Utilisation de la ligne de commande et compilation

1 Organisation des fichiers dans un ordinateur

1.1 Stockage persistant des données

Un ordinateur étant un appareil destiné au traitement de données, nous avons déjà eu l'occasion de souligner le besoin de mémoriser à moyen et long terme des informations, qu'elles soient des données, des programmes, etc. Les moyens de stockage sont variés. Si l'on exclut le stockage en réseau qui peut prendre différente formes, il est possible de conserver des données sur des disques durs, des disques optiques, des bandes magnétiques, des clés USB, etc.

Bien vite, il y a une quantité gigantesque de données sur un disque. Pour s'y retrouver, les données sont regroupées en *fichiers*, correspondant à un programme, un texte, un enregistrement audio, une photographie, des données numériques expérimentales, etc. Pour mieux s'y retrouver, on attribue à chaque fichier un nom.

L'ordinateur mémorise où se trouvent, physiquement, les bits associés à chacun des fichiers. Que ce soit sur un disque optique de type DVD, sur le plateau magnétique d'un disque dur, ou bien encore dans quelle cellule électrique d'une clé USB. Pour ce faire, il utilise une *table d'allocation*, enregistrée sur le média, qui mémorise les noms des fichiers et les emplacements physiques des données associées. Cette table d'allocation, indispensable au bon fonctionnement du média, est crée lors du *formattage* du média.

1.2 Arborescence des répertoires et fichiers

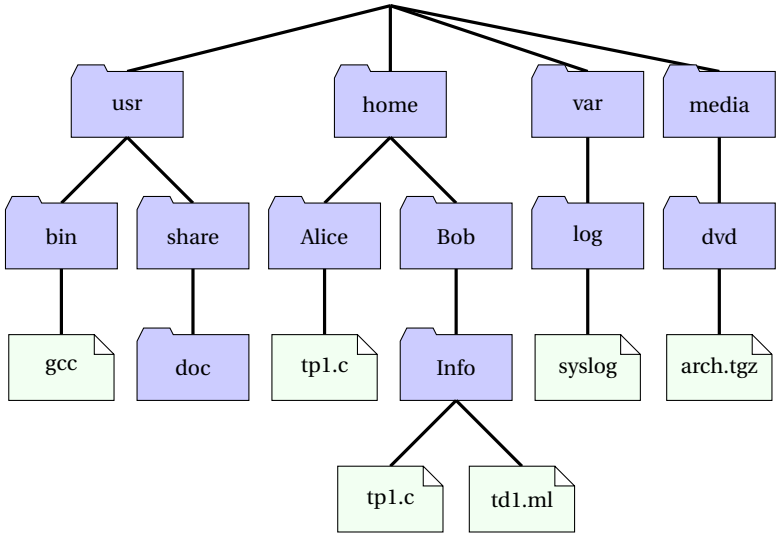
Vu le nombre conséquent de fichiers [1], le seul nom ne permettrait pas de s'y retrouver, et on pourrait être tenté de mettre le même nom à des fichiers différents, ce qui causerait des problèmes. Une organisation de ces fichiers est nécessaire. La quasi-totalité du temps, l'organisation retenue est une structure arborescente. Les fichiers sont regroupés dans des *dossiers*, qui eux-mêmes peuvent être regroupés dans d'autres dossiers, et ainsi de suite.

Dans l'écosystème Linux, un petit morceau de l'organisation des fichiers sur le disque ressemble à ce qui est représenté ci-après. Les programmes mis à disposition de tous les utilisateurs, tel que le compilateur `gcc`, sont généralement placés dans le répertoire `bin`, lui-même inclus dans le répertoire `usr` qui regroupe les ressources mises à disposition des utilisateurs de la machine. Les différents disques et média apparaissent comme des répertoires dans l'arborescence.

Chaque utilisateur de la machine possède son propre répertoire, à son nom, dans le répertoire `home`, de façon à ce qu'il puisse y placer ses propres fichiers, par exemple un fichier source C `tp1.c` pour Alice.

