Table des matières

Introduction

Le rendu photorealiste d'image en trois dimensions est un secteur de plus en plus actif, la puissance des machines actuelles et des algorithmes recents permettant enfin d'approcher la qualite d'une photo ou d'un film. Mais les logiciels tirant parti de ces algorithmes sont pour la plupart reserves aux professionels, et sont en consequence vendus plusieurs dizaines de milliers de francs. Parmi eux, il existe cependant un moteur de rendu gratuit jouissant d'une tres bonne reputation : POV (Persistance of Vision). Il s'agit d'un moteur de rendu particulierement complet base sur l'algorithme de raytracing. Il a cependant quelques gros defauts :

Pov est ecrit en C, il n'utilise pas l'ensemble des facilites et l'abstraction lies a la programmation objet, ce qui rend son evolution lente, et sa maintenance compliquee.

Pov peut utiliser la puissance d'un reseau ou d'une machine multi-processeur, mais il s'agit de patches et de solutions de rechanges, cela n'etait pas prevu lors de la conception du logiciel.

De meme, il integre un algorithme de radiosite, un algorithme moderne particulierement realiste, mais il n'en tire pas vraiment parti, ne l'implementant que partiellement car la encore, l'utilisation de cet algorithme n'a pas ete prevue lors de la conception du logiciel.

Pov n'est donc pas parfait, et l'ensemble des points que nous venons d'aborder montre clairement que l'ecriture d'un moteur de rendu moderne, integrant radiosite, raytracing et calcul sur un reseau s'imposait. Ce logiciel est zRcube. Il integre de plus une interface graphique et un previewer en openGL pour faciliter son utilisation. Il se base sur le language de description de POV afin de permettre a tous les utilisateurs de POV de tester et d'utiliser rapidement zRcube. On peut donc dresser une liste de ce que doit pouvoir faire zRcube :

- Rendu en raytracing sur un reseau et en multi-processeur
- Rendu en radiosite
- Combinaisons des algorithmes de radiosite et raytracing
- Interface graphique et Preview openGL
- Compatibilite POV

Chapitre 1

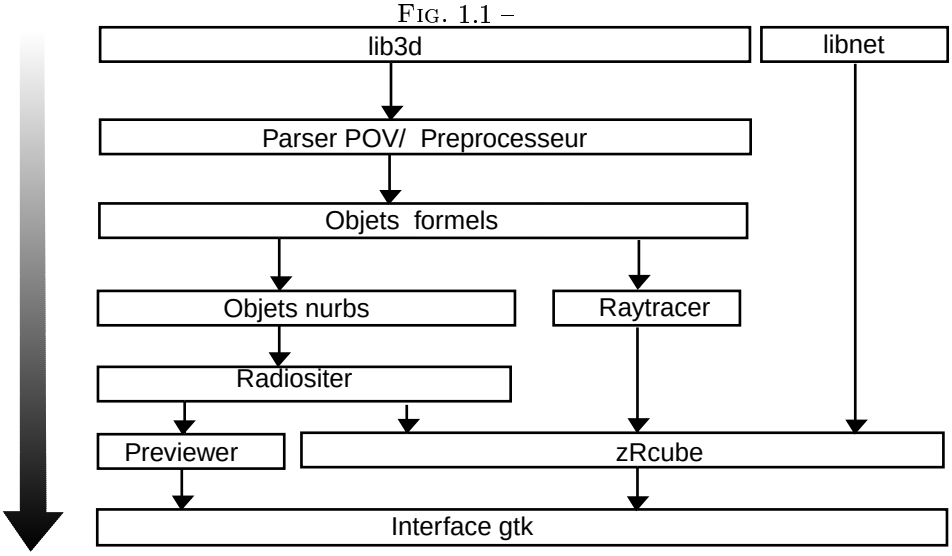
Organisation du projet

1.1 Les differents modules

Lors de la phase d'analyse du projet, nous l'avons decoupe en plusieurs modules, en essayant de les rendre chacun les plus independants possibles des autres, afin de pouvoir les coder separement facilement.

- La lib3d : c'est la premiere librairie que nous avons code, et probablement la plus utilisee. Elle definit les classe Vector et Matrix et l'ensemble de leurs operateurs associes (produit scalaire, produit vectoriel, multiplication de matrices, ...).
- La libpic : elle se positionne comme une interface aux differentes librairies de format d'image. Elle permet de lire les formats ppm,jpeg et png et d'ecrire en png.
- La libnet : ecrite au debut du developpement, c'est un librairie generaliste, permettant de traiter tout type de donnees, permettant de calculer sur un reseau de maniere simple.
- Les objets : divises en deux parties , les objets "formels" et les objets en nurbs, on y retrouve tous les objets de base de POV (sphere, boite, couleur, lumiere ...).
- Le loader et le preprocesseur : ecrits a l'aide de flex++ et bison++, ils permettent de charger des fichiers POV vers nos classes d'objet.
- Le raytraceur : c'est le module permettant de rendre une scene en raytracing, il prend une liste d'objets en entree et en sort une image.
- Le radiositeur : il permet de calculer l'illumination d'une scene decrite en nurbs.
- zRcube : c'est le module principal, permettant de rendre une scene combinant radiosite et raytracing vers une image.
- L'interface gtk : elle appelle les differents binaires et permet d'eviter d'utiliser des longues lignes de commande pour lancer zRcube.
- Le previewer : il permet d'afficher les objets en fil de fer, de se deplacer , et de lancer le calcul de radiosite, afin d'ajuster les reglages sans avoir a

FIG. 1.1 –



raytracer la scene a chaque fois.
La figure 1.1 presente la hierarchie de ces differents modules, on en deduit facilement l'ordre dans lequel nous devions les coder.

1.2 Choix des outils

1.2.1 CVS

zRcube est un gros projet, de l'ordre de 30 000 lignes de c++, la premiere tache a laquelle nous nous sommes atteles fut de determiner les outils que nous allions utiliser pour mettre en commun nos sources.

CVS s'est impose. Ce logiciel, disponible sur toutes les plateformes, centralise toutes les sources sur un serveur internet. Chaque developpeur a chez lui un mirroir de l'ensemble de fichiers sources sur lesquels il travaille. Lorsque ce dernier veut envoyer ses changements sur le serveur (cvs commit), CVS verifie que personne n'a modifie les fichier a envoyer, si c'est le cas, il demande de resoudre le conflit.

Grace au serveur cvs de zRcube (cvs.zrcube.sourceforge.net), chacun a pu mettre a jour chaque jour tous ses changements, et profiter de ceux des autres au jour le jour, evitant ainsi les fastidieuses etapes de mise en commun des sources. Nous l'avons beaucoup utilise : sur un 6 mois de developpement, nous avons realise 1,222 commits (mise a jour sources du serveur), et ajoute 246 fichiers sur le serveur CVS.

1.2.2 C++

Etant donne la taille du projet, et de part son orientation graphique, nous sommes rapidement tous tombe d'accord pour utiliser le C++ a la place du C, afin de pouvoir beneficier de toute la capacite d'abstraction que permet la programmation objet : encapsulation, surcharge, heritage, parties privee/publique ...

On notera ainsi certains gros avantages a l'utilisation du C++ par rapport au C :

- Une plus grande sécurité : en effet les risques d'interactions indésirables entre plusieurs modules sont réduits. Bien qu'un certain nombre d'effets spéciaux puissent apparaître.
- La maintenance d'un programme est simplifiée : si l'on veut modifier une caractéristique du programme il suffit de modifier ou de remplacer le module adéquat.
- Tel module développé pour tel programme pourra facilement être réutilisé pour un autre programme.

1.2.3 Librairies et autres outils

zRcube doit etre capable d'utiliser la plupart des formats d'images connus. Or decompresser du jpeg par exemple n'est vraiment pas une operation aisee a coder soit-meme. Puisque cela n'etait absolumement pas le but du projet, nous avons choisit d'utiliser des librairies qui permettent de lire facilement des formats compliques :

- libjpg
- libpng

De plus, nous avions besoin d'une preview rapide et d'etre capable d'afficher des objets 3d facilement pour debugger (par exemple pour verifier que les objets sont bien charges). La possibilite de beneficier d'une accleration materielle et la simplicite d'openGL nous a pousse a l'adopter.

Nous avons choisit d'utiliser Lex et Yacc pour mettre en place notre parser de fichier pov. Nous y reviendrons en detail dans la section concernant le parser.

1.2.4 Organisation du code source

Avant de commencer a coder, nous avons etablit une "charte" de code afin de garder un code propre et homogene. Cette charte est fournie en annexe.

En plus de donner quelques regles de presentation simple (noms de classes avec une majuscule, pas de majuscules aux methodes, ...), elle definit deux regles :

- pas de variables globales
- pas de variables dans la partie publique d'une classe : pour acceder au contenu d'une variable on doit coder une methode d'interface (la maniere d'interfacer une variable est normalisee).

Ces regles peuvent paraitre trop restrictive, mais leur respect force a coder proprement et de maniere homogene, cette homogeinite etant particulierement utile lors d'un travail en groupe.

1.3 Repartition des taches et echeancier

Les interdependances presentes sur la figure 1.1 impose un ordre de realisation, cependant, tous les modules d'un meme niveau ont pu etre codes en meme temps. Le tableau suivant presente la maniere dont nous avons repartit les taches de maniere individuelle et temporelle.

Mois	Jean-Baptiste	Guillaume	Edouard	Henry
Janvier	Libnet	loader pov	libnet	previewer
Fevrier	Libnet	loader pov	libnet	previewer
Mars	Nurbs	Raytracing	Raytracing	Nurbs
Avril	Radiosite	Raytracing	Raytracing	Radiosite
Juin	Radiosite/ Raytracing	Radiosite/ Raytracing	Raytracing resequ Interface GTk	Radiosite/ Raytracing

Il est evident que cette repartition est inexacte : chaque module a ete mis a jour en fonction de nos besoins. Ainsi, par exemple, nous avons ajoute les instructions necessaires a la radiosite au parser lorsque le module de radiosite a ete code.

Chapitre 2

Parser POV/Pre-processeur

2.1 Objets POV

La partie parser est la partie du programme final qui prend en entrée un fichier texte comprenant un descriptif de scene de type POV-Ray, et qui construit tous les objets representant cette scene en memoire. Avant de pouvoir commencer le parseur, il faut donc ecrire les classes de tous les objets qui pourraient etre trouvés dans dans la norme POV.

2.1.1 Objets de base

Heureusement, la documentation de POV-Ray est bien faite, et j'ai donc trouvé tout ce qu'il me fallait assez rapidement : Sphere, Box, Cylinder, Cone, Plane, Disc Il y a d'autres objets plus complexes (height fields, fractals ...) mais nous avons decidé de ne pas le supporter pour l'instant. Avant de commencer, j'ai tout d'abord téléchargé le code source de POV-Ray (www.povray.org), pour voir comment les auteurs avaient tout construit, et pour trouver toutes les propriétés des objets. Apres avoir survolé tout le code quelques fois et rendus quelques images avec POV-Ray pour me remémoriser ses possibilités, j'avais en tête la structure que j'allais utiliser. Le projet est réalisé en C++, donc nous pouvons utiliser l'heritance de classes, ce qui facilite beaucoup les choses. Notre code sera beaucoup plus facile a maintenir et relire que celui de POV.

2.1.2 Description des propriétés de base

J'ai donc créé une classe Object, qui servira de classe de base pour tous les objets, qui ont tous les memes propriétés de base :

```
int type;
int flags;
Object *sibling;
Texture *texture;
```

```
Interior *interior;
BBox *bbox;
Transform *transform;
```

type : Le type est un champ de bits qui correspond a des informations sur cet objet qui seront utilisées plus tard pour le ray-tracing. Dans POV, il existe treize differents types, que je n'enumérerai pas ici, mais voici quelques exemples : Le bit 2 nous indique si cet objet possede une texture, le bit 3 nous indique s'il a des fils (elements suivants dans une liste chainée), le bit 6 nous indique si cet objet est une source lumineuse... Chaque puissance de 2 correspondant au bit est defini par une constante (ex : #define TEXTURED_OBJECT 2).

flags : Flags est similaire au type, c'est un autre champ de bits. Le premier bit indique que l'objet ne provoque pas d'ombre, le bit 2 indique que l'objet est fermé... Comme pour le type, chaque bit a une constante (ex : #define NO_SHADOW_FLAG 1).

***sibling :** Sibling est un pointeur vers un autre objet, qui sera utilisé plus loin pour construire les listes d'objets a rendre.

***texture :** Un pointeur vers une structure de texture, structure qui n'a pas encore été implémenté, j'y reviendrai plus tard.

***interior :** Un pointeur vers une structure Interior, structure qui servira a determiner les rayons presents dans les objets pour le tracé de rayons, et l'indice de refraction du milieu. Ici aussi, j'y reviendrai avec beaucoup plus de details lors de la partie Ray-Tracing.

***bbox :** Un pointeur vers une bounding box. Une bounding box est un ensemble de deux vecteurs qui definissent une boite qui entoure tout l'objet. Cet element est utilisé pour accelérer l'algorithme qui cherche des intersections dans tous les objets. Avant de calculer le point precis d'intersection du rayon avec l'objet, on regarde s'il y a une intersection avec la bounding box, ce qui est plus rapide a calculer. Si ce n'est pas le cas, il ne sert a rien d'essayer avec l'objet lui meme.

***transform :** Un pointeur vers une structure de transformation 3d, qui contient une matrice directe et une matrice inverse. Ces matrices sont remplies lorsque l'on utilise des transformations des objets POV.

J'ai ensuite definis des fonctions qui permetront de mettre des bits a 1 ou 0, et de tester tous les champs possibles dans Type et Flags, a partir des constantes definies.

2.1.3 Derivation des classes d'objets

Une fois la classe Object definie, j'ai pu la deriver pour créer les vrais objets. Chaque objet aura donc les propriétés de base de la classe Object, et sera complété par ses propriétés particulières. Voici une definition generale de chaque objet :

```
class <NomObjet> : public Object
{
  // Propriétés locales a cet objet
public :
  // Fonctions Constructors de cet objet
  // Fonctions Interface
  // Fonctions Generales
};
```

Les propriétés locales de l'objet Sphere sont un centre et un rayon. Les fonctions constructeurs sont les initialisations possible de l'objet quand il est créé. Les fonctions interface sont les fonctions qui permettent de recupérer les propriétés (qui ne sont pas publiques), et de les remplir (ex : Sphere : :setRadius(float r) { radius = r ; }). Les fonctions generales sont toutes les fonctions qui viendront plus tard pour la suite du projet (ex : intersection de rayons..)

