[0] = pos[0] + dir[0] * hit.distance;
pos[1] = pos[1] + dir[1] * hit.distance;
pos[2] = pos[2] + dir[2] * hit.distance;
d3_set(dir, dir_next);
prim_from = hit.prim;
```

3.7.7 Écrire l'image en sortie

Une fois l'image rendue, nous pouvons désormais la stocker dans le fichier en sortie. Nous avons décidé d'écrire cette image dans le format PGM et pour cela nous utilisons la fonction `write_image_pgm` définie ci-après.

76b ⟨*write the image to the output file* 76b⟩≡
```
write_image_pgm(pt);
```

76c ⟨`pt.c` *helper functions definitions* 49c⟩+≡
```
static void
write_image_pgm(struct pt* pt)
{
  ⟨write_image_pgm local variables 76d⟩
  ASSERT(pt);

  ⟨write PGM header 76e⟩
  ⟨write PGM pixels 77b⟩
}
```

Le format PGM se décompose en deux parties : un en-tête décrivant la structure de l'image, et la liste des pixels la composant. Nous commençons donc tout d'abord par écrire l'en-tête de l'image.

76d ⟨`write_image_pgm` *local variables* 76d⟩≡
```
size_t img_size[2];
```

76e ⟨*write PGM header* 76e⟩≡
```
img_size[0] = pt_image_get_width(pt->image);
img_size[1] = pt_image_get_height(pt->image);
fprintf(pt->output, "P2 %lu %lu\n",
  (unsigned long)img_size[0],
  (unsigned long)img_size[1]);
fprintf(pt->output, "255\n"); /* Max value */
```

L'identifiant `P2` notifie que la couleur des pixels est représentée par une chaîne de caractères et non par une donnée binaire. Sur la même ligne nous stockons aussi la définition de l'image. Nous écrivons ensuite sur la ligne suivante la valeur entière maximale qu'un pixel peut prendre ; une valeur de 255 signifie que la couleur d'un pixel peut être stockée sur 8-bits.

Nous écrivons alors la couleur de chaque pixel dans le fichier de sortie. Le format PGM assume que les pixels de l'image sont triés par ligne. C'est pourquoi nous itérons d'abord sur les lignes puis sur les colonnes de l'image. Pour chaque pixel, nous commençons par calculer l'espérance de sa luminance à partir de l'estimation réalisée par notre algorithme de rendu. Nous convertissons ensuite cette luminance en couleur que nous écrivons enfin dans le fichier de sortie.

77a ⟨`write_image_pgm` *local variables* 76d⟩+≡
```
size_t x, y;
```

77b ⟨*write PGM pixels* 77b⟩≡
```
FOR_EACH(y, 0, img_size[1]) {
  FOR_EACH(x, 0, img_size[0]) {
    ⟨compute the estimated pixel radiance 77d⟩
    ⟨gamma correct the pixel radiance 77e⟩
    ⟨encode and write the pixel color to the output file 77f⟩
  }
}
```

L'espérance de la luminance du pixel s'évalue simplement en moyennant les poids Monte-Carlo accumulés dans le pixel lors de l'estimation de sa luminance (section 3.7.5).

77c ⟨`write_image_pgm` *local variables* 76d⟩+≡
```
struct pt_pixel* pixel;
double L;
```

77d ⟨*compute the estimated pixel radiance* 77d⟩≡
```
pixel = pt_image_at(pt->image, x, y);
ASSERT(pixel->nweights);
L = pixel->sum / (double)pixel->nweights;
```

Nous convertissons alors cette luminance en couleur. À cette étape, nous pourrions appliquer une correspondance tonale entre la luminance et la couleur du pixel à stocker. Cette correspondance consiste principalement à déterminer la dynamique des luminances de l'image que l'on souhaite *in fine* afficher. Notre moteur de rendu ne calcule cependant que des luminances entre $[0, 1]$. Une telle plage de valeurs est bien représentée sur 8-bits et par conséquent, comme nous ne souhaitons pas sur-exposer les couleurs de l'image, nous n'appliquons tout simplement pas d'opérateur de mappage tonal à la luminance du pixel.

Ceci étant, la luminance que nous avons calculée est définie dans un espace de couleurs *linéaire* là où il est communément admis qu'une image stocke ses couleurs dans un espace *non linéaire*, c'est à dire après qu'une correction de gamma y ait été appliquée. Ne nous discuterons pas plus avant de ce travail sur les couleurs qui dépasse largement le cadre de ce programme. Nous nous contentons simplement d'appliquer ici une correction de gamma conventionnelle afin de répondre à l'attendu des logiciels de visualisation d'images. Nous assimilons donc notre luminance à une couleur RGB, dont les trois composantes ont la même valeur, et la convertissons dans l'espace de couleurs sRGB comme suit :

77e ⟨*gamma correct the pixel radiance* 77e⟩≡
```
if(L < 0.0031308) {
  L = L * 12.92;
} else {
  L = 1.055 * pow(L, 1.0/2.4) - 0.055;
}
```

Nous encodons enfin notre couleur dans un octet non signé que nous écrivons enfin dans le fichier de sortie comme couleur du pixel courant.

77f ⟨*encode and write the pixel color to the output file* 77f⟩≡
```
L = CLAMP(L, 0, 1);
fprintf(pt->output, "%u\n", (unsigned char)(L * 255.0));
```

3.8 Fonction principale

Le programme que nous développons est un exécutable en ligne de commande qui par conséquent doit proposer une fonction `main`, c'est à dire une fonction principale qui sera appelée à son lancement et qui pilote l'exécution du programme. C'est en définitive la dernière partie de notre programme qu'il nous reste à développer. Cette fonction principale retourne un entier notifiant si son exécution s'est bien passée et prend deux paramètres d'entrée : un entier dénombrant le nombre d'arguments sur la ligne de commande et un tableau de chaîne de caractères listant ces arguments.

78a ⟨*src/pt_main.c* 78a⟩≡