1. Approchant souvent le milion sur un ordinateur « normal ».

Le *chemin*, en partant de la « racine » de cette structure arborescente (tout en haut) menant à ce fichier `tp1.c` est donc home ⟼ Alice ⟼ tp1.c. Sous Unix, on sépare les éléments par le caractère `/`, aussi le fichier peut être désigné par « `/home/Alice/tp1.c` ». C'est ce que l'on appelle un chemin `absolu` vers le fichier. De même, « `/usr/bin/gcc` » est un chemin absolu vers le fichier exécutable du compilateur `gcc`.



Il n'est pas toujours très pratique d'utiliser des chemins absolus, donc beaucoup de programmes (y compris les terminaux permettant l'utilisation de la *ligne de commande*) mémorisent une « position » dans la structure de l'arbre. Si l'on se trouve dans le dossier `home`, par exemple, on pourra faire référence au fichier précédent par « `Alice/tp1.c` ». Et simplement par « `tp1.py` » si l'on se trouve dans le dossier `Alice`. Ce sont des chemins *relatifs*. On remarquera l'absence de `/` au début des chemins relatifs.

Il est possible, pour un chemin relatif, de remonter l'arborescence : « `..` » désigne le répertoire *parent*. Ainsi, si l'on se trouve dans le répertoire `Alice`, on peut accéder au fichier `td1.ml` de Bob avec le chemin relatif « `../Bob/Info/td1.ml` ».

1.3 Types de fichiers et extensions

Un fichier peut contenir des données de types très différents (texte, son, images, programmes...) Souvent, les fichiers ne contiennent que les données proprement dites, et pas d'information sur le type de contenu, même s'il est parfois possible de le deviner en examinant le contenu du fichier.

Pour faciliter l'identification des différents contenus, il est d'usage de terminer le nom du fichier par une *extension*, un point suivi d'un code de quelques lettres donnant un indice sur le contenu du fichier (et, par conséquent, sur les applications qui pourraient interagir avec le fichier en question). Ainsi, un fichier nommé `donnees.mp4` laisse penser que le contenu est une vidéo MPEG, un nom `donnees.wav` suggère un son, `données.txt` un fichier texte, etc[2].

L'extension usuelle pour un fichier source C est « .c », celle pour un fichier source Caml « .ml ». Les exécutables sont généralement créés sans extension. Outre le fait de faciliter la vie à l'utilisateur, les extensions sont utilisées par certains éditeurs (Emacs, Scite, Vim, etc.) pour identifier le contenu de ficher et permettre la coloration syntaxique de son contenu, voire l'activation de menus et d'actions spécifiques.

1.4 Droits d'accès

Très rapidement, plusieurs personnes se sont retrouvées amenées à utiliser un même ordinateur. Comme vous ne voulez pas qu'une autre personne que vous puisse supprimer un de vos fichiers, par erreur ou malveillance, ou même soit en mesure de les regarder (voire même de savoir qu'ils existent), il a fallu introduire un système de droits d'accès aux différents fichiers et dossiers.

En fait, même si vous êtes seul à utiliser l'ordinateur, la gestion des droits d'accès peut être très utile : par exemple, vous pouvez protéger un fichier du système contre un effacement par erreur. Ou bien d'empêcher un virus ou un ver que vous auriez exécuté involontairement de modifier des fichiers systèmes (raison pour laquelle il est déconseillé de travailler sur un ordinateur avec des droits d'administrateur de la machine).

Ces systèmes de droit d'accès peuvent être complexes, aussi prendrons nous l'exemple des droits d'accès les plus élémentaires d'Unix. Pour chaque fichier, l'ordinateur conserve des informations supplémentaires, et notamment

- à quel utilisateur appartient le fichier (Alice ou Bob par exemple) ;
- à quel groupe d'utilisateurs il est associé (on peut créer un groupe MP2I, ou LV2Allemand par exemple[3], et associer le fichier à ce groupe) ;
- les droits accordés au propriétaire du fichier ;
- les droits accordés membres du groupe ;
- les droits accordés aux autres utilisateurs.