2.1.4 Objets speciaux

Il y a aussi les objets speciaux, qui ne sont pas forcement dérivés de la classe Object ; les objets Ray, Camera, et LightSource. LightSource est le seul objet special a être dérivé de la classe Object, c'est un point lumineux qui servira de lumière pour les calculs de radiosité et le tracé de rayons. La Camera contient des vecteurs qui definissent son emplacement dans la scene, son orientation et des paramètres d'effets speciaux (ex : focal blur, field of view) qui seront utilisés pour le ray-tracing. Et Ray, c'est un ensemble de deux vecteurs (origine et direction), munit d'une liste chainée de structure Interior, pour savoir dans quels objets un rayon est present a n'importe quel moment du tracé de rayon, notamment pour calculer l'attenuation d'intensité.

2.2 Parseur

Une fois les objets de base definis, je pouvais commencer le parseur. Je me suis donc remis dans les sources de POV pour voir comment les auteurs avaient fait. Le code est immense, et rien n'est general. Le fait que ce soit du C et non du C++ ne facilite pas la lecture. Le parseur POV est fait de deux sous parties : un Tokenizer et le Parseur proprement dit. Le Tokenizer ce charge d'extraire des expressions regulières du fichier, en les passant sous forme de Token au Parseur, qui gère les règles de grammaire pour construire la scène en memoire. Je me suis donc mis sur le net a la recherche de documentation détaillée sur les grammaires et le parsing. Heureusement, j'avais fait un exposé sur le sujet l'année dernière, donc il me restait nombreuses docs, et je n'avais pas encore tout oublié.

2.2.1 Definition d'un langage formel

2.2.1.1 Definitions

Un vocabulaire est un ensemble fini de symboles : V = { a, b, c, ..., z } Une chaîne sur un vocabulaire V est une concaténation de symboles : x = ab La concaténation de deux chaînes a pour résultat une chaîne obtenue en juxtaposant ces deux chaînes : avec x = ab et y = cd, on obtient xy = abcd. V* est l'ensemble de toutes les chaînes sur le vocabulaire V. Par définition un langage est un sous-ensemble de V*. C'est un ensemble de chaînes construites avec ce vocabulaire.

2.2.1.2 Propriétés

Il est important de se rappeler que les langages sont des ensembles construits sur des vocabulaires donnés, il faut donc toujours spécifier le vocabulaire associé à un langage.

Réunion : Si L est la réunion des langages L1 et L2, il est composé des chaînes appartenant à L1 ou à L2 ou aux deux.

Intersection : Si L est l'intersection des langages L1 et L2, il est composé des chaînes appartenant à L1 et à L2.

Concaténation : Le produit par concaténation de deux langages L1 et L2 contient toutes les chaînes formées par la concaténation d'une chaîne de L1 avec une chaîne de L2.

Extension : L'extension d'un langage, noté L* est la réunion de tous les langages obtenus en élevant le langage à toutes les puissances entières (non négatives). Exemple : si L = { a, b, c, ..., z }, L* est l'ensemble de toutes les chaînes de longueur quelconque formées des lettres de L.

2.2.2 Notions de grammaire formelle

2.2.2.1 Introduction

On rappel qu'un langage naturel est un sous-ensemble de l'extension d'un certain vocabulaire. Une grammaire doit donc au moins servir à définir quelles sont les chaînes qui appartiennent à un langage. Sur le plan formel, une grammaire est un système constitué d'un nombre fini de composants et destiné à la description d'un langage. Système de production : Un système de production est un couple (V, P) où V est un vocabulaire fini de symboles et P est un ensemble fini de productions. Les productions ont la forme x -> w où x et w sont des chaînes sur V (mais x ne peut être vide), c'est-à-dire x appartient a V+ et w appartient a V*. Pour définir un système de productions, il sufit de définir ses deux composantes V et P. Ainsi : V = {a, b, c } et P = [ba -> ab, ca -> ac, cb -> bc] constituent un système de productions. a -> b est une dérivation directe, et a => b est une dérivation indirecte, qui peut être décomposée en une suite de dérivations directes.

2.2.2.2 Définition d'une grammaire formelle

Une grammaire formelle est un quadruplet : G = {Vn, Vi, P, S} où V t est un vocabulaire terminal fini. C'est sur ce vocabulaire que sont formées les chaînes (phrases) du langage que la grammaire définie. Ces chaînes appartiennent donc à Vi*. Vn est un vocabulaire non terminal fini. C'est un vocabulaire auxiliaire utilisé dans les règles de grammaire. Le plus souvent Vn contient les éléments du métalangage utilisé pour dénommer les constituants. Vn et Vi n'ont aucun élément en commun ; le vocabulaire V de la grammaire est la réunion de Vn et Vi. Le couple (V, P) est un système de productions. P est un ensemble de productions, appelées dorénavant règles de la grammaire de la forme : x A y -> z , tel que A appartient a Vn, et x,y,z appartiennent a Vi*. Les deux membres d'une règle sont appelés "partie gauche" et "partie droite" : la partie gauche contient toujours au moins un élément de Vn ; si l'on considère chaque règle x -> y comme un couple (x, y), on peut alors dire que P est un sous-ensemble du produit cartésien V+ X V* . S est un élément particulier de Vn appelé le symbole initial ou axiome de la grammaire. Une grammaire n'a toujours qu'un seul symbole initial. Le langage défini par une grammaire G s'écrit L(G). Une phrase de L(G) est toute chaîne x de Vi*, telle que S => x. Le langage défini par une grammaire G est : L(G) = { x / x appartient a Vi* et S => x }

2.2.3 Parseur POV

Quand il s'agit d'utiliser une grammaire pour réaliser un parseur, un nouveau mot-clef important apparait : les Tokens. Un token est l'unité atomique d'un langage. Il consiste d'une description de syntaxe et d'un type. Une instance de token est appellé lexeme : une chaine de charactères qui conforme a la description syntaxique du token. Le parseur doit donc lire les charactères du fichier, extraire les tokens, et s'assurer que les lexemes sont valides, avant d'etablir de relations avec la grammaire. Par exemple le token UNDECLARED_IDENTIFIER est renvoyé si on trouve un mot qui n'appartient pas a l'ensemble de definition des identifieurs POV (ex : sphere, light_source, camera, ...).

Le vocabulaire est constitue de :

- symboles terminaux ("a", "b",... "Z", "0", ... "9", ".", ",", "}", ...)
- symboles auxiliaires ("|", "[", "]", "...")
- une suite de symboles terminaux est un identifier
- Un identifier entre crochets [] est optionel
- | signifie un ou (exclusif)
- Un identifier suivi de ... signifi qu'il peut y avoir une concatenation d'instantiations de cet identifier.

2.2.3.1 Exemple raccourci de la grammaire POV

```
DIGIT : 0|1|2|3|4|5|6|7|8|9
EXP : e|E
SIGN : +|-
```

```
FLOAT : [DIGIT...] [.] DIGIT... [EXP [SIGN] DIGIT...]
SPHERE : sphere { FLOAT|VEC2D|VEC3D, FLOAT [OBJECT_MODIFIERS...] }
```

OBJECT_MODIFIERS n'est pas defini ici, normalement il le serait, ce serait une liste de operations que l'on peut faire sur tous les objets (ex : texture, colour, normal, transform....).

2.2.3.2 Implementation

Il y a deux implementations possibles pour le parseur :

2.2.3.2.1 La grammaire est codé dans le parseur ("hard coded grammar").

Principe token, par rapport a la grammaire, donc on renvoyer un message d'erreur si c'est un token non attendu.

Avantages : facile a réaliser. Inconveniants : le code sera très grand, peu lisible, avec pleins de switch et de case partout, et on pourra difficilement modifier les règles de la grammaire.

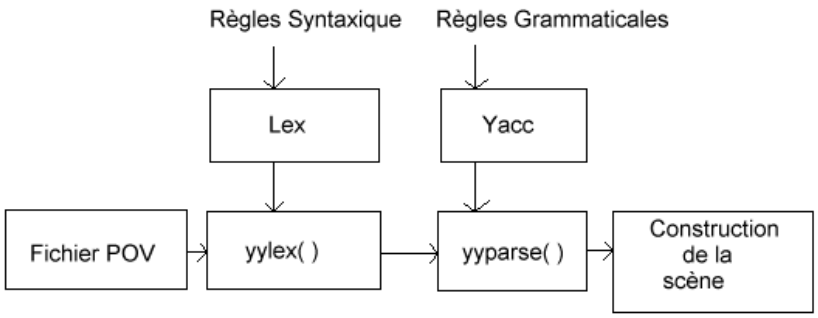
2.2.3.2.2 La grammaire est lu dans un fichier texte externe que l'on peut facilement modifier ("external grammar") :

Principe règles syntaxiques. Tant qu'on est pas arriver a la fin du fichier (tant que GetNextToken renvoye une valeur non-nulle), on traite le token recuppéré. Le traitement va donc aller chercher dans le fichier grammaire (ou dans la memoire si on charge toute la definition de la grammaire en memoire dès le debut pour gagner du temps après) quel type de token doit arriver après pour que la grammaire soit satisfaite. Si GetNextToken ne renvoit pas ce type de token, on renvoye un message d'erreur. Le fichier grammaire serait de la même forme que l'exemple plus haut. La fonction qui renverrai le token attendu par rapport a la grammaire serait GetNextGToken, par exemple GetNextGToken(SPHERE_TOKEN) renverrai d'abord LEFT_BRACE_TOKEN ("{"), puis FLOAT|VEC2D|VEC3D, puis COMMA_TOKEN (",") ... Le code du parseur serait donc beaucoup plus compacte que la version précedente, et on pourrait ajouter des règles de grammaire plus facilement. Mais ca ne fait que reporter le probleme aux constructeurs d'objets : tout les cas particuliers fait dans le code du parseur 1) devront être reportés dans chaque constructeur d'objet, qui doit savoir toutes les correspondances entre l'ordre des paramêtres. Un petit exemple : On sait que l'on est en train de construire une sphere (on a deja recupéré les tokens SPHERE_TOKEN et LEFT_BRACE_TOKEN). Si le prochain token est un FLOAT_TOKEN (grammaire validée), et que l'on passe cette valeur au constructeur de sphere, il doit determiner si il doit remplir sa variable Center ou Radius... Donc les switch sont dans le code des constructeurs, ce qui rend quand même le code un peu plus modulable.

Avantages : code compact et plus generalisé Inconveniants : il ne faudrait pas qu'il y ait trop de cas particuliers, et il faut deux tokenizers. Il faut aussi charger toutes les règles en memoire avant de commencer le parsing, donc on rajoute le temps de chargement et le temps de recherche en memoire a la complexité generale.

2.2.4 Lex

Avant de me lancer dans l'implementation, j'ai suivi le conseil tres averti de Jean -Baptiste qui me disait de regarder s'il n'y avait des outils unix (lex et yacc) qui pourraient me simplifier la tache. J'ai donc trouvé des sites qui expliquaient le fonctionnement de ces programmes qui sont utilisés notamment pour faire des compilateurs, et plus generalement tout ce qui traite du parsing (http ://www.combo.org/lex_yacc_page/). Ce sont deux programmes qui prennent du texte en entré et donnent du code source en sortie. Ils sont concus pour être utilisés ensembles : lex créé une fonction qui va chercher les tokens dans l'input, et yacc utilise cette fonction pour faire correspondre les tokens récupérés à des règles de grammaire. Donc l'entré de lex est une definition syntaxique des tokens possibles, et l'entré de yacc est une definition grammaticale du langage que l'on cherche a parser. En resumé, si j'utilise lex et yacc pour le parseur :



Je me suis donc plongé dans la documentation de Flex (une version optimisé de lex, qui fait pareil). Le principe est assez simple, on lui donne un fichier qui contient des definitions et des règles sur les expressions regulières que l'on veut qu'il trouve dans l'input.

2.2.4.1 Vocabulaire d'operateurs

Un vocabulaire special est utilisé pour les definitions et les règles : " \ [ ] ^ -? . * + | ( ) $ / { } % < > Tous ces charactères rencontrés dans une definition correspondent a des operations sur la chaine. Quelques explications sur une partie du vocabulaire :

- Les crochets ("[", "]") indiquent une classe de charactères (ex : [0-9] indique un chiffre entre 0 et 9, [a-zA-Z] indique une lettre de l'alphabet

minuscule ou majscule...)
- Les curly brackets ("{", "}") indiquent une derivation (ex : {ALPHA} sera remplacé par [a-zA-Z])
- Une barre ("|") est utiliseé pour un OU exclusif.

2.2.4.2 Definitions

Le fichier contient d'abord une partie de definitions qui permettent de simplifier le reste :

```
DIGIT [0-9]
ALPHA [a-zA-Z]
ALPHANUM {ALPHA}|{DIGIT}
```

La partie gauche devient un symbole qui pourra être dérivé en sa partie droite si rencontré dans les règles. Ce sont en fait une sorte de macros qu'un pre-processeur vient remplacer avant le vrai traitement.

2.2.4.3 Règles

Ensuite il y a la partie des correspondances expression-action, qui utilise le même vocabulaire special :

```
{DIGIT}    return (DIGIT_TOKEN);
"{"        return (LEFT_CURLY_TOKEN);
sphere     return (SPHERE_TOKEN);
\n         num_lines++;
```

La partie gauche est l'expression régulière et la partie droite est le code C qui y correspond. Toutes les constantes *_TOKEN sont definies dans un fichier .h qu'on peut inclure au debut du fichier lex. Quand une expression est trouvé dans l'input, le lexeme de ce token est mis dans une variable globale yytext.

2.2.4.4 Routines Utilisateur

Il y a une troisième partie dans le fichier lex, ou l'on peut mettre du code C. En general on met la fonction main qui appelle la fonction yylex jusqu'a ce que celle ci renvoye 0, mais dans le cas ou on utilise lex en externe, il suffit de definir la fonction yywrap. Cette fonction sera appellé internement a lex quand la fin du fichier arrive a l'input. Si la fonction renvoye 1, cela veut dire que le traitement est finie, si elle renvoye 0, cela veut dire qu'on s'est occupé de changer de fichier ou de changer l'input et le traitement continue. Une fonction yywrap habituelle serait :

```
int yywrap() { return 1; }
```

Pour le pre-processeur, cette fonction s'occupera de gerer une pile de fichiers ouverts, pour supporter les #include.