```
⟨licensing 83a⟩

⟨pt_main.c inclusions 78c⟩

int
main(int argc, char** argv)
{
  ⟨main local variables 78d⟩
  res_T res = RES_OK; /* Temporary variable used to check return status */
  int err = 0; /* Error code. 0 means that no error occurs */

  ⟨setup the application arguments 78e⟩
  ⟨init the application 79c⟩
  ⟨run the application 79d⟩

exit:
  ⟨clean up the initialised data 79e⟩
  ⟨check memory leaks 79h⟩
  return err;
error:
  err = -1; /* An error occurs */
  goto exit;
}
```

78b ⟨*list of source files* 46c⟩≡

```
pt_main.c
```

Dans cette fonction, nous commençons par analyser les arguments passés à la ligne de commande. Pour ce faire nous utilisons l'interface de programmation des arguments développés dans la section 3.3. En cas d'erreur lors de l'analyse des arguments (option inconnue, argument invalide, *etc.*), nous quittons notre programme en retournant une erreur. Nous vérifions également si le programme doit s'arrêter après l'analyse des arguments et le cas échéant nous le quittons sans retourner d'erreur ; son exécution étant ici simplement terminée. C'est notamment le cas lorsque l'option `-h` est utilisée. Cette option notifie que l'utilisateur souhaite afficher l'aide de la ligne commande sans effectuer le moindre traitement (section 3.3.8).

78c ⟨`pt_main.c` *inclusions* 78c⟩≡

```
#include "pt_args.h"
```

78d ⟨`main` *local variables* 78d⟩≡

```
struct pt_args args = PT_ARGS_DEFAULT;
```

78e ⟨*setup the application arguments* 78e⟩≡