En général, ces droits sont de trois types :

- le droit de lire le fichier (noté `r` pour *read*)
- le droit de modifier le fichier (noté `w` pour *write*)
- le droit d'exécuter le fichier (noté `x` pour *execute*)

2. Un réglage par défaut de Windows discutable peut « cacher » ces extensions dans l'explorateur de fichier (mais s'en sert pour sélectionner l'icone qui désigne le fichier), ce qui ne manque pas de causer des surprises, notamment des fichiers qui se retrouvent avec deux extensions lorsque l'on tente de les renommer.

3. Un même utilisateur peut être membre de plusieurs groupes.

Par exemple, si le fichier est décrit comme

```
- rw- r-- ---  Alice  MP2I  112237  Sep  9 09:04  tp1.c
```

alors Alice peut lire et modifier le fichier, les membres du groupe MP2I peuvent lire le fichier mais pas le modifier, et les autres utilisateurs ne peuvent ni lire ni modifier le fichier.

Les dossiers ont des droits très similaires : le droit en lecture permet de savoir quels sont les fichiers et dossiers contenus dans le dossier en question, le droit en écriture permet de créer des fichiers ou des dossiers à cet endroit, et le droit en exécution autorise à y entrer (sans le droit en exécution, on ne peut même pas se promener dans l'arborescence : par exemple si Bob retire au groupe et aux autres utilisateurs le droit en exécution du répertoire Bob, la branche de l'arborescence en question devient inaccessible à tous excepté lui-même).

2 Utilisation de la ligne de commande

2.1 Utilisation du terminal

Lorsque l'on travaille sur un ordinateur, il est pratiquement impossible de ne pas être confronté à un moment ou un autre à un *terminal*. Il s'agit d'une application particulière qui permet de communiquer avec l'ordinateur via la *ligne de commande*. On y entre des commandes sous forme de texte, et l'ordinateur y répond de même. Malgré le développement des interfaces graphiques, le terminal reste un outil incontournable car il permet d'effectuer de très nombreuses tâches très rapidement et simplement.

Même si la ligne de commande a un comportement assez basique, on dispose quand même de quelques facilités : en utilisant la flèche vers le haut, on peut retrouver des commandes que l'on a entrées précédemment. Et via la touche *tabulation*, on dispose d'un outil de complétion automatique capable de s'adapter au contexte. Dans la suite, nous allons voir quelques commandes permettant d'exploiter au mieux cet environnement.

2.2 Navigation dans l'arborescence

La commande « `pwd` » permet de savoir où l'on se trouve dans l'arborescence. Pour connaître le contenu du répertoire courant (fichiers et sous-répertoires), on utilisera la commande « `ls` » (**ls**t). On peut également demander le contenu d'un répertoire spécifique en faisant suivre `ls` d'un chemin (relatif ou absolu).

Les commandes UNIX/Linux acceptent généralement de nombreuses options. Celles-ci sont généralement passées comme une séquence d'un ou plusieurs caractères, généralement précédées du symbole « `-` ». Pour obtenir davantage d'informations sur les fichiers et dossiers, on peut utiliser l'option « `-l` » de `ls`. On y voit le propriétaire des fichiers et dossiers, la date de dernière modification, la taille et les permissions sur le fichier. Les répertoires sont identifiables par un « `d` » en début de ligne.

On peut également utiliser l'option « `-a` » pour voir les fichiers qui sont normalement masqués (sous Linux, les fichiers et répertoires commençant par un point sont par défaut cachés car ils servent notamment de fichiers de configuration). Avec la plupart des commandes Linux, on peut « regrouper » les options à une seule lettre, et ainsi écrire « `ls -la` » plutôt que « `ls -l -a` »

La commande « `find` » permet de lister tous les fichiers dans le répertoire indiqué (ou le répertoire courant sinon) et dans tous les sous-répertoires. Elle dispose de nombreuses options pour filtrer les fichiers afin de localiser un fichier spécifique que l'on cherche.

Pour se déplacer dans l'arborescence, on utilise la commande `cd` (**c**hange **d**irectory). On peut lui fournir un chemin (relatif ou absolu) ou bien « `-` », qui permet de revenir au répertoire précédent (on peut ainsi aisément faire des allers-retours entre deux répertoires). « `cd` » enfin, sans argument, permet de revenir dans son répertoire personnel.