2.3 Le Parseur

2.3.1 Yacc

Yacc est un programme qui, comme Lex, prend un fichier texte en entrée et nous donne un fichier de code source C en sortie (Lex et Yacc sont des programmes inclu dans nombreuses distributions Unix, il existe même des versions Win32 et Mac). Comme je l'avais expliqué plus haut, Lex reconnait des expressions régulières dans le texte entrant et renvoye les tokens correspondants. Exemple de règles Lex qui renvoyent des tokens :

```
"{"     return (LEFT_CURLY_TOKEN);
sphere  return (SPHERE_TOKEN);
```

Yacc examine donc les tokens renvoyés par la fonction int yylex( ) (fonction créé par Lex a partir des règles...) pour voir si leur ordre et presenceéabsence concorde avec les règles de grammaire établies. S'il y a correspondance, du code spécifié par l'utilisateur est excuté, sinon une fonction d'erreur est appellé et le parsing s'arrete (l'arret n'est pas obligé, comme on le verra plus loin, mais a lieu par default). La fonction C que sort Yacc est int yyparse( ). Notez qu'il n'est pas indispensable d'utiliser Lex pour réaliser l'analyseur lexical, cepedant, c'est hautement recommandé car ces deux programmes ont été fait pour fonctionner ensemble. Si vous choisissez d'ecrire votre propre fonction d'analyse lexicale, elle doit s'appeller int yylex( ) car c'est le nom qu'utilise Yacc pour aller chercher ses tokens. Je vais ici présenter les grandes fonctionnalités de Yacc. Je ne rentrerai pas trop dans les details car cela depasserai le cadre du rapport, etant donné que l'on peut réalisé un parseur sans utiliser toutes les options de Yacc... Pour plus d'information, veuillez consulter la documentation Yacc ("man yacc" sous unix).

2.3.1.1 Spécification de la grammaire

Comme pour Lex, le fichier spec que lit Yacc comporte trois parties séparées de %% :

```
declarations
%%
règles
%%
programmes
```

Le but des règles de grammaire est d'imposer une structure au texte qui est traité pour un programme. Commencons par un exemple simple : Si on a un analyseur lexical qui nous retourne JANVIER_TOK, FEVRIER_TOK, ... s'il trouve un mois dans le texte d'entrée, LUNDI_TOK, MARDI_TOK, ... s'il trouve un jour, VIRGULE_TOK s'il trouve une virgule, et DATE_TOK s'il trouve une suite d'entiers, alors on peut definir les règles de grammaire suivantes dans le fichier Yacc :

```
date : jour DATE_TOK mois VIRGULE_TOK DATE_TOK ;
mois : JANVIER_TOK | FEVRIER_TOK ... | DECEMBRE_TOK ;
jour : LUNDI_TOK | MARDI_TOK ... | VENDREDI_TOK ;
```

(les "..." ne sont pas permis) Chaque règle est terminée par un " ;". Par convention, tous les mots non-terminaux (qui peuvent être dérivés) sont en minuscule, et les mots terminaux (Tokens) sont en majuscule. La barre "|" est utilisée quand on veut qu'un symbole non-terminal puisse être dérivé en plusieurs sequences sans réécrire le symbole a chaque fois. Par exemple, on aura peut faire :

```
mois : JANVIER_TOK
mois : FEVRIER_TOK
...
mois : DECEMBRE_TOK
```

Mais il est recommandé d'utiliser la barre pour que le code soit clair (c'est aussi beaucoup plus rapide a coder). Avec ces règles, une entrée valide sera "mardi 6 avril, 1992", mais aussi "jeudi 42 mars, 2010429", ce qui est un peu laxiste. On pourrait donc definir des intervales de nombres. Ceci serait plutot fait dans les definitions Lex, mais pourrait aussi se faire dans Yacc. Cependant, il faut faire attention de bien dissocier la grammaire de la syntaxe. Revenons a la grammaire pour le moment. Il est possible qu'un symbole soit dérivé en chaine vide, dans ce cas, on ne met rien entre le " :" et le premier "|" ou " ;" suivant :

```
empty : ;
```

Les symboles representant des tokens doivent être déclarés en tant que tels dans la partie declarations. La facon la plus simple de declarer des tokens est la suivante :

```
%token tok1 tok2 ...
```

Tout symbole non défini dans la partie declarations est considéré comme un symbole non-terminal et doit figurer a gauche d'au moins une règle. Le symbole non-terminal le plus important est le symbole de départ, appellé "start". Le parseur que créé Yacc est concu de manière a reconnaitre une structure qui correspond a ce symbole. C'est donc la structure la plus generale decrite par les règles de grammaire. Par default, le symbole start est le coté gauche de la première règle dans la partie des règles. Cepedant, on peut le declarer dans la partie declarations en faisant :

```
%start symbol
```

La fin des entrées de données texte est signalé au parseur par un token special appellé "endmarker". Si les tokens qui composent le texte d'entrée forment une structure qui correspond au symbole start, précédé du endmarker, la fonction yyparse retourne quand le endmarker est lu ; dans ce cas la l'entrée est accèptée. Si le endmarker est lu autre-part, c'est une erreur. En general le endmarker est retourne par l'analyseur lexical quand il rencontre la fin du fichier.

2.3.1.2 Actions

Pour chaque règle de grammaire, l'utilisateur peut associer du code C a être executer a chaque fois que la règle est reconnue dans les données d'entrée. Ces parties de code sont appellées actions et peuvent retourner des valeurs et récupérer des valeurs retournées par des actions précedantes. L'analyseur lexical peut aussi renvoyer des valeurs avec les tokens (ex : la valeur entière que represente le token si celui-ci est un token qui definit les entiers). Voici deux exemples simples d'actions attachés a leur règle de grammaire :

```
A : '(' B ')'
        { bonjour( 1, "abc" ) ; }
  ;
XXX : YYY ZZZ
        { printf("un messageèn") ; flag = 25 ; }
    ;
```

Pour faciliter la communication entre la grammaire, les actions, et le parseur, on utilise les symboles de retour :

```
$$, $1, $2 ...
```

Pour retourner une valeur, l'action assigne cette valeur a $$. Par exemple, une action qui renvoye 1 serait ecrite :

```
{ $$ = 1 ; }
```

Pour obtenir des valeurs qui ont été retournées par des actions precedantes et l'analyseur lexical, l'action utilise les pseudo-variables $1, $2, $3.... qui réfèrent aux valeurs retournées par les symboles de droite d'une règle, en partant du premier apres le " :". Par exemple si on a une règle :

```
A : B C D ;
```

Alors $2 contient la valeur renvoyée par C, et $3 par D. Par default, la valeur que renvoye une règle est la valeur que renvoye le premier symbole de droite. Donc on n'a jamais vraiment besoin d'ecrire $$ = $1. Un autre exemple simple, si on veut utiliser des paranthèses autour d'une expression arithmetique :

```
expr : '(' expr ')'
           { $$ = $2 ; }
;
```

Par default, les valeurs renvoyés par les actions sont de type entier (int), mais il est souvent nécessaire de travailler avec d'autres types. Pour cela, on utilise %union et %type dans la partie declarations :

```
%union { int ival ; float fval ; }
...
%type <ival> iexpr
%type <fval> fexpr
```

Le symbole "fexpr" renverra donc des valeurs a virgule flottante, et "iexpr" renverra des entiers.

2.3.1.3 Conflicts d'ambiguite

Il est possible que des règles de grammaire soient ambigues. C'est le cas si une chaine d'entrée peut être arrangée de plusieurs facons. Par exemple :

```
expr : expr '-' expr
```

Cette règle est utilisé pour traiter les operations arithmetiques. L'ambiguité découle du fait qu'on a aucune information sur l'associativité. Ainsi, la chaine "expr - expr - expr" pourrait être traité par associativité gauche ou droite :

```
( expr - expr ) - expr
ou
expr - ( expr - expr )
```

Yacc détêcte ce problème, il ne sait donc pas si il doit continuer a lire des tokens ou dériver tout de suite. Ce conflict s'appelle un conflict shift-reduce (shift est le fait de continuer a prendre des tokens, reduce est une dérivation). Il se peut aussi qu'on ait des conflicts reduce-reduce, ou Yacc ne sait pas quelle règle il doit utiliser quand plusieurs sont valides. Ces conflicts sont inévitables. Lorsque vous compilez votre fichier Yacc, il vous affiche combien de conflicts shift-reduce et reduce-reduce il a trouvé, mais il compile quand même. Il y a des règles qui determinent a l'avance s'il doit reduce ou shift selon la situation. Par default, ces règles sont les suivantes :

1. Lors d'un conflict shift-reduce, il shift.
2. Lors d'un conflict reduce-reduce, il dérive en utilisant la règle de grammaire définie en premier (pour le membre gauche en question)

Un exemple de ces règles mis en action :

```
stat : IF '(' cond ')' stat
     | IF '(' cond ')' stat ELSE stat
   ;
```

Dans ces règles, le IF et ELSE sont des tokens, cond est un symbole non-terminal qui decrit une expression logique de condition, et stat est un non-terminal decrivant une phrase. Ces deux règles forment une construction ambigue. Par exemple, si on a la chaine "IF (C1) IF (C2) S1 ELSE S2", elle pourrait être structurée de deux facons :

```
IF ( C1 ) { IF ( C2 ) S1 } ELSE S2
ou
IF ( C1 ) { IF ( C2 ) S1 ELSE S2 }
```

La deuxième interprètation est celle qui est utilisée dans la plupart des langages de programmation : chaque ELSE est associé avec le dernier IF qui n'a pas encore de ELSE associé. Si le parseur voit la sous-chaine "IF (C1) IF (C2) S1" et est en train de regarder le premier ELSE, il peut soit dériver pour avoir

```
IF ( C1 ) stat
et lire le reste
ELSE S2
et dériver
IF ( C1 ) stat ELSE S2
```

Ceci nous donne la première interpretation. Mais il peut aussi continuer a lire des tokens apres le ELSE, donc lire S2, et donc la partie droite de

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

peut être dérivée par la deuxième règle pour donner

```
IF ( C1 ) stat
```

qui peut ensuite être dériver par la première règle pour donner la deuxième interpretation, qui est preferable. Il y a donc un conflict shift-reduce, et donc Yacc va shifter par default, ce qui nous donne l'option preferable.

2.3.1.4 Precedence des operateurs

Il y a une situation ou les regles de resolution de conflict ne suffisent pas : les expressions arithmetiques. Pour cela il faut rajouter les notions de niveaux de precedence pour les operateurs, et preciser quels types d'associativite sont permis. Ces informations sont attaches aux tokens dans la partie declarations :

```
%left '+' '-'
%left '*' '/'
```

Tous les tokens qui sont sur la meme ligne ont la meme precedence et associativite : les lignes sont listees en ordre croissant de precedence. Plus et moins sont associatifs gauche (left), et ont une precendence plus faible que la multiplication et la division, qui sont eux aussi associatifs gauche. %right est utilise pour definir une associativite droite, et %nonassoc pour decrire des operateurs qui ne peuvent pas associer avec eux-memes. Voici un exemple de ces regles :

```
%right '='
%left '+' '-'
%left '*' '/'
%%
expr : expr '=' expr
     | expr '+' expr
    | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | NAME
     ;
```

Avec cette grammaire on pourrait structurer

```
a = b = c*d - e - f*g
```

de telle sorte :

```
a = ( b = ( ((c*d)-e) - (f*g) ) )
```

Mais il manque encore quelquechose : les precedences des operateurs unaires, tel que le "-". Dans ce cas, l'operateur binaire "-" a le meme symbole que le moins unaire, mais pas la meme precedence. Pour cela, on utilise Il suffit de mettre un operateur qui a le niveau de precedence desire, juste apres le Par exemple, pour le moins unaire, il faudrai la meme precedence que la multiplication :

```
expr : expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | '-' expr %prec '*'
     | NAME
     ;
```

Ces systemes sont utilises par Yacc pour resoudre des conflicts de parsing.

2.3.1.5 Traitement d'erreurs

Lorsque survient une erreur, il se peut que l'on doivent liberer de la memoire, modifier des tableaux internes etc... bref il faut pouvoir savoir quand une erreur a lieu. On a ensuite le choix de continuer de parser ou pas. Il y a donc des moyens de faire continuer le parseur lorsqu'une erreur est trouvee que je ne couvrirai pas ici (le parser POV s'arrete a la premiere erreur). Pour que l'utilisateur puisse avoir un certain controle, Yacc a un token reserve "error" qui peut etre place dans les regles de grammaire pour donner une possibilite d'action en cas d'erreur. Voici l'exemple d'arithmetique qui detecte des erreurs de syntaxe :

```
expr : error
         { printf("Syntax error in arithmetic expr.\n"); }
     | expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | '-' expr %prec '*'
     | NAME
     ;
```

Dans ce cas, si une expression "expr" est attendue et que les tokens qui arrivent ne correspondent a aucune des derivations possibles, le parseur quittera en renvoyant le message d'erreur.

2.3.2 Implementation de la grammaire POV en Yacc

Une fois la doc de Yacc lu et comprise, et quelques exemples compiles, je commencai a convertir la grammaire de POV en grammaire pour Yacc. La premiere chose faite etait de recopier tous les tokens definis par le langage POV dans la partie declaration.

2.3.2.1 Scene

Dans la documentation POV, la structure la plus generale d'un fichier pov est definie par le non-terminal "scene" :

```
SCENE : SCENE_ITEM...
SCENE_ITEM : LANGUAGE_DIRECTIVES | camera { CAMERA_ITEMS... } | OBJECTS | ATMOSPHERIC_E
| global_settings { GLOBAL_ITEMS }
```

On remarque que leur syntaxe de grammaire est pratiquement la meme que celle de Yacc. Il y a cependant quelques changements a faire pour que Yacc accepte ces regles. "SCENE_ITEM..." defini ici une suite non-nulle de type "SCENE_ITEM". En Yacc, il faut donc rajouter un non-terminal pour implementer la liste. Cela donne ceci :

```
scene : scene_item
      | scene scene_item
     ;
scene_item : language_directives
           | camera
           | object
           | atmospheric_effects
           | global_settings
          ;
```

Il faut aussi definir '%start scene' dans la partie declarations pour que Yacc sache que scene est la structure la plus large possible d'une scene a parser. J'ai ensuite pris chaque non-terminal, trouve sa definition grammaticale dans la doc POV, et l'ai converti en regles de grammaire pour Yacc.