```
res = pt_args_init(&args, argc, argv);
if(res != RES_OK) goto error;
if(args.quit) goto exit;
```

Nous utilisons ensuite l'interface de programmation de l'application (section 3.6) pour initialiser son traitement à partir des arguments que nous venons d'analyser.

79a ⟨`pt_main.c` *inclusions* 78c⟩+≡
```
#include "pt.h"
```

79b ⟨`main` *local variables* 78d⟩+≡
```
struct pt pt = PT_NULL;
```

79c ⟨*init the application* 79c⟩≡
```
res = pt_init(&args, &pt);
if(res != RES_OK) goto error;
```

Nous pouvons alors lancer le traitement de l'application en tant que tel.

79d ⟨*run the application* 79d⟩≡
```
res = pt_run(&pt);
if(res != RES_OK) goto error;
```

Nous libérons alors les arguments et l'application précédemment initialisés.

79e ⟨*clean up the initialised data* 79e⟩≡
```
pt_args_release(&args);
pt_release(&pt);
```

Enfin nous vérifions que l'ensemble de l'espace mémoire alloué à l'aide des allocateurs de la bibliothèque `RSys` a été effectivement libéré. Nous utilisons pour ce faire la fonction `mem_allocated_size` que retourne le nombre d'octets encore alloués. S'il n'est pas nul, nous affichons un message d'erreur avant d'affecter au code de retour un entier différent de zéro, notifiant qu'une erreur s'est produite.

79f ⟨`pt_main.c` *inclusions* 78c⟩+≡
```
#include <rsys/mem_allocator.h>
#include <stdio.h>
```

79g ⟨`main` *local variables* 78d⟩+≡
```
size_t memsz;
```

79h ⟨*check memory leaks* 79h⟩≡
```
memsz = mem_allocated_size();
if(memsz != 0) {
  fprintf(stderr, "Overall memory leaks: %lu Bytes\n", (unsigned long)memsz);
  err = -1; /* Notify a memory leak error */
}
```

Chapitre 4

Rétrospective

La programmation lettrée révolutionne le regard porté par les programmeurs sur leurs actes de développement. Avant de conclure, nous discutons ici de ce paradigme de programmation tel que nous la percevons suite à l'écriture de ce document, et des outils de développement qui permettent de le pratiquer.

4.1 La programmation lettrée

Il est admis qu'un programme lettré est bien plus laborieux à rédiger que le même programme développé de manière conventionnelle, et l'écriture de ce document ne vient que renforcer cette évidence. Ce travail supplémentaire se justifierait par une maintenance et une évolution facilitées du programme ainsi rédigé. En première analyse, cet argument nous semble difficile à défendre tant un programmeur lit les sources d'un programme de manière non linéaire, sautant d'un morceau de code à un autre selon le seul axe de lecture que lui seul définit à l'instant T. Or, un programme lettré se veut au contraire plus rigide, proposant une lecture linéaire du code source en l'inscrivant dans le seul discours porté par les auteurs. L'écrit peut donc au contraire venir alourdir la lecture d'un code lors de sa maintenance et de son évolution. L'intérêt de la programmation lettrée ne nous semble donc pas à rechercher d'abord dans des gains de productivité en devenir mais davantage dans ce que permet, ici et maintenant, l'articulation d'un code dans un texte structuré.

Et bien plus qu'un moteur de rendu, le présent document est avant tout un *écrit* qui cherche à transférer un savoir matérialisé *in fine* dans des lignes de code. Ce faisant, l'écriture de ces lignes s'insère avant tout dans une logique d'explicitation radicale au regard de l'objectif et du public visé. La programmation lettrée se veut donc particulièrement exigeante dans le sens où elle pousse les auteurs à ciseler leurs choix de conception et de développement dans le but premier de *transmettre* et de *former* ; le code source n'étant en définitive que la traduction d'un savoir soutenu par l'écrit. Loin d'invalider l'idée d'une maintenance et d'une évolution facilitée, nous décalons ici la représentation qu'on s'en fait habituellement. Car un programme lettré ne semble pas particulièrement plus simple à maintenir ou à faire évoluer, les auteurs devant garder en cohérence et son code source et le texte qui le structure. Cependant, en se rendant accessible à un public bien plus large que ses seuls auteurs, il élargit la communauté de programmeurs en capacité de se l'approprier et donc de le modifier et de l'étendre.

4.2 Rédiger un programme avec `noweb`

Bien qu'il ne soit plus en développement, `noweb` reste néanmoins un solide outil de programmation lettrée. Sa licence libre lui assure un certain niveau de pérennité, nous laissant la pleine responsabilité de le faire évoluer au besoin, ce qui, après une brève lecture de son code source, est loin d'être impensable. Par ailleurs, son inscription dans la pensée UNIX nous a permis de l'intégrer facilement dans une chaîne d'outils construite autour de cette logique. Et il a été tout aussi naturel de remplacer son *backend* pour modifier son comportement et ainsi faciliter l'écriture du présent document. Tant est si bien qu'une fois

l'environnement de développement mis en place, nous avons rédigé notre programme en ayant comme seule idée que nous écrivions du LaTeX et du C.

noweb s'est tout de même rappelé à nous, et notamment en ce qui concerne l'utilisation du caractère '_' dans les noms identifiant les blocs de code. LaTeX oblige, chaque *underscore* devait être précédé par le caractère d'échappement '\', nécessitant alors un traitement particulier lors de l'extraction par notangle des blocs de code en question. Plus gênante reste l'impossibilité d'utiliser le mécanisme de définition proposé par noweb avec des variables ou des fonctions contenant un ou plusieurs *underscore* : le source LaTeX en résultant étant dès lors invalide. Nous pensons cependant qu'une modification du paquet LaTeX de noweb devrait corriger cet écueil.

4.3 Conclusion

En venant enrichir la technicité d'un programme avec une dimension littéraire, la programmation lettrée est quoiqu'il en soit un puissant paradigme de programmation qui articule un code avec le savoir et la pensée qui le sous-tendent. Et noweb permet d'adopter ce paradigme pour rédiger des programmes qui sortent de la seule preuve de concept. Il est donc tout à fait possible aujourd'hui d'utiliser au quotidien une telle approche de la programmation qui n'en demeure pas moins très exigeante. Exigeante dans l'acte de développement qui loin de n'être plus qu'un geste technique s'articule désormais avec un discours. Exigeante enfin de par son caractère lettré qui subvertit l'acte de programmation en une pratique d'écriture. Le travail nécessaire à la rédaction d'un tel écrit est évidemment d'une toute autre intensité, expliquant sans doute que la programmation lettrée reste aujourd'hui marginale dans une communauté informatique dont l'objet est avant tout de traiter des données et d'effectuer des calculs ; bien loin des préoccupations de transmission de savoir et de formation qui restent le cœur de la programmation lettrée.

Annexe A

Licences

Dans cette annexe nous listons les en-têtes de fichier qui notifient les détenteurs des droits patrimoniaux sur le code ainsi que la licence qui le régit. Une même licence est parfois présente plusieurs fois, la seule différence étant les caractères de commentaire utilisés pour insérer cet en-tête dans les différents fichiers.

Seules deux licences différentes sont utilisées dans les codes source que nous développons ici. La quasi totalité des codes sources, y compris les sources du présent document, sont licenciés sous la *GNU General Public License* version 3 [GPL] dont l'en-tête `C` est le suivant.

83a ⟨*licensing* 83a⟩≡