2.3 Obtenir de l'aide

On aura tôt fait d'oublier la moitié des options des différentes commandes. Fort heureusement, un système d'aide est disponible. Il suffit d'utiliser la commande « `man` » (**man**ual) suivi du nom de la commande pour laquelle on souhaite des informations, par exemple « `man ls` ». Les informations fournies par `man` sont parfois assez succintes et réduites au plus important. On peut parfois disposer d'une commande `info` qui fonctionne de la même façon et donne généralement davantage d'informations.

2.4 Utilisation des fichiers

Pour obtenir le contenu d'un fichier, on dispose de la commande « `cat` ». Attention si le fichier est long ou contient des données binaires ! Si on souhaite le parcourir visuellement, on peut utiliser la commande `less` qui permet de monter/descendre dans le fichier et dispose de quelques fonctions élémentaires de recherche par exemple (on sort de la visualisation avec la touche `q`).

Pour examiner le contenu d'un fichier binaire, on préférera la commande `hexdump` à `cat`. L'option `-C` permet d'avoir un affichage plus facile à lire.

Une des notions importantes de la ligne de commande Linux est que les commandes peuvent être chaînées. Le symbole « `|` » permet de transférer le résultat d'une commande à une autre (on parle souvent de « pipe »). Ainsi, pour étudier à loisir le contenu d'un fichier binaire, on peut écrire « `hexdump -C monfichier | less` ».

On dispose aussi de « `>` » pour rediriger le résultat d'une commande vers un fichier (qui sera créé pour l'occasion (attention, si le fichier existe, il sera effacé !). Par exemple, lorsque l'on écrit « `ls > contenu.txt` » crée un fichier texte `contenu.txt` avec la liste des fichiers du répertoire courant obtenu avec la commande `ls` (qui n'affichera donc rien à l'écran).

De même, « `>>` » permet d'ajouter les données *à la fin* du fichier plutôt que de remplacer les données existantes.

2.5 Manipulation de répertoires et fichiers

Pour créer un nouveau répertoire, on peut utiliser la commande « `mkdir` » (**ma**k**e** **d**irectory) suivi d'un nom valide pour un répertoire. Le répertoire est créé comme sous-répertoire du répertoire courant (ou à l'endroit demandé si on a spécifié un chemin).

Pour supprimer un répertoire, on dispose de la commande « `rmdir` » (**re**m**o**ve **d**irectory), mais celui-ci doit être vide. Pour supprimer un fichier, on dispose de la commande « `rm` » (**re**m**o**ve), suivie du nom du fichier (et éventuellement du chemin qui y mène). À utiliser évidemment avec prudence !

La commande « `mv` » (**m**o**v**e) sera à la fois employée pour déplacer un fichier et pour le renommer. La commande « `cp` » (**co**p**y**) permettra de le copier. Dans les deux cas, les commandes prennent deux arguments, le fichier à déplacer/renommer/copier et le nom du nouveau fichier.

Pour changer les permissions, on utilise la commande `chmod` suivi d'un code indiquant les modifications de permissions à effectuer et du nom du fichier. Par exemple, le code « `a-w` » signifie on retire (le `-`) à tous (`a` pour **a**ll) le droit en écriture (`w` pour **w**rite). On peut modifier le groupe auquel appartient le fichier avec `chgrp`, et la personne à qui appartient le fichier avec `chown`.

2.6 Autres outils utiles

La commande « `wc` » (**w**ord **c**ount) permet d'obtenir des statistiques (nombre de caractères, de mots, de lignes) d'un fichier spécifié. Il est fréquemment utilisé avec un « pipe ». Par exemple, « `ls | wc` » permet aisément de compter le nombre de fichiers et dossiers dans le répertoire courant.

La commande « `grep` » permet d'identifier les lignes d'un fichier contenant un « mot » ou un motif spécifié. Par exemple, « `grep include monfichier` » permet de trouver toutes les lignes du fichier `monfichier` contenant « include ». Lui aussi est fréquemment combiné : « `find / | grep TP` » liste tous les fichiers sur le disque dont le nom contient « TP ». La commande a de nombreuses options, que vous découvrirez selon vos besoins.