2.3.2.2 Types

Les types utilises dans la grammaire POV sont :

float : valeurs litterales representants des nombres reels, ou alors des fonctions renvoyant des float (ex : sin, cos,...).

vector : note < float, float, float >, ils representent des vecteurs de R*R*R, ou des fonctions renvoyant des vecteurs.

int : entiers ou fonctions renvoyant des entiers.

color : vecteurs de composantes Rouge Vert Bleu, ou fonctions renvoyant des couleurs.

string : chaine de characteres.

Pour l'instant j'ai implemente le float et le vector (l'entier est un float typecaste en int). J'ai donc utilise l'union de Yacc pour lui dire que plusieurs types pourront etre renvoyes par les actions des regles (pour l'instant il manque la couleur) :

```
%union {
   char * tval;
int ival;
   float fval;
   float vval[3];
}
```

Pour implementer ces types, je me suis base sur les exemples de precedences d'operateurs trouve dans la documentation Yacc, car la grammaire POV est incorrecte pour les types. Par exemple, voici la definition (raccourci) du float dans POV :

```
FLOAT : NUMERIC_TERM [SIGN NUMERIC_TERM]
SIGN : + | -
NUMERIC_TERM : NUMERIC_FACTOR [MULT NUMERIC_FACTOR]
MULT : * | /
NUMERIC_FACTOR : FLOAT_LITERAL | FLOAT_FUNCTION
```

Mais si on copie ces regles telles quelles dans Yacc, on s'appercoit tres vite qu'il y a des problemes d'associativite et de precedence entre les operateurs. La definition Yacc (non complete, mais equivalente a celle de POV cite au dessus) est donc :

```
float : fexpr;
fexpr : LEFT_PAREN_TOKEN fexpr RIGHT_PAREN_TOKEN
          { $$ = $2; }
      | fexpr PLUS_TOKEN fexpr
          { $$ = $1 + $3; }
      | fexpr DASH_TOKEN fexpr
          { $$ = $1 - $3; }
      | fexpr STAR_TOKEN fexpr
{ $$ = $1 * $3; }
      | fexpr SLASH_TOKEN fexpr
          { $$ = $1 / $3; }
      | DASH_TOKEN fexpr %prec UMINUS
          { $$ = - $2; }
     | FLOAT_TOKEN
          { $$ = atof(yytext); }
      | float_function
     ;
```

Les precedences et associativites des operateurs sont donc definis dans la partie declaration, et les non-terminaux "fexpr" , "float" et "float_function" sont definis de %type <fval>. J'ai procede de la meme facon pour vector.

2.3.3 Flex++Bison++

En creant mon fichier grammaire, je me suis rendu compte d'une erreur de conception assez grave de ma part qui m'a fait perdre beacoup de temps par la suite (2 jours entiers) : j'utilisais Yacc et Flex, ce qui est tres bien quand on programme en C, mais notre projet est en C++, donc des que j'ai voulu inclure mes classes d'objects, probleme. par la syntaxe et la grammaire de flex et yacc, je savais qu'ils sortaient du C, mais j'etais sur de pouvoir encapsuler les fonctions yyparse et yylex dans une classe pour pouvoir m'en servir en C++, mais une fois arriver au point de no-retour, j'ai realise que c'etait infaisable dans le temps qu'il m'etait accorde. Je suis donc parti sur le net a la recherche d'equivalents de ces outils pour C++. J'ai trouve Flex++ et Bison++, qui sont des conversions faites expres : (ftp ://ftp.th-darmstadt.de/pub/ programming/languages/C++/tools/flex++bison++/LATEST/). Heureusement, la syntaxe des deux fichiers de regles avait ete conserve entierement entre les deux versions, mais certains details m'ont fait tourner en rond pendant des heures (notamment l'ordre dans lequel il fait inclure les fichiers headers des Objects et ceux generes par Bison++ et Flex++). C'est en examinant des exemples trouve sur le net que je reussi enfin a convertir mes anciens fichiers lex et yacc pour que Flex++ Bison++ les prennent sans probleme et que le tout compile en me donnant un parseur de base pour langage Pov.

2.3.3.1 PovLoader

Bison++ compile mon fichier de regles pour me donner PovParser.cpp et PovParser.h, qui contiennent une classe PovParser qui contient la fonction parse( ), qui est la fonction qui parse tout le fichier en se se servant de la fonction lex( ) qui est comprise dans PovLexer.h et PovLexer.cpp, generes par Flex++. J'ai cree un object Scene, qui contient les listes d'objects que contient la scene, et une camera. Cet object est utilise comme membre de la classe PovParser. Donc dans le code de PovParser (qui est defini dans les actions des regles du fichier yacc), l'object Scene est rempli au fur et a mesure du parsing. Voici un petit extrait de la grammaire camera :

```
cam_vector : LOCATION_TOKEN vector
                { scene.cam.setLocation($2); }
           | LOCATION_TOKEN error
                { printf("location <vector>\n");}
           | RIGHT_TOKEN vector
                { scene.cam.setRight($2); }
           | RIGHT_TOKEN error
                { printf("right <vector>\n"); }
```

```
            ...
           ;
```

On peut voir ici le remplissage de certains des parametres de la camera, et les messages d'erreur renvoyes lors d'erreurs de syntaxe. Pareil pour les objects :

```
sphere : SPHERE_TOKEN LEFT_CURLY_TOKEN vector COMMA_TOKEN float objmods RIGHT_CURLY_TOK
           {
             Sphere *nSphere = new Sphere($3,$5);
             scene.objectList.sphereList.add(nSphere);
           }
       ;
```

Une fois que le parseur est sur d'avoir la definition d'un object, il cree un nouvel object de ce type et le rajoute dans la liste d'objects de la scene. Pour finir, j'ai creer une classe qui herite de PovParser et PovLexer, appele PovLoader. Un programme qui inclut donc PovLoader.h est peut utiliser un object qui va parser l'entree et creer la scene en memoire. Cet object s'arret en cas d'erreur de syntaxe et vide la liste d'objects.

Henry a donc utiliser ces listes generes pour realiser un preview de la scene en OpenGL.

2.4 Le Pre-processeur

2.4.1 Description

2.4.1.1 Les Macros

Les macros sont un outil tres pratique lorsqu'il s'agit de programmer, ou plus specifiquement dans notre cas, ecrire des scenes POV complexes (le pre-processeur POV fonctionne comme un pre-processeur C ou C++). Par exemple, il suffit de de declarer une fois un objet pour pouvoir l'utiliser plusieurs fois dans la scene, sans devoir le redefinir a chaque fois.

Voici un exemple simple et pratique de l'utilisation du macro :

```
#declare Red = rgb <1,0,0>;
#declare Blue = rgb <0,0,1>;
#declare Green = rgb <0,1,0>;
...
sphere { <0,0,0>, 1.0
pigment { Red }
}
```

Ici, on definit une sphere de centre 0 et de rayon 1, en lui mettant une couleur uniforme rouge.

Le pre-processeur est donc un outil qui lit le fichier POV et qui s'occupe de construire une liste de tous les identifiants qui sont declares avec #declare,

avec leur texte correspondant. A chaque fois qu'il rencontre un identifiant se trouvant dans la liste, il le remplace par le texte qui a ete definit.

Ainsi "pigment { Red }" serait transforme en "pigment { rgb <1,0,0> }".

2.4.1.2 Les Includes

Autre fonction importante du pre-processeur : les includes. Quand on commence a avoir d'importantes collection de macros, il est souhaitable de les garder dans un fichier a part de le scene, pour pouvoir les reutiliser facilement en incluant le fichier quand on en a besoin, sans avoir a redefinir tous les macros. Comme en C, on utilise en POV la syntaxe "#include", suivi d'un nom de fichier entre guillemets.

Par exemple, si les macros que l'ont veut utiliser sont dans un fichier "colors.inc", il suffit de faire :

```
#include "colors.inc"
```

et vous pourrez ensuite les utiliser sans avoir a les redefinir.

2.4.1.3 Extension #zfor

Une extension speciale zRcube a ete ajoute au pre-processeur qui permet de declarer une suite d'objets en utilisant une boucle. Sa syntaxe est la suivante :

```
#zfor (n) TOKENS... #zend
```

La valeur n est en entier, et correspond au nombre de fois que sera copier le code POV defini par tous les tokens trouves entre #zfor et #zend. La variable de boucle que l'on peut utiliser a l'interieur de ce bloc est "zr". Cette variable sera instanciee de 0 a n en etant incremente a chaque recopie du code POV.

Par example, si l'on voulait faire une ligne de spheres selon l'axe x et une ligne selon z en meme temps on pourrait faire :

```
#zfor (10)
sphere { <zr*3,0,0>, 3 }
sphere { <0,0,zr*3>, 3 }
#zend
```

2.4.2 Implementation

Normalement un pre-processeur est aussi capable de detruire des entrees de sa liste ("#undefine") et de supprimer des parties de code a partir de conditions etablies ("#ifdef"), mais comme nous devions passer au Ray-tracer rapidement, je ne me suis pas atarde sur ces fonctions, et j'ai juste implemente le "#declare", le "#include", et la suppression des commentaires (/* ... */, //, comme en C++).

Ayant peu de temps pour le construire (2 jours), j'ai decide d'utiliser Flex (version C), vu que je commencais a bien connaitre. J'ai choisi donc de faire un programme a part du parseur, qui prend un fichier POV en entree et qui sort

un fichier “tmp.pov”. Le parseur appelle donc le pre-processeur, et si tout c'est bien passe, il parse le fichier “tmp.pov”.

Ce pre-processeur est tout simplement une fonction main qui appelle yylex. Cette fonction (cree par Flex) s'occupe de tout. Je vais maintenant expliquer la creation du fichier flex qui nous donnera cette fonction.

Le mode normal (initial) du pre-processeur peut detecter 5 elements principaux, voici les cotes gauches des regles Flex :

1. Les commentaires : “//” et “/*”
2. Les macros : “#declare”
3. Les includes : “#include”
4. La fin d'un fichier : <<EOF>>
5. Tous les autres characteres : .
6. Les identifieurs : {IDENTIFIER} (voir le rapport de soutenance 2 pour la definition)

Pour le 5. ces characteres sont tout simplement rediriges vers la sortie, donc le fichier “tmp.pov”.

2.4.2.1 Suppression des commentaires

Deja pour supprimer tout ce qui se trouve apres “//” (comme en C++), il faut une ligne en Flex :

```
"//".*\n     /* trim out comments */
```

Le commentaire a droite est simplement pour se rappeler a quoi sert la ligne en relisant le code, on aurait tres bien pu laisser le cote droit vide. D'ailleurs, du point de vue de Flex, il est vide. La ligne indique donc de remplacer “//” et n'importe quel charactere qui suit, jusqu'au charactere de nouvelle ligne '\n' par du rien.

Pour le “/* ... */”, il suffit de trouver les ouvertures “/*” et de mettre en action du code qui lit l'entree jusqu'a ce que la fermeture soit trouve (“*/”). Tous les characteres lus avec la fonction input() de Flex ne seront pas transmis vers la sortie, donc il y a besoin de rien faire de plus.

2.4.2.2 Les Macros

A chaque fois qu'un “#declare” est trouve, le pre-processeur passe en mode start condition “decl” avec la commande BEGIN de flex. Toutes les regles qui appartiennent a cette start condition commencent avec “<decl>”, par exemple, la regle qui supprime les espaces et les tabs est :

```
<decl>[ \t]*   /* eat up white spaces */
```

Ensuite il y a la regle qui trouve le cote gauche de la declaration, donc l'identifieur que l'on cree, des espaces et le signe egal “=” :

```
<decl>{IDENTIFIER}[ \t]*"="       { /* action code */ }
```

Le code d'action pour cette regle comporte trois parties. D'abord il faut extraire du yytext l'identifieur. Ce texte est garde dans une variable globale. Ensuite, il faut trouver le cote droit de la declaration. Pour faire cela, on lit les characteres d'entree en utilisant la fonction input() de Flex. A chaque fois que l'on trouve une accolade "{" on augmente un compteur, et quand on trouve une accolade fermante "}", on decremente le compteur. Quand le compteur arrive a 0 ou que l'on lise un point-virgule " ;", on sait qu'on est arrive a la fin de la declaration. Tous les characteres lus avec input sont concatenes dans une chaine temporaire. Ensuite vient la troisieme partie, qui consiste a re-scanner cette chaine. J'expliquerai cette partie plus bas, pour des raisons que vous comprendrez par la suite. Une fois ceci fait, l'identifieur et sa chaine correspondante sont ajoute a une liste chainee.

Quand Flex n'est pas en start condition "decl", et qu'il trouve un identifieur, il faut regarder dans cette liste si l'identifieur y est. S'il y est, on le remplace par sa chaine correspondante, sinon, on reecris cet identifieur vers la sortie du pre-processeur, le parseur s'occupera de savoir si c'est un identifieur valide ou non (c'est a dire s'il appartient au langage POV).

Un probleme survient donc si on veut utiliser un macro dans la definition d'un autre macro. Par exemple :

```
#declare White = rgb <1,1,1>;
#declare Gray50 = White * 0.5;
```

C'est pour cela que l'on doit re-scanner le cote droit. On doit donc sortir du mode "decl". Et il faut aussi indiquer a Flex que l'on scanne plus le fichier mais la chaine. Pour cela on sauvegarde le buffer courant YY_CURRENT_BUFFER, et on cree un autre buffer avec cette chaine en utilisant la fonction yy_scan_string. Cette fonction nous renvoye un pointeur de buffer Flex, qui est passe a la fonction yy_switch_to_buffer. L'analyse lexicale aura lieu maintenant sur cette chaine, mais il faut changer de start condition. On passe donc a la start condition "macro" avec la fonction BEGIN.

Dans ce mode, on prend tous les identifieurs et on regarde s'ils sont dans la liste chainee d'identifieurs. Si oui on met la chaine equivalente dans notre chaine temporaire, sinon on copie l'identifieur tel quel dans la chaine temporaire. Tout ce qui n'est pas un identifieur est copier directement dans la chaine temporaire. Lorsque Flex arrive a la fin de la chaine, on efface le buffer, et on repasse au fichier en utilisant yy_switch_to_buffer avec le buffer qu'on avait sauvgarder. C'est a cet endroit la que l'identifieur et sa chaine correspondante sont mis dans la liste chainee. On repasse ensuite en mode normal en faisant BEGIN(INITIAL).