```
/* Copyright (C) 2019 |Meso|Star> (vincent.forest@meso-star.com)
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program. If not, see <http://www.gnu.org/licenses/>. */
```

Une copie de cette licence est distribuée avec les sources de ce programme. Vous pouvez également la consulter à l'url `https://www.gnu.org/licenses/gpl-3.0.html`. En plus de l'en-tête pour le langage `C`, nous listons ci-après ce même en-tête qui utilise le caractère '#' comme caractère de commentaire afin de pouvoir l'insérer dans un fichier de configuration `CMake` ou dans un script `Bash`.

83b ⟨*licensing formatted with # comments* 83b⟩≡

```
# Copyright (C) 2019 |Meso|Star> (vincent.forest@meso-star.com)
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
```

```
# You should have received a copy of the GNU General Public License
# along with this program. If not, see <http://www.gnu.org/licenses/>.
```

Enfin nous utilisons également la *GNU All-Permissive licence* qui, comme son nom l'indique, est particulièrement laxiste. Nous appliquons cette licence uniquement sur des petits scripts `Bash`.

84 ⟨*GNU All-Permissive license* 84⟩≡

```
# Copyright (C) 2019 |Meso|Star> (vincent.forest@meso-star.com)
#
# Copying and distribution of this file, with or without modification, are
# permitted in any medium without royalty provided the copyright notice and
# this notice are preserved. This file is offered as-is, without any warranty.
```

Bibliographie

[ANS90] ANSI/ISO. ISO C Standard 1990. Technical report, 1990.

[CC-] Creative Commons Non-Commercial 3.0. `https://creativecommons.org/licenses/by-nc/3.0/us/`.

[CDF+] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, , and R. Thayer. OpenPGP Message Format. RFC 4880 (Proposed Standard). Updated by RFC 5581.

[CeC] CeCILL v2.1. `https://cecill.info/licences/Licence_CeCILL_V2.1-fr.html`.

[DM98] Leonardo Dagum and Ramesh Menon. OpenMP : an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1) :46–55, 1998.

[GPL] GNU General Public License v3.0. `https://www.gnu.org/licences/gpl.html`.

[Gro18] The Open Group. Utility Conventions. In *The Open Group Base Specifications Issue 7, 2018 edition IEEE Std 1003.1-2017*, chapter 12. 2018.

[GWA16] Christiaan Gribble, Ingo Wald, and Jefferson Amstutz. Implementing Node Culling Multi-Hit BVH Traversal in Embree. *Journal of Computer Graphics Techniques (JCGT)*, 5(4) :1–7, November 2016.

[Han96] David R. Hanson. *C Interfaces and Implementations : Techniques for Creating Reusable Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[Hil91] David Hilbert. *Math. Ann.*, volume 38, chapter Über die stetige Abbildung einer Linie auf ein Flächenstück, pages 459–460. 1891.

[Ker88] Brian W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[Knu84] Donald Knuth. Literate Programming. *The Computer Journal*, 27 :97–111, 1984.

[Mor66] G. M. Morton. A computer oriented geodetic data base ; and a new technique in file sequencing. Technical report, 1966.

[PJH16] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering : From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2016.