La commande « `tar` » permet de créer et d'ouvrir des archives éventuellement compressées (ce sont des fichiers généralement d'extention `tar` ou `tgz`). Il s'agit du format le plus courant quand il s'agit de distribuer des fichiers dans le monde Unix. Il existe de nombreux paramètres pour la commande `tar`, pour l'instant on se contentera de la commande « `tar xvz arch.tgz` » qui permet d'extraire le contenu de l'archive `arch.tgz` dans le répertoire courant (e**x**tract, **v**erbose, pour un**z**ip et **f**ile : on lui demande d'extraire le contenu du fichier et de le décompresser en détaillant ce qu'il fait).

Il est impossible de lister ici toutes les commandes utiles, mais tout est fait pour permettre d'effectuer à peu près n'importe quelle tâche depuis la ligne de commande. On pourra s'intéresser à `echo`, `touch`, `sed`, `awk`, `head`, `tail`, `watch`, `split`, `join`, `paste`, `cut`, `sort`, `uniq`, `diff`, `xargs`, `finger`…

2.7 Récupération de données sur le réseau

`curl` est un outil permettant de transférer des données identifiées par une URL. C'est une véritable trousse à outil qui peut servir à de très nombreux usages différents. Nous l'emploierons à la fois comme outil de téléchargement en ligne de commande que comme mini-navigateur.

« `curl -O https://chemin/vers/un/document` » va récupérer le document dont l'URL est spécifiée et le sauvegarder dans le répertoire courant. Si l'on omet l'option « `-O` » (O en capitale), le fichier sera directement retourné plutôt que d'être sauvegardé. Par défaut, il apparaîtra dans le terminal, mais il est possible de le rediriger vers un autre outil.

Il existe quelques services destinés à être directement utilisés grâce à `curl`. Par exemple, pour connaître le temps qu'il fait à Paris et les prédictions sur trois jours, vous pouvez simplement taper « `curl wttr.in/Paris` »!

Le service le plus utile est certainement `cheat.sh` car il fournit une documentation importante sur les commandes Unix et les langages de programmation courants. Par exemple, pour obtenir des informations sur la commande `tar` avec des exemples, on peut utiliser « `curl cheat.sh/tar` ». De façon plus élaborée, si vous voulez savoir comment lire un fichier binaire en C, vous pouvez essayer « `curl cheat.sh/C/open+binary+file` ».

On dispose également d'un outil appel « `wget` » qui supporte moins de protocoles que `curl` mais dispose d'autres fonctionnalités, comme la possibilité de récupérer des sites entiers d'un seul coup.

3 Compilation de programmes C

3.1 Un premier exemple

Nous allons à présent télécharger une source en C, tenter de la compiler et de l'exécuter depuis la ligne de commande.

1. Si ce n'est fait, ouvrez un terminal et naviguez vers votre répertoire personnel. Vous pouvez, si vous le souhaitez, créer un dossier pour la séance (par exemple `TP1`), et entrer dans ce répertoire. Assurez-vous d'être dans le bon répertoire avec `pwd`.

Pour récupérer et décompresser l'archive du TP, on utilisera :

```
curl http://cdn.sci-phy.org/mp2i/tp1-compilation.tgz | tar xvz
```

Vous devriez voir apparaître deux nouveaux dossiers (`compilation` et `syracuse`) dans le répertoire où vous vous trouvez.

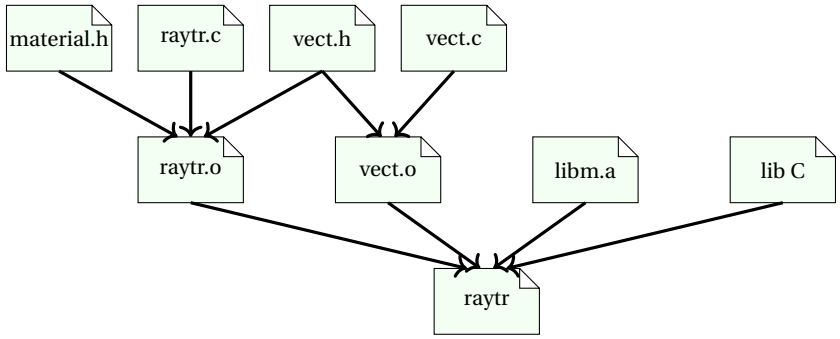
1.a Entrez dans le dossier `compilation`, et examinez son contenu. On y trouve notamment des fichiers sources d'extensions `.c`, des fichiers d'entête d'extension `.h`. Ouvrez les fichiers `.c` et `.h` dans l'éditeur de votre choix.