2.4.2.3 Les Includes

Quand on trouve "#include" on passe en start condition "incl". Dans ce mode, il n'y a que deux regles :

```
<incl>[ \t]*\"        /* trim spaces and first quote */
<incl>[^ \t\n]+/\"    { /* action code */ }
```

La premiere mange les espaces et la premiere guillemet, car on sait que nous avons une chaine de la forme

```
#include “file.inc”
```

La deuxieme prend des characteres autres que des espaces et le charactere nouvelle ligne \n (la partie [^ \t\n]+), en sachant qu'il y a une guillement a la fin (la partie /\"). Le nom du fichier est a ce moment la dans yytext. On sauvegarde donc le YY_CURRENT_BUFFER dans une pile, et on cree un nouveau buffer avec yy_create_buffer, en lui donnant en parametre le pointeur du fichier que l'on ouvre avec fopen. Cette fonction cree un buffer Flex, qui est passe a yy_switch_to_buffer. Ensuite on indique qu'on veut repasser en mode normal : BEGIN(INITIAL). A partir de ce moment, le pre-processeur travail sur ce fichier, donc le parcours et ajoute a la liste chainee les macros qu'il rencontre.

Quand il arrive au charactere de fin de fichier <<EOF>>, il regarde si la pile de buffers est vide. Si c'est le cas, c'est que son travail est fini, donc on sort de la fonction yylex. Sinon, on efface le buffer en cours, et on pop la pile pour recuperer le dernier fichier sur lequel on travaillait. Ce buffer est remis en position avec yy_switch_to_buffer, et le pre-processeur continue d'ou il avait arrete sur ce fichier.

En sortant de yylex, on efface la liste chainee, et on sort du programme.

2.4.2.4 L'extension #zfor

Quand on trouve “#zfor”, on passe en start condition “zfor” avec BEGIN. Ensuite il n'y a que trois regles :

La premiere trouve le “n” (nombre d'iterations de la copie) :

```
<zfor>“(“+{INTEGER}+“)”    { /* action code */ }
```

A cette regle on associe du code qui prend la string qui correspond a l'entier, et qui la converti en un int avec la fonction atoi. Cet int est garde dans une variable globale “loop”.

La deuxieme prend tout le texte et le met dans une string :

```
<zfor>.    { strcat(zstr,yytext); }
```

Enfin, la troisieme regle est celle qui s'occpue de fermer le bloc et de le recopier n fois :

```
<zfor>#zend  { /* action code */ }
```

A cette regle on associe du code qui va recopier n fois la string zstr dans un fichier temporaire “zfor.tmp”. A chaque iteration, on recherche chaque occurrence de “zr” et on la remplace par l'index de boucle, pour que tous les “zr” soient bien instancies. Une fois ceci fait, on remet dans le text d'entree la chaine “ #include “zfor.tmp” ” avec la fonction flex unput. En faisant ceci, le pre-processeur ira lire le fichier propremement en remplacant les macros.

Chapitre 3

La libnet

3.1 Le besoin

Une rapide analyse de la parallelisation des algorithmes que l'on utilise montre que nous avons besoin que le serveur transmette au client une collection de valeur (un rayon, une pixel dans le cas du raytracing) et que le client doit renvoyer un ensemble de résultats (une couleur dans le cas du raytracing, un flottant dans le cas de la radiosité).

On aimerait en conséquence pouvoir demander au serveur "présente nous le résultat du calcul sur ces valeurs". Pour une utilisation la plus intuitive possible, cela devrait même être la seule fonction publique du serveur.

Du côté du client, la meilleure idée apparait être de lui donner la fonction de calcul, et qu'il lance lui même cette fonction sur les données qu'il reçoit.

De plus, nous n'avions pas encore implémenté ni le ray tracing ni la radiosité lorsque nous avons code cette librairie, en conséquence il aurait ete très ambitieux de dresser une liste exhaustive des valeurs que nous aurons à transmettre. Notre librairie devait donc être capable de transmettre sur le réseau des paquets de données de taille quelconques, celle-ci étant de type quelconque. On pourrait ainsi, si on le désirait, transmettre un int et deux float au client, celui-ci nous renverrait par exemple 3 float.

3.2 Organisation et algorithmes

3.2.1 Organisation

La description de notre besoin met en évidence que la nécessité d'au moins deux objets : un objet serveur et un objet client.

Cependant, serveur et client auront tous deux à utiliser de nombreuses fonctions très similaires : ouvrir une sockette, envoyer/recevoir des données dessus... D'o&urgave ; la nécessite de créer un objet socket, se plaçant comme une interface aux sockets BSD.

3.2.2 Le principe des patterns

Afin de pouvoir transmettre un nombre quelconque de données d'un type quelconque, nous devons spécifié le modèle des données transmises lors de l'initialisation du client et du serveur. Pour cela, nous avons mis en place un pattern sur le modèle de printf : une chaîne de caractère du type "%<type1> %<type2>" où <type> peut être n'importe lequel des types de base (int, float, char, double, string...). Ce pattern est compatible avec les chaines de formatage utilisée par printf et scanf. Pour chaque objet, il existe un pattern de réception et un d'émission, car rien n'oblige le client à renvoyer des données du même type que celles envoyées par le serveur.

3.2.3 La negociation

Avant de transmettre les données, le serveur doit s'assurer du type des données qu'il va échanger avec les clients. A la connection d'un client, le serveur débute donc une "négociation" : il demande au client ses differents patterns et vérifie qu'ils concordent avec les siens. Lors de cette négociation, nous comptons inclure les résultats d'une évaluation des performances du client, afin de permettre au serveur de répartir ses calculs en fonction de la vitesse de chaque client, mais cette partie n'a pas encore été codée

3.2.4 Le serveur

Nous voulons que le serveur tourne dès qu'il est initialisé, afin de pouvoir ajouter des clients sans interrompre l'ensemble du programme. De même il nous a paru interessant de permettre à des clients de se connecter alors que le calcul est déjà commencé et de les inclure dans la boucle de calcul. Pour cela, nous devons donc réaliser deux tâches en parallèle, ce qui nécessite l'utilisation de threads (l'utilisation de la fonction fork étant à proscrire étant donne qu'elle duplique la ram et donc ne permet pas le partage des données). Une introduction aux threads posix peut être trouvée dans [12]

Lors de l'initialisation du serveur, celui-ci commence donc par créer une sockette d'écoute, puis lance un thread qui se chargera d'attendre la connection d'un client. Pendant ce temps, le reste du programme continue à s'executer normalement. Si un client se connecte, la négociation est alors lancée, et en cas de réussite, le client est ajoute a une liste chainée circulaire maintenue par le serveur.

Lorsque l'on demande au serveur de calculer pour un ensemble de données, il commence par incrémenter sa liste de clients, puis envoie les données à ce client. Puisque la liste est circulaire, il envoie donc successivement les données à chaque client.

3.2.5 Le client

Le client est très simple : il se connecte au server, envoie ses patterns quand ce dernier les lui demande, puis a chaque fois qu'il reçoit des données, il leur

applique sa fonction de calcul et renvoie le résultat au serveur.

3.3 Implementation

3.3.1 Les donnees quelconques

Il existe peu de manières de coder en C++ des fonctions prenant un nombre d'argument quelconques en paramètres : on peut faire une liste chainée de void* ou utiliser les stdarg (definis dans <stdargs.h>). Si les deux méthodes sont similaires d'un point de vue interne, la seconde nous a parue plus propre. C'est, de plus, celle utilisée pour coder les fonction printf et scanf dans la libc.

Notre besoin est en fait très proche des fonctions printf et scanf de la libC : au lieu d'écrire/lire sur la sortie/entrée standard, on désire écrire sur des sockettes. Cette similitude nous a conduit à coder deux nouvelles fonctions printf() et scanf() définies dans la classe socket, dont la syntaxe est exactement la même que celles de la libc.

3.3.2 La gestion des paquets de donnees

Nous serons probablement amené à transmettre beaucoup de très petit paquets de données, et nous ferons tout pour limiter la quantité de données transmises afin de limiter le temps de communication. Or un en-tête tcp/ip a pour taille 20 octets, il apparait alors absurde d'envoyer un unique int, prenant 2 octets, a chaque fois .

D'où la nécessite de de regrouper les données en paquet. Nous avons pour cela mis en place un système de file : à chaque printf, la chaîne de caractère qui doit être envoyée est stockée dans une file (définie dans la classe Socket), lorsque la taille de l'ensemble des chaînes contenues dans la file excède une valeur limite (nous avons choisit 1024), l'ensemble de la file est envoyée d'un coup. Symétriquement, lorsque l'on reçoit un paquet, la file est recrée et à chaque appel de scanf, une chaîne est défilée. La reformation de la file lors de la récéption impose un séparateur entre chacune des chaînes (nous avons choisit '|'). Lorsque la file de récéption est vide, la sockette est automatiquement passée en mode récéption.

Ce principe des files permet de plus de gérer les différences de vitesses entre client et serveur : si un client n'arrive pas à traiter toutes les données envoyée par le serveur, celles-ci sont tout simplement enfilées (donc mise en attente), les données étant traitées dans leur ordre d'arrivée. Nous avons vu que les données étaient envoyées sans nécessiter un appel explicite de l'utilisateur. Cependant, il existe certaines situations où l'on doit transmettre la donnée immédiatement, sans attendre d'avoir remplit toute la file. C'est notamment le cas de la négociation, où le client doit envoyer son pattern (une chaîne par définition assez petite). Nous avons pour cela ajouté la fonction send() à la classe socket qui envoie le contenu de la file, même si celle-ci n'est pas pleine.

3.3.3 Le mini-protocole

Nous avons mis en place un “mini-protocole permettant de distinguer les types donnes envoyes, nous avons ainsi mis distingues trois type de donnee transitant sur entre nos clients et serveur :

- Les donnees : le valeur, dont le type est defini par le pattern
- Les messages : des messages d'information, le pourcentage de calcul effectue
- Les erreurs

Nous avons attribue un Prefixe a chacun de ces type de donnees : D pour les donnees , E pour les erreurs et M pour les messages, un packet libnet est donc constitue selon le modele :

<Prefixe> <donnee1> <donnee2> <...> |

Le type des donnees est filtre au niveau de la classe socket : suivant le prefixe, le message est stockee dans une file differente. Regulierement, le client doit faire des appels a socket.getError() et socket.getMessages() pour voir les derniers messages/erreurs qui lui ont ete envoyes.

3.4 Synchronisation

Lors des premiers tests realises grace a la librairie reseau avec plusieurs clients, nous avons souvent assiste a des situations ou le serveur attendait les resultats, et ou le client n'avait pas assez de donnees a envoyer (le client comme le serveur attend d'avoir remplit un paquet avant de l'envoyer) et donc etait lui aussi en etat d'attente. Il s'est donc avere necessaire d'envoyer un signal de fin de donnee que le serveur envoie a tous les clients. A la reception d'un tel signal, le client envoie toutes les donnees en attente. D'un point de vue implementation, il s'agit d'un comportement "reflexe" : le code du client n'a pas le choix, c'est la classe socket qui prend la decision. Ce "reflexe" permet de garder un niveau d'abstraction important du point de vue du code du client, qui n'a ainsi pas besoin de se synchroniser explicitement avec le server.

L'utilisation de ce signal a necessite l'implementation d'une routine *endComputing()* dans la classe serveur qui termine explicitement le calcul, on a donc du modifier la routine de calcul, qui est maintenant :

```
server.compute(donnes);
server.endComputing();
server.checkResults(&resultats);
```

Un autre probleme de synchronisation se pose lorsque les differents clients ne se connectent pas en meme temps (ce qui est la plupart du temps le cas) : certains clients peuvent avoir fini d'envoyer leur donnees alors que d'autre non. Il est donc inutile pour le serveur d'attendre des donnees des clients qui ont fini (puisqu'il ne doivent plus en envoyer) mais il doit attendre celles des clients qui n'ont pas fini. D'ou la necessite de garder un historique du nombre de donnees envoyees a chque client, et ainsi d'en deduire le nombre de donnees que le serveur doit attendre.

3.5 Problemes rencontres : utilisation des threads sous NetBSD

Lorsque nous avons voulu tester la libnet a l'école (elle avait été initialement developpée sous Linux), nous avons été surpris que, si aucune erreur n'etait generée, il ne se passait rien ! Lorsqu'un client se connectait, le serveur ne le prenait pas en compte. Nous avons commencé par regarder si cela n'etait pas un problème de réseau : La fonction d'attente du serveur aurait pu être bloquée sur la primitive accept suite a une mauvaise initialisation des sockets de notre part.

Cependant, après quelques tests, il s'est avéré que le thread ne se lançait tout simplement pas du tout. Nous avons donc commence à investiguer du cote des threads. Nous avons rapidement remarque que la lib pthread (le p signifie posix) de netBSD n'etait en fait qu'une interface a la librairie gnu pth (<pth.h>, le p signifie portable). Or, la documentation [16] de cette librairie précise que afin d'obtenir une portabilité maximum, cette bibliothèque permet de créer et de manipuler que des threads coopératifs, et non pas préemptifs (les threads posix correspondent à des threads préemptifs), contrairement a la librairie pthread disponible sous Linux (Linux Threads).

Notons les principales différences lorsque l'on code avec des threads coopératifs au lieu de thread préemptifs :

- un thread doit laisser explicitement la main au sheduler, celui-ci la donnant alors a un autre thread. Dans le cas de la Gnu Pthread, on peut notamment utiliser la fonction pth_yield(NULL). Notons que cette fonction n'existe pas dans les spécifications posix.
- toute fonction bloquante (par exemple la primitive accept() est par défaut bloquante) bloque l'ensemble du processus, et non pas juste le thread ou elle est lancée, comme dans le cas de thread préemptifs. La fonction sleep() arrête elle aussi l'ensemble des threads alors que dans un modèle base sur un multitâche préemptif, elle n'arrête que le thread ou elle est lancée.