Nous allons ici essayer de compiler un mini-lanceur de rayons (un programme générant des images de synthèse en utilisant les lois de la réflexion et de la réfraction, de façon similaire à ce qui a été fait dans le cours de physique, en un peu plus élaboré).

On dispose d'un fichier `raytr.c` qui contient le programme pour calculer l'image 3D et d'un fichier `vect.c` qui contient des outils pour effectuer quelques calculs vectoriels en trois dimensions. S'y ajoutent un fichier d'entête `vect.h` qui contient les définitions des fonctions de `vect.c` pour les programmes souhaitant effectuer des calculs vectoriels, et un fichier `material.h` qui contient des définitions de différents types de matériaux (caoutchouc coloré, métal, verre...).

Le but est de rassembler tout cela en un programme que l'on pourra exécuter! Il va nous falloir effectuer les opérations suivantes :

- compiler le fichier `vect.c` en un fichier binaire objet `vect.o`;
- compiler le fichier `raytr.c` en un fichier binaire objet `raytr.o`;
- rassembler les fichiers `raytr.o`, `vect.o` et les fonctions déjà compilées de la bibliothèque standard du C et de la bibliothèque de fonctions mathématiques en un seul fichier binaire exécutable `raytr` (on parle d'étape de *liaison*).



Pour compiler un fichier source C en un fichier binaire objet, on utilise un compilateur. Aujourd'hui, nous utiliserons l'outil `gcc`. Bien qu'il soit possible d'effectuer toutes les étapes de compilation en une seule opération, nous allons réaliser chaque étape une à une. L'avantage étant que si un seul fichier change, il n'est pas nécessaire de tout recompiler (sur un projet de cette taille, ce ne serait pas un gros problème, mais des projets d'ampleur peuvent nécessiter des heures voire des jours de compilation).

La compilation d'un fichier `vect.c` se fait avec cette commande :

```
gcc -c vect.c
```

Le `-c` indique que l'on veut une compilation vers un fichier binaire objet. Le nom du fichier obtenu sera le même excepté l'extension qui sera changée en « `.o` » pour indiquer la nature binaire du fichier.

2.a Compiler le fichier `vect.c` et vérifier que la compilation se passe bien.

On l'a dit, le C est un langage où il est aisé de faire des erreurs. Aussi va-t-on ajouter bon nombre d'options lors de la compilation. On préférera utiliser cette commande :

```
gcc -std=c99 -pedantic -Wall -Wuninitialized -c vect.c
```

`-std=c99` permet de choisir la norme C que l'on souhaite suivre (C99) ;

`-pedantic` demande une attention particulière au respect du standard ;

`-Wall` précise que l'on souhaite qu'il prenne garde aux maladresses courantes (on peut ajouter `-Wextra` pour qu'il soit encore plus attentif) ;

`-Wuninitialized` pour qu'il surveille l'usage de variables non-initialisées.

2.b Compiler le fichier `vect.c` avec les options supplémentaires, et remarquer qu'il suggère cette fois une possible erreur (qui se trouve effectivement en être une !). Corriger cette erreur et compiler le fichier à nouveau.

Attention, toutes les maladresses ne seront pas identifiées. Pour s'en convaincre, on peut commenter (temporairement) l'initialisation de la variable `res` dans la première fonction (`vect_pscal`) et constater en compilant que le compilateur ne se rend pas compte que l'on utilise une variable non-initialisée, même si on lui a demandé d'y prendre garde ! On n'oubliera pas de remettre l'initialisation ensuite.

2.c Compiler le fichier `raytr.c` de la même façon, en corrigeant les coquilles signalées par le compilateur (il s'agit de *petits* détails, esentiellement des fautes de frappe ou des directives oubliées, pour avoir une première habitude des messages d'erreur fournis par le compilateur... si vous n'êtes pas sûr de comprendre une erreur, n'hésitez pas à solliciter de l'aide).

A présent, on a besoin de s'occuper de la liaison. On a besoin de quatre fichiers binaires :

- `vect.o`
- `raytr.o`
- la bibliothèque standard C qui est automatiquement jointe à la liaison
- la bibliothèque de mathématiques qui est disponible déjà compilée sous le nom `libm.a` dans le répertoire `/usr/lib`

Les bibliothèques prêtes à l'utilisation portent généralement le nom `libXXX.a` où `XXX` est le nom de la bibliothèque. L'extension est `.a` car elles ont été compilées de façon un peu particulière. En toute logique, la bibliothèque « `math` » que l'on a annoncé vouloir utiliser avec un `include` devrait s'appeler « `libmath` », mais l'usage a consacré le nom « `libm` ».

Là encore, on utilise gcc, avec la commande suivante :

```
gcc raytr.o vect.o -lm -o raytr
```

Les fichiers objets à inclure sont juste listés, dans l'ordre inverse de leurs dépendances (un fichier utilise les fonctions de ceux placés à sa droite). Les bibliothèques standard, disponibles dans `/usr/lib` sont ajoutées avec l'option `-lXXX` où `XXX` est le nom de la bibliothèque. Ici « `-lm` » pour la bibliothèque mathématique[4], mais on aurait « `-lpthreads` » pour la bibliothèque `pthreads` par exemple.

On termine avec l'option `-o` qui permet de choisir le nom du fichier exécutable produit (généralement un nom sans extension), car sinon le fichier exécutable portera le nom par défaut `a.out` !

3. Lier le programme et vérifier la création d'un fichier pour lequel on a bien un droit en exécution.

4. Exécuter le fichier en tapant la commande « `./raytr` » (le `./` permet d'indiquer que l'on cherche le fichier exécutable dans le répertoire courant, il s'agit d'une mesure de sécurité) et vérifier qu'il a créer un nouveau fichier. Utiliser `file` pour identifier la nature du fichier produit, et ouvrir le fichier résultat. On pourra le faire depuis le navigateur de fichier plutôt que depuis la ligne de commande. Il est possible d'ouvrir un navigateur de fichiers directement dans le bon répertoire depuis la ligne de commande en entrant « `explorer&` ». Sur les machines du lycée, on ouvrira le fichier `render.ppm` produit de préférence avec la visionneuse d'image `Irfanview`.

3.2 Automatisation de la compilation

Le fichier `Makefile` doit contenir un ensemble de règles expliquant quelles commandes il faut exécuter pour construire tel ou tel objet (typiquement, un programme, mais pas seulement). Les règles s'écrivent de la façon suivante :

```
cible: dépendances
    commande(s) pour fabriquer la cible
```

« `cible` » est ce que vous cherchez à construire. « dépendances » est l'ensemble des fichiers qui sont utilisés pour fabriquer la cible (si ces fichiers n'existent pas, `make` cherchera automatiquement une règle pour les fabriquer) simplement séparés par une espace. Suivent ensuite, sur des lignes indentées *par une tabulation*, les commandes permettant de transformer les dépendances en cibles.

Par exemple, pour fabriquer le fichier `raytr.o`, on se base sur les fichiers sources `raytr.c`, `vect.h` et `material.h`, et la règle sera :

```
raytr.o: raytr.c vect.h materials.h
    gcc -std=c99 -Wall -Wuninitialized -pedantic -c raytr.c
```

Comme les variables utilisées pour la compilation sont toujours les mêmes, il est fréquent

4. Dans la pratique, gcc rajoutera cette bibliothèque de lui-même même si on l'oublie, tant elle est fréquemment utilisée.

de définir un alias `CFLAGS` pour l'ensemble des options, et on appellera cet alias avec `$(CFLAGS)` dans la commande de compilation.