Ainsi, un thread utilisant une boucle codée avec des threads posix :

```
while(1){ /*instructions*/ }
```

doit être transformée en

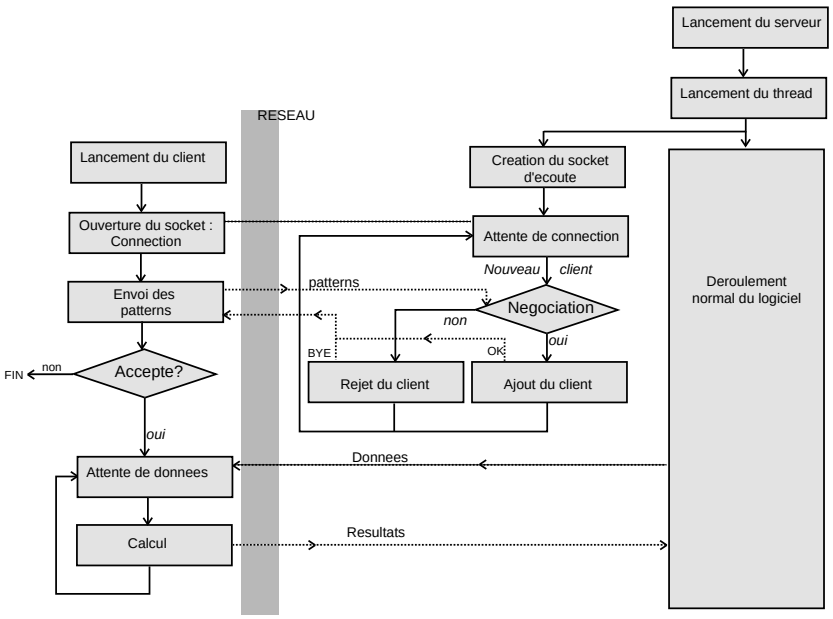
```
while(1){ /*instructions*/ pth_yield(NULL) ;}
```

Cette différence a nécessite de nombreux changement dans notre code pour être prise en compte.

Il est important de noter que des threads coopératifs sont inutiles a notre projet. En effet, rappelons les raisons qui nous ont pousse à utiliser des threads :

- la possibilité d'une abstraction supplémentaire et d'une souplesse de programmation en permettant de à la fois traiter l'arrivée d'un client, mais aussi de continuer les calculs. Avec les threads coopératifs, cela n'est possible qu'au prix de nombreux passage explicite de la main aux autres threads, ce qui revient quasiment à appeler à chaque boucle de calcul une fonction qui regarderait si un nouveau client etait connecte. Un effet similaire a l'utilisation de threads coopératifs aurait pu être d'utiliser le

FIG. 3.1 –



signal ALARM. Les threads coopératifs ne sont donc probablement pas la solution la plus simple pour un résultat équivalent dans ce cas.

– la possibilité d'utiliser des machines multiprocesseurs, chaque thread de calcul (clients virtuels) s'exécutant sur un processeur. Les threads coopératifs sont tous intègres a un même processus. En conséquence, ils ne peuvent être repartis sur plusieurs cpus, et donc sont inutiles.

En vue de ces nombreux désavantages, nous avons été tentes de recoder les parties utilisant les threads et en les recodant avec un algorithme itératif qui nous aurait assure un maximum de portabilité. Cependant, il aurait été dommage de priver les utilisateurs d'OS disposant d'une librairie gérant les threads préemptifs (linux, solaris ...) de la possibilité de repartir le calcul sur plusieurs cpus (les machines bi-processeurs étant de plus en plus courantes). En conséquence nous avons décide de continuer a utiliser les threads et de rajouter des pth_yield() inclus lorsque le projet est compile avec la librairie Gnu Pthread (via des directives de precompilation, la détection de la librairie étant faite par le script configure).

3.6 Conclusion

La figure 3.1 presente l'organisation de la libnet.

Chapitre 4

Le raytracing

4.1 Ray-Caster

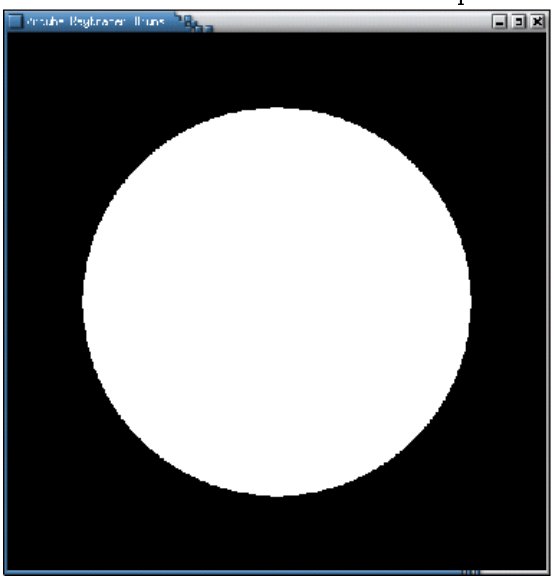
4.1.1 Introduction

Afin de bien preparer la partie raytracer, j'ai deja commence par faire de longues recherches sur internet pour y trouver les principes de bases et de mieux comprendre certains points qui restaient flous dans mon esprit. J'ai alors decide de me lancer dans un petit raycaster qui est un raytracer simplifie qui ne gere pas les recursions des rayons, il ne peut donc pas y avoir des reflections et des refractions. L'objet le plus simple a realiser comme nous l'avons deja souligne dans le precedent rapport est la sphere. Grace a l'equation mathematique de la sphere en coordonnees cartesiennes, il est facile de trouve le point d'intersection entre une demi-droite qui est notre rayon et cette sphere. Dans la majorite des cas, le nombre d'intersections entre une droite et une sphere est egale a 2. Il suffit alors de prendre le point le plus proche du point initial du rayon. Je decidai donc de voire tous les aspects du raytracing en passant par les spheres et afin d'avoir le maximum de connaissance sur le sujet, j'entrepris de gerer aussi les textures.

4.1.2 Interface

Avant de me lancer dans les calculs, je choisi deja l'interface qui allait le mieux se preter a ce programme. Mon choix s'est oriente vers la librairie *gtk/gdk* que j'avais deja utilise pour le programme de test de la librairie reseau. Cette librairie possede un composant qui se nomme *GdkRGB* qui represente un buffer de couleur RGB. La representation en RGB des couleurs etant fortement utilisee, il nous serait aise d'implementer les couleurs dans le raytracer. Comme le principe du lancer de rayon fonctionne sur le principe ou l'on envoie un rayon pour chaque pixel de l'image, il suffit donc de calculer la couleur du pixel et de la mettre dans le buffer. Une fois tous les points calcules, l'image peut etre afficher dans un fenetre *GTK* classique.

FIG. 4.1 – Le premier rendu.



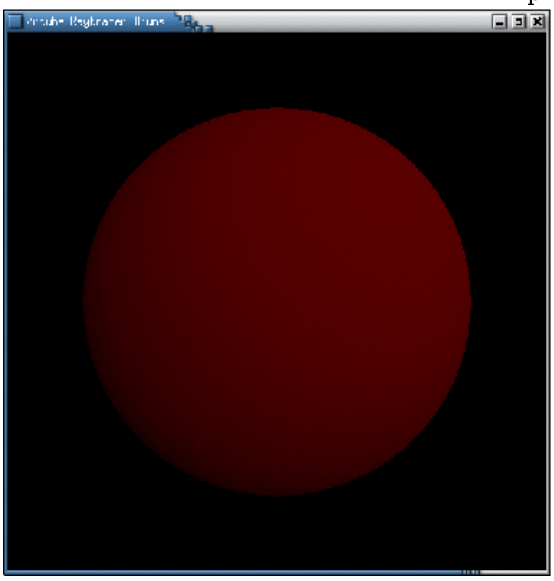
4.1.3 Les premiers tests

Des que l'implementation de l'interface fut terminee, je m'empressai de mettre en place mes fonctions des tests d'intersections avec une sphere. Pour les premiers tests, mon idee etait simple : je renvoyais la couleur blanche s'il y avait une intersection et la couleur noire s'il n'y en avait pas. Sans surprise, la premiere image fut toute noire. Apres quelques corrections, j'avais reussi a avoir un quart de disque blanc dans le coin en bas a gauche de mon image. J'aurais du garder en tete que dans le cas du raytracing, le repere par default est centre dans l'image. Il faut donc considere le point central de l'image comme le *<0,0,0>* du repere en trois dimensions. Grace a cela, j'obtins enfin mon premier objet *raycaste* que l'on peut voir sur la figure 4.1.

4.1.4 La lumiere

Les images rendues au debut ne donnaient en fait qu'une impression de 2D car la couleur etait uniforme sur toute la surface de la sphere. Pour avoir cet effet de 3D, il faut creer une source de lumiere qui n'est en realite qu'un point defini de l'espace. Plusieurs methodes d'eclairage existent, la premiere que j'ai realise est un modele ou la lumiere s'affaiblit selon la distance a laquelle on se trouve. Ce modele n'est pas correct et ne donne donc pas de bons resultats car les points situes en derriere la sphere par rapport a la lumiere doivent etre completement noirs. Le second modele que j'ai realise est beaucoup plus realiste. Il est base sur le produit scalaire entre le vecteur normal a la surface de l'objet au point d'intersection et le vecteur pointant vers la lumiere. On peut donc calculer le

FIG. 4.2 – Luminosite sans produit scalaire.



coefficient de luminosite grace a la formule :

$$lum = \overrightarrow{normal}.\overrightarrow{lumiere}$$

Ainsi, pour tous produits scalaires negatifs, l'eclairage doit etre nul. Ce mode de calcul d'eclairage donne de tres bons resultats que l'on peut coupler avec une petite attenuation selon la distance. Les deux differents resultats sont visibles sur les figures 4.2 et 41.

4.1.5 Les textures

L'aplication de textures sur un objet du type d'une sphere est un probleme tres simple a resoudre. Tout d'abord, il faut commencer par implementer les fonctions qui vont nous permettre de lire un fichier image. Je choisi le format *TIFF* dont les fonctions sont disponibles dans la librairie standard *libtiff*. Avant de me lancer dans les calculs de placage spherique de texture, je voulais tester un placage lineaire pour etre certain de mes fonctions de lecture de fichier images.

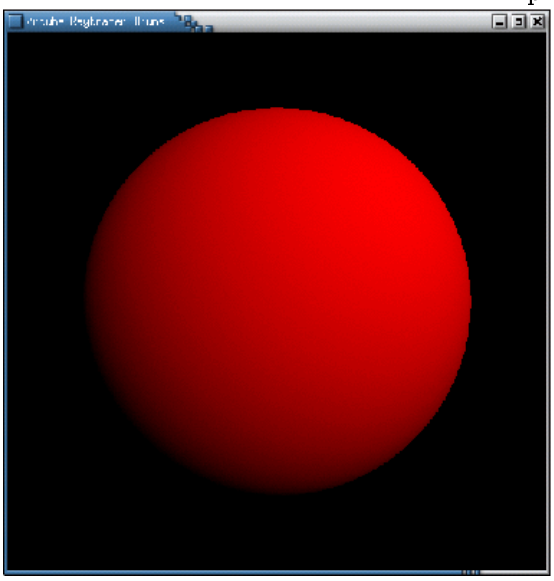
Dans le cas de textures carrees , il suffit de prendre la projection du point d'intersections un plan 2D comme le plan xOy. On calcule donc les coordonnees du pixel sur l'image grace aux formules :

$$dx = \|size + ix * u + w\| \, \%size$$

ou :

– dx represente le deplacement horizontal sur la texture

FIG. 4.3 – Luminosite avec produit scalaire.



- $size$ est le nombre de pixel horizontal de l'image
- ix est la composante x du point d'intersection
- u est le coefficient de multiplicite horizontal de la texture sur l'objet
- w est l'offset horizontal (le deplacement horizontal initial sur la texture)

et :

$$dy = \|size + iy * v + z\| \,\%size$$

ou :

- dy represente le deplacement vertical sur la texture
- iy est la composante y du point d'intersection
- v est le coefficient de multiplicite vertical de la texture sur l'objet
- z est l'offset vertical (le deplacement horizontal initial sur la texture)

Enfin, comme la plupart des representations des images sont lineaires, on obtient le numero du pixel grace a la formule :

$$pixel = (dy * 256 + dx)$$

Ainsi, on obtient les resultats visibles dans les figures 4.4 et 4.5.

4.2 Nouveaux elements de la classe Object (Guillaume)

Afin de pouvoir afficher des resultats aussi bien que POV, il fallait rajouter des informations concernant la texture et les proprietes des objets.

Object contient donc maintenant une structure Texture et une structure Interior.

FIG. 4.4 – Rendu avec le logo de l'ecole.

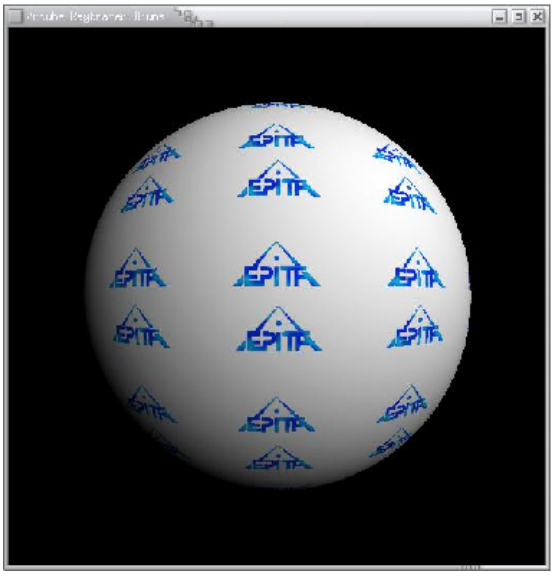
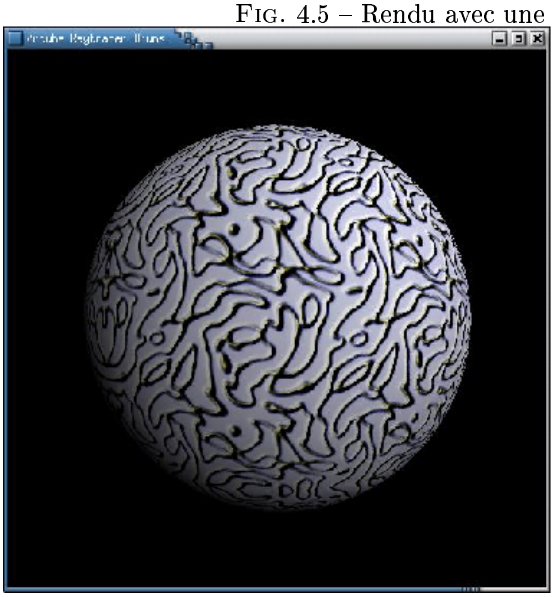


FIG. 4.5 – Rendu avec une texture 3D.



4.2.1 Texture

La structure Texture contient pour l'instant une structure Pigment et une structure Finish.

Le Pigment d'un objet defini sa couleur de base. POV permet de mettre des gradients de couleurs dans le Pigment, mais pour l'instant zRcube ne supporte qu'une couleur unique. La couleur contient les valeurs flottantes Red, Green, Blue, Filter, et Transmit. Filter et Transmit sont utilise pour la transparence (voir Rayon Refracte).

Le Finish d'un objet defini ses proprietes de reflection, ses coefficient de lumiere diffuse, speculaire, et phong, ainsi que son aspect (metallic ou plastique...). Ces proprietes seront expliques plus en detail dans la partie Couleur d'un Rayon.

4.2.2 Interior

La structure Interior decrit l'interieur de l'objet, elle contient son ior (index of refraction) et ses proprietes d'attenuation de la lumiere.

4.2.3 Transform

Une structure contenant la matrice de transformation de l'objet ainsi que sa matrice inverse.

4.3 Algorithme du Ray-Tracer

Nous utilisons une version un peu speciale de l'algorithme de ray-tracing, par rapport a celles rencontrees d'habitude. Ceci est du au fait que zRcube utilise le ray-tracing et la radiosite. La radiosite s'occupe des ombres, et des lumieres, donc le ray-tracing ne cree pas d'ombres.

Voici en pseudo-code l'algorithme que nous utilisons :

```
void RenderImage()
{
  for (y=0 ; y<image.hauteur ; y++)
for (x=0 ; x<image.largeur ; x++)
    {
      rayon = creerRayonInitial(x,y,camera) ;
      couleur = RayColor(rayon) ;
      image.putPixel(x,y,couleur) ;
    }
}
```

Cet algo va calculer la couleur de chaque pixel de l'image, en creant un rayon partant de la camera allant vers ce pixel. Voici en pseudo-code l'algorithme qui calcule la couleur d'un rayon :

```
Color RayColor(Ray rayon)
{
  pt = trouverPointIntersectionLePlusProche(rayon);
  if (pt==NULL) return Black;
  couleurLocale = calculCouleurLocale(pt);
  couleurRefl   = calculCouleurReflechie(pt);
  couleurRefr   = calculCouleurRefracte(pt);
resultante = couleurLocale + couleurRefl + couleurRefr;
  resultante -= attenuation(rayon);
  return resultante;
}
```

Cet algo trouve le point d'intersection le plus proche du rayon avec tous les objets de la scene. S'il n'y a pas de point d'intersection, on renvoye la couleur noir, cela veut dire que le rayon est sorti de la scene. Sinon, on calcule la couleur locale a ce point, la couleur du rayon reflechi a partir de ce point, et la couleur du rayon refracte a partir de ce point. On ajoute ces trois composantes et on applique une formule d'attenuation. Enfin, on renvoye la couleur resultante.

La couleur locale est calcule en fonction de :

1. La couleur de la texture de cet objet a ce point
2. L'angle entre la lumiere et la normale en ce point
3. Le coefficient de lumiere ambiente de cet objet
4. Le coefficient de lumiere diffuse de cet objet, qui agit avec le 2.
5. Le coefficient de lumiere speculaire de cet objet, qui agit avec le 2. et le vecteur vue.

Ces composants seront explique avec plus de precisions plus bas.

Pour calculer la couleur reflechie/refracte, on doit d'abord voir si les proprietes de l'objet permettent la reflection/refraction. Si oui, on calcul un rayon reflechi/refracte par rapport au rayon incident et la normale de l'objet au point d'intersection, et on relance la fonction RayColor avec ce rayon. Une fois la couleur de se rayon calcule, on la modifie par rapport aux proprietes de reflection/refraction de l'objet. Ceci lance donc un processus recursif qui se rappelle au maximum deux fois a chaque point d'intersection.

4.4 La classe RayTracer

Passons maintenant a l'implementation du ray-tracer.
Voici les membres prives de la class RayTracer :

```
class RayTracer {
private :
  // Scene
  Scene *scene;
```

```
  Octree *octree;
// Recursion level
  int traceLevel;
  // Image
  int imageWidth, imageHeight;
  guchar *rgbbuf;
  // Statistics
  ...
}
```

On rappelle que la classe Scene contient toutes les informations concernant la scene POV que rempli le parseur (camera, objets, proprietes general de la scene...). Donc avant de pouvoir commencer le trace de rayons, il faut utiliser un objet PovLoader pour parser la scene, et passer le pointeur de l'objet Scene du PovLoader a l'objet RayTracer (en utilisant la fonction interface "RayTracer : :setScene(Scene *scene)").

Pour l'instant l'octree n'est pas utilise, mais cela sera pareil, il faudra construire un octree avec les objets de la scene, et le passer a l'objet RayTracer (en utilisant la fonction interface "RayTracer : :setOctree(Octree *octree)").

La variable traceLevel est un compteur qui indique a quel niveau de recursion nous nous situons pendant le trace de rayon. Ceci pour pouvoir limiter la recursion pour des raisons qui seront expliquees plus bas.

Ensuite, les proprietes de l'image que le RayTracer doit rendre; sa taille, et le buffer RGB ou sera rendu l'image et afficher dans la fenetre gtk.

Finalement, les variables de statistiques, qui ne sont pas montres ici vu leur nombre. Il s'agit de tenir compte du nombre de rayons initiaux/ reflechis/r efractes/ transmis calcules, du nombre d'intersections avec tous les types objets, le niveau de recursion le plus haut atteint...

La taille de l'image peut etre redimensionee en utilisant la fonction "RayTracer : :redimImageBuffer(int w,int h)", ensuite un appel a la fonction "RayTracer : :renderImage(void)" lance le ray-tracing. Une fois termine, on peut recuperer le buffer image avec la fonction "RayTracer : :getBuffer(void)" qui renvoy le pointeur du buffer. Ce buffer est affiche dans une fenetre gtk (une fonction dumpToFile est prevu, pour pouvoir sauvegarder l'image).

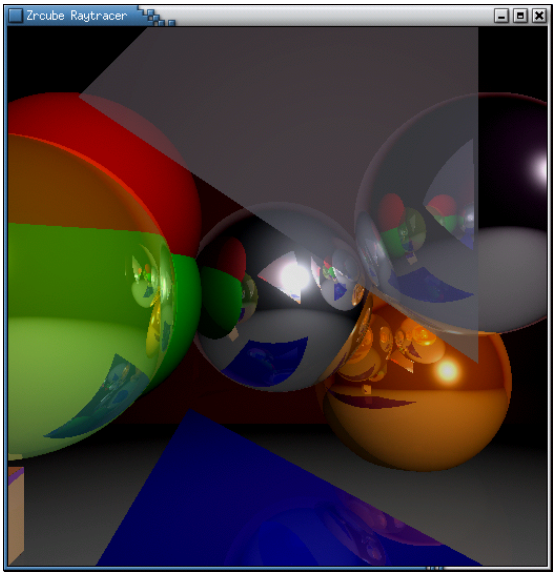
On peut aussi afficher les statistiques en appelant la fonction "RayTracer : :coutStats(void)"

4.5 Rayon initial

4.5.1 La camera dans POV

La camera est dans l'absolue le point d'ou l'on regarde la scene. Pour satisfaire la syntaxe POV, la camera doit gerer plusieurs attributs. Tout d'abord, nous devons specifier le type de camera, celui par default etant la projection avec perspective. Ce parametre va agir sur la direction du rayon qui va etre lancer a

FIG. 4.6 – Rendu avec angle de 60.



partir de l'eye. Le second parametre est la position de la camera. Il s'agit tout simplement du point a partir duquel on lance les rayons. Deux parametres qui sont respectivement *up* et *right* servent a donner le rapport de la longueur et de la largeur de l'image, et le sens de celle-ci. Le parametre *sky* permet quant a lui de definir le sens de la camera. Il est le guide pour orienter la face superieure de la camera. Enfin, le dernier parametre de base est le vecteur de direction. Il indique la direction generale dans laquelle regarde la camera. Tous ces parametres de bases ont ete implementes dans la version actuelle du raytracer.

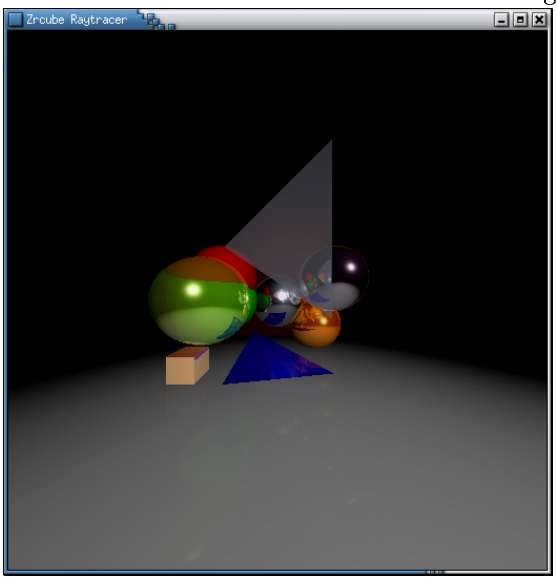
D'autres parametres sont aussi tres important parce qu'ils agissent sur les parametres de bases. Ce sont des paramtres de modification. Le premier est le vecteur *look-at* qui permet de specifier a quel endroit precis la camera va pointer. Ce parametre, pour qu'il soit actif doit etre placer apres les parametres de base. Le dernier parametre est nomme *angle*. Il permet d'elargir ou de diminuer le champs de vision. Plus concretement, il va permettre de zoomer plus ou moins sans changer le point de position de la camera. On peut voir sur les images suivantes l'effet du parametre *angle* :

4.5.2 Calcul du rayon initial

Tous les parametres de la camera etant prepares par le parseur, il ne reste plus qu'a calculer le rayon initial pour le pixel x,y. Voici le procede en pseudo-code :

```
i = x - (imageWidth/2);
j = y - (imageHeight/2);
```

FIG. 4.7 – Rendu avec angle de 130 .




```
ray.init = cam.location;
i0 = i / imageWidth;
j0 = j / imageHeight;
ray.direction = cam.direction + cam.right*i0
              + cam.up*j0;
ray.direction.normalize();
```

4.6 Rayon reflechi

4.6.1 Reflection dans POV

La reflection dans POV est controle par le mot-clef "reflection". La reflection est un membre de la partie Finish de l'objet. C'est une couleur, qui definie la couleur par laquelle sera multiplie la couleur obtenue par le rayon reflechi.

Par exemple regardez cette sphere :

```
sphere { <0,0,0> 1.0
         pigment { Red }
         finish  { reflection 0.5 }
}
```

Cette sphere sera rouge, et reflechira son entourage en reduisant la lumiere recu de moitie.

4.6.2 Calcul du rayon reflechi

La reflection d'un rayon sur une surface est base sur le principe fondamental des loi de *Descartes*. Il s'agit de faire repartir un nouveau rayon au point d'intersection du precedent. L'angle de depart du nouveau rayon est identique a l'angle d'arrive du rayon incident. Pour cela, on calcul le vecteur normal au point d'intersection et on obtient le vecteur reflechi grace a l'une des deux formules equivalentes :

$$\overrightarrow{R} = 2 * \overrightarrow{N} * cos(\theta_i) - \overrightarrow{I}$$

et

$$\overrightarrow{R} = 2 * \overrightarrow{N}(\overrightarrow{N}.\overrightarrow{I}) - \overrightarrow{I}$$

avec :
- $\overrightarrow{R}$ = le vecteur norme reflechi.
- $\overrightarrow{N}$ = le vecteur normal a l'interface orientee dans le materiau ou se propage le rayon incident.
- $\overrightarrow{I}$ = le vecteur incident inverse.
- θ_i = l'angle entre le vecteur incident et le vecteur normal.

4.7 Rayon refracte

4.7.1 Refraction dans POV

La refraction a lieu quand un objet est transparent et que l'indice de refraction (ior) de son interieur est different de celui d'ou vient le rayon. Le rayon passe alors a travers mais avec un angle modifie. Si l'ior est le meme, le rayon est dit transmis, il passe a travers sans changer d'angle. Il se peut que le rapport des iors fasse que le rayon soit emprisonne dans l'objet, ceci s'appelle la reflection totale interne. Si cela se produit, on calcul un rayon reflechi et on renvoye sa couleur.

Dans POV, il y a des termes qui controlent la transparence : filter et transmit. Ce sont les 4eme et 5eme elements de l'objet Color. La transparence est donc etablie dans les composantes filter et transmit de la couleur dans Pigment.

La composante filter indique que la lumiere qui passe a travers de l'objet sera teinte. Par exemple, si une lumiere grise telle que rgb<0.9,0.9,0.9> passe a travers un filtre tel que rgbf<1,0.5,0,1>, la lumiere resultante est rgb<0.9,0.45,0>.

La composante transmit indique quelle pourcentage de lumiere passe a travers l'objet sans etre modifie.

Voici un exemple d'un objet transparent :

```
sphere { <0,0,0> 1.0
         pigment  { Red transmit 0.8 }
          interior { ior 1.3 }
}
```

4.7.2 Calcul du rayon refracte

Le rayon refracte est calcule avec la formule suivante, deduite de la loi de Descartes :

$$\overrightarrow{T} = \left[n(\overrightarrow{N} \cdot \overrightarrow{I}) - \sqrt{1 - n^2(1 - (\overrightarrow{N} \cdot \overrightarrow{I})^2)} \right] \overrightarrow{N} - n\overrightarrow{I}$$

Avec :
- $\overrightarrow{N}$ = le vecteur normal a l'interface orientee dans le materiau ou se propage le rayon incident.
- $\overrightarrow{I}$ = le vecteur incident inverse.
- n = ni / nt (avec ni l'ior du milieu d'ou vient le rayon incident, et nt l'ior de l'interieur de l'objet dans lequel il est transmit).

Quand on a calcule rayon refracte, on met a jour son pointeur d'Interior pour qu'il pointe vers l'objet dans lequel il se trouve actuellement. Le rayon a en fait un tableau de pointeurs d'Interior, et un indice de tableau. Si l'indice est a -1, cela veut dire que le rayon est dans l'air libre. L'indice est donc incremente quand un rayon entre dans un objet et decremente quand il en sort.