Une fois le fichier `makefile` écrit, pour fabriquer un objet, par exemple notre exécutable `raytr`, il suffira de taper dans la ligne de commande « `make raytr` ». `make` déterminera automatiquement les étapes à appeler pour y parvenir, et de façon intelligente : il regarde la date des fichiers à construire, et si un fichier cible est plus récent que ses dépendances, alors il sait qu'il n'a pas besoin de le recompiler !

La plupart du temps, on trouve dans un `makefile` quelques cibles particulières, qui ne sont techniquement pas des fichiers à construire [5] :

- `all` : qui liste tous les fichiers à construire dans un projet, de sorte que la commande « `make all` » compile l'intégralité du projet ;
- `run` : qui décrit une règle qui exécute le projet (après l'avoir compilé si nécessaire), ainsi « `make run` » compile (si nécessaire) et exécute un projet [6] ;
- `clean` : qui supprime tous les fichiers temporaires, notamment de compilation (les `.o`) pour faire un peu de ménage après la compilation ;
- `mrproper` : un peu plus aggressif que `clean`, cette option supprime aussi les exécutables construits pour ne laisser en principe que les sources.

5.a Ouvrir le fichier `makefile` et compléter les règles manquantes.

5.b Lancer la commande « `make mrproper` » pour effacer les compilations précédentes, puis « make all » pour vérifier que la compilation automatique se passe bien (les commandes exécutées doivent être affichées à l'écran).

6. Parcourir les fichiers sources du projet, et regarder s'il est possible de changer quelques détails de la scène produite (la position d'une sphère, sa couleur...) ou du rendu (la focale de la caméra...) en utilisant « make run » après les modifications pour recalculer l'image. Vous pouvez changer le nom du fichier pour conserver les rendus précédents ! On pourra activer la profondeur de champ en augmentant le nombre d'échantillons (« *samples* ») calculés par pixels, mais attention, le temps d'exécution est directement proportionnel au nombre d'échantillons demandés !

4 Suite de syracuse

Dans le second répertoire provenant de l'archive, `syracuse`, vous trouverez un programme C affichant les termes de la suite de syracuse pour un u_0 indiqué dans le fichier jusqu'à atteindre $u_k = 1$ pour la première fois, ainsi qu'un makefile permettant de le compiler.

1.a Naviguer jusqu'au répertoire, et lancer le programme (en provoquant sa compilation) avec « `make run` »

1.b Ouvrir le fichier source avec l'éditeur de votre choix et examiner le programme.

2. Modifier ce programme pour essayer répondre, tour à tour, aux problématiques ci-après.

• Déterminer le temps de vol de la suite, c'est-à-dire le plus petit k tel que $u_k = 1$ (on essayera le programme avec différents u_0).

• Déterminer le plus petit u_0 tel que le temps de vol de la suite soit supérieur ou égal à 20.

• Pour quel u_0 dans $[\![1\,..\,99]\!]$ obtient-on la plus grand temps de vol ?

• Déterminer l'altitude maximale de la suite, c'est-à-dire le plus grand des u_i (on essayera le programme avec différents u_0).

• Déterminer le plus petit u_0 tel que la suite ait une altitude supérieure ou égal à 500.

• Existe-t-il une suite dont l'altitude soit exactement égale à 500 ?

• Pour quel u_0 dans $[\![1\,..\,99]\!]$ obtient-on la plus grande altitude ?

• Déterminer le temps de vol en altitude de la suite, c'est-à-dire le plus petit k tel que $u_{k+1} < u_0$ (on essayera le programme avec différents u_0).

• Déterminer le plus petit u_0 tel que le temps de vol en altitude de la suite soit supérieur ou égal à 20.

• Pour quel u_0 dans $[\![1\,..\,99]\!]$ obtient-on la plus grand temps de vol en altitude ?

• Déterminer le plus petit u_0 tel que la suite contienne le terme 404.

5. pour prévenir que ce ne sont pas des fichiers, on ajoute généralement une commande `.phony` dans le makefile.

6. S'il s'agit de quelque chose que l'on souhaite généralement installer sur un ordinateur, la commande `run` est généralement remplacée par une commande `install`.