4.8 Intersections Rayon/Objet

4.8.1 Objets supportes

La classe Object a une fonction virtuelle s'appellant "bool intersect(Ray *ray, Vector *point, Vector *pnormal)". Cette fonction doit etre definie pour tous les objets derives de la classe Object pour que l'algorithme de ray-tracing puisse tous les afficher correctement. Elle prend en parametres un rayon, un pointeur vers le point d'intersection a remplir, et un pointeur vers la normale a ce point a remplir. Elle renvoye vrai ou faux selon la presence d'un point d'intersection entre le rayon et l'objet en question.

Pour l'instant la fonction intersection a ete implemente pour les objets suivants : sphere, plane, triangle, box.

Pour l'instant l'octree n'est pas utilise, donc on utilise une fonction qui trouve le point d'intersection le plus proche parmis tous les objets de la scene. L'algorithme utilise est tres simple, mais tres lent s'il y a beaucoup d'objets dans la scene (d'ou l'utilite de l'octree). Il s'agit de parcourir tous les objets en appellant la fonction intersect a chaque fois. Si celle-ci renvoye vrai, on regarde la distance entre le point d'intersection trouve et le point de depart du rayon. Si cette distance est plus petite que la plus petite distance (que l'on initialise a INFINI au debut), on la remplace, et on garde ce point d'intersection, le pointeur vers l'objet, et la normale. A la fin de la boucle on a le point d'intersection le plus proche du depart du rayon, on sait a quel objet appartient ce point, et on a la normale de cet objet en ce point. Toute cette information est renvoyee a la fonction RayColor (voir la partie Algorithme).

4.8.2 Heuristique pour les points proches

Cependant il y a un probleme avec cette methode. Imaginez un cube transparent, pose sur un plan. Un rayon entre dans le cube par le haut, et va toucher le bas du cube. Quand on cherche le point d'intersection le plus proche, le point sur le plan et le point sur le bas du cube sont les memes. Donc si le plan est traite apres le cube dans la boucle, c'est le point sur le plan que l'on conservera. Or il faut que le rayon touche d'abord le bas du cube. Si l'on conserve le point sur le plan, la couleur renvoyee sera celle du plan a cet endroit et non celle du cube. Ceci nous donne des artefacts de rendu horribles, donc il a fallu trouver une solution a ce probleme (solution qui sera aussi utilisee avec l'octree). Voici cette solution :

Si la distance calculee est plus petite d'une valeur EPSILON (definie entre 0.1 et 0.0001, selon l'echelle de la scene) que la plus petite distance courante, alors on remplace. Mais si cette distance est a peu pres egale a la plus petite distance, c'est a dire que la difference est superieure a -EPSILON et plus petite que EPSILON, on remplace si certaines conditions sont remplies.

Ces conditions constituent une heuristique, les voici, en ordre de priorite :

- Si le rayon n'est pas a l'interieur d'un objet, on garde ce point.
- Si le rayon est a l'interieur de l'objet intersecte, mais en dehors de tout objet qui est lui meme a l'interieur de l'objet intersecte, alors on garde ce point.
- Si le rayon est a l'interieur d'un objet et que l'objet intersecte est un objet de type PATCH (c'est a dire qu'il na pas d'interieur, comme les triangles, les plans...), alors on ne garde pas ce point.
- Si on arrive jusqu'ici sans avoir pu decider, on garde le point.

Dans le cas de notre cube pose sur le plan, le point sur le plan sera rejete, alors que le point sur le bas du cube sera conserve. Ceci elimine tous les artefacts de ce type.

Il est facile d'evaluer ces conditions, car le rayon a un pointeur vers l'interieur de l'objet (chaque Object a un membre Interior, qui est une structure comportant la description de l'interieur) dans lequel il se situe.

4.8.3 Problemes d'echelle

Cette valeur EPSILON est difficile a bien choisir. Si nous avons une scene de grande echelle, avec la camera placee loin, un petit ecart sur l'ecran se traduit par un grand ecart dans la scene. Donc si nous avons une scene a petite echelle, une valeur de 0.0001 serait bien, mais si on conserve cette valeur en rendant une scene a grande echelle, on voit apparaitre quelques artefacts. Une methode serait peut-etre de changer EPSILON lors de ses utilisations en fonction de la distance de la camera par rapport aux objets de la scene, mais nous n'avons pas encore teste.

4.9 Couleur d'un rayon

4.9.1 Couleur locale

Comme je l'avais explique plus haut, la couleur locale est la somme de plusieurs composantes :

Ambiente + Diffuse + Speculaire

Ceci est le modele globale d'illumination qui est normalement utilise. Voici une description des termes :

Ambient : Intensity = $ka \cdot Ia \cdot c$
avec Ia l'intensite ambiente de la lumiere, ka le coefficient ambient de l'objet et c la couleur de l'objet.

Diffuse : Intensity = $kd \cdot I \cdot c \cdot (\overrightarrow{L} \cdot \overrightarrow{N})$
avec I l'intensite de la lumiere, kd le coefficient diffuse de l'objet, $\overrightarrow{L}$ la direction unitaire du vecteur lumiere, et $\overrightarrow{N}$ le normal a l'objet au point d'intersection, et c la couleur de l'objet.

Speculaire : Intensity = $ks \cdot I \cdot (\overrightarrow{V} \cdot \overrightarrow{R})^n$
avec ks le coefficient de reflection speculaire de l'objet, I l'intensite de la lumiere, $\overrightarrow{V}$ le vecteur vue, $\overrightarrow{R}$ la direction unitaire du rayon reflechi, et n le facteur de brilliance du reflet.

POV utilise ce modele avec quelques ameliorations. Un objet completement transparent (filter+transmit $\geq$1) verra son terme ambient mis a zero, pour que l'objet soit reelement transparent et que l'on ne puisse pas voir sa couleur. Ceci va de meme pour le terme diffuse. Le terme diffuse peut aussi etre modifier pour donner une apparence plus brilliante a l'objet, ceci est controle par le terme Finish.brilliance de l'objet. Ils ont rajoute une autre sorte de reflection speculaire controle par le terme phong. Voici les changements apportes :

Ambient : Intensity = $ka \cdot Ia \cdot c \cdot att$
avec att = (1.0 - MIN(c.filter + c.transmit,1.0)), ka = Finish.ambient, c = Pigment.color.

Diffuse : Intensity = $kd \cdot I \cdot c \cdot att \cdot (\overrightarrow{L} \cdot \overrightarrow{N})^b$
avec kd = Finish.diffuse, b = Finish.brilliance.

Phong : Intensity = $p \cdot C \cdot (\overrightarrow{R} \cdot \overrightarrow{L})^{ps}$
avec p = Finish.phong (le coefficient de reflection phong), ps = Finish.phongSize (la taille de l'effet), C la couleur de la surface (Pigment.color) ou de la lumiere, dependant de Finish.metallic.

Speculaire : Intensity = $s \cdot C \cdot (\overrightarrow{H} \cdot \overrightarrow{N})^{1/r}$
avec s = Finish.specular (le coefficient de reflection speculaire), r = Finish.roughness (taille de l'effet), C la couleur de la surface (Pigment.color) ou de la lumiere, dependant de Finish.metallic, et $\overrightarrow{H} = (\overrightarrow{L} - \overrightarrow{V}) / \sqrt{(\overrightarrow{L} - \overrightarrow{V}) \cdot (\overrightarrow{L} - \overrightarrow{V})}$ (le vecteur bissection entre $\overrightarrow{L}$ et $\overrightarrow{V}$).

4.9.2 Couleur du rayon reflechi

La couleur du rayon reflechi est tout simplement la couleur locale (retourne par RayColor) au point d'intersection du rayon, multiplie par la couleur de reflection (Finish.reflection) de l'objet d'ou est parti le rayon.

4.9.3 Couleur du rayon refracte

La couleur du rayon refracte est un tout petit peu plus complique :

RefrColor : $localColor \cdot ((c \cdot c.filter) + c.transmit)$

C'est en fait la coleur locale (retourne par RayColor) au point d'intersection du rayon, multiplie par une fonction de la couleur de l'objet d'ou est parti le rayon avec ses coefficients de transparence transmit et filter.

4.9.4 Modifications Radiosity :

Quand la radiosite est prise en compte, l'equation est un peu modifiee. La composant diffuse n'est pas prise en compte car c'est la radiosite qui s'en occupe, le terme ambient est change en :

Ambient : Intensity $= ra \cdot Ia \cdot c \cdot att$
avec att = (1.0 - MIN(c.filter + c.transmit,1.0)), c = Pigment.color, ra est la couleur renvoye par la radiosite sur ce point, plus d'informations seront apportes dans la partie Radioste.

4.10 Attenuations et optimisations

4.10.1 Distance Attenuation

Les rayons de lumiere sont normalement attenues par la distance qu'ils parcourent. Donc pour avoir des effets realistes avec un ray-tracer, il faut attenuer la couleur resultante que calcule RayColor.

Ici aussi, la fonction devrait dependre de l'echelle de le scene pour donner toujours des resultats convenables, mais pour l'instant nous utilisons : resColor -= A, avec

$A = C \cdot ray.init.distance(pt)^2$

$C = 1 \cdot 10^{-8}$

pt etant le point d'intersection du rayon. En fait C devrait etre en fonction de la distance entre l'origine de la camera et la moyenne des objets de la scene. Cette methode sera teste lorsque nous utiliserons l'octree (pour recuperer le centre de la scene sans perdre de temps).

4.10.2 TraceLevel

Le traceLevel est un compteur qui indique a tout moment le niveau de recursion auquel nous nous trouvons dans les appels a RayColor (le premier appel est

au traceLevel 0). Pour pas que l'on mette une heure a rendre une scene simple, on peut limiter le niveau de recursion, avec la variable maxTraceLevel, qui est controle dans l'objet Global_settings de la scene. En general un niveau maximum de 5 a 7 donne des resultats convenables sans que le rendu prenne trop de temps. Evidement quand on travaillera en multi-thread/multi-cpu nous pourrons augmenter ce niveau maximum. Cette fonctionnalite est aussi tres pratique pour debugger le code pendant les premieres etapes de la creation du ray-tracer.

4.10.3 Weights

A moins qu'on rajoute un systeme d'attenuation sur les rayons reflechis/refractes, si on a un MaxTraceLevel eleve, les reflections internes peuvent vite tout ralentir, tout en etant en saturation, c'est a dire renvoyant de la lumiere blanche, comme un effet larsene dans un ampli. POV utilise un systeme de poids pour contrer cet effet, et accelerer un peu le rendu.

Les rayons initiaux sont donnes un poids de 1. A chaque fois qu'un rayon reflechi ou refracte est calcule, ils prennent le poids de leur rayon initiateur, mais attenue. Au debut de RayColor, on regarde si le poids est en dessous d'une certaine valeur seuil, si c'est le cas, on renvoye du noir. Si on est reste dans RayColor, on multiplie la composante locale calcule par le poids du rayon. Ceci cree un systeme performant d'attenuations.

De plus cela permet de donner plus de validite physique au phenomene de reflection/refraction, car la lumiere reflechie/refracte est toujours attenue, a moins qu'on est un miroir parfait ou un verre parfaitement transparent.

Voici comment on calcule le nouveau poids pour chaque type de rayon :

Attenuation newWeight = weight * MAX3(RC.red, RC.green, RC.blue) ;

Attenuation de la refraction : newWeight = weight * MAX(w1, w2) ;
avec w1 = pigColor[3] * MAX3(pigColor[0], pigColor[1], pigColor[2]) ;
et w2 = pigColor[4] ;

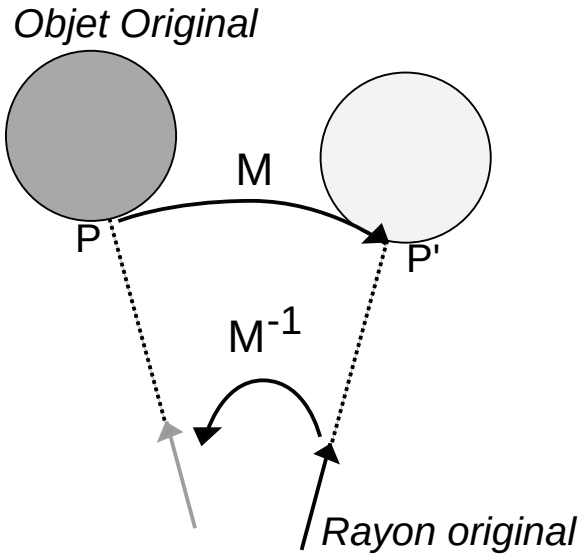
avec weight etant le poids du rayon initiateur, RC la propriete Finish.reflection de l'objet d'ou part le rayon reflechi/refract, et pigColor la propriete Pigment.color de cet objet.

4.11 Transformations

Pour prendre en compte les transformations (scale, translate, rotate) des objets, on utilise les deux matrices stockees dans la class Transform qui appartient a la classe Object.

Le principe est tout simple, lorsque les tokens scale, translate ou rotate sont trouves dans le parseur, celui-ci met a jour la matrice de transformation et sa matrice inverse de l'objet en question, mais on ne touche pas aux proprietes de l'objet. Ensuite, lorsque l'on teste l'intersection d'un objet avec un rayon, on applique la transformation inverse au rayon. Si ce rayon transforme intersecte avec

FIG. 4.8 – Transforms



l'objet en question, on applique la transformation directe au point d'intersection pour retrouver le point d'intersection dans la base originale du rayon.

4.12 Textures

La composante pigment d'un objet n'est pas oblige d'etre une couleur uniforme, on peut aussi applique des textures externes ou des des degrades de couleurs appliques a des fonctions que l'utilisateur peut choisir. La classe Pigment contient donc une fonction doPointColor qui prend le point d'intersection en parametre et qui s'occupe de mapper une texture ou une fonction (ou une couleur uniforme) au point. Le raytracer appelle donc cette fonction pour avoir la couleur locale, a laquelle il rajoute les couleurs des rayons reflechis et refractes pour avoir la couleur finale.

4.12.1 Color Map

Voici comment on declare un color map en POV :

```
sphere { <0,0,0>, 3
         leopard
         pigment {
          color_map {
               [0.0 Black]
               [0.5 White]
               [1.0 Black]
```

```
            }
          }
    }
```

Le mot-clef leopard declare le type de fonction qui sera applique au degrade de couleurs Noir-Blanc-Noir.

Chaque fonction de pattern (leopard, gradient, mandelbrot, onion) prend un point d'intersection en parametre et renvoye une valeur flottante "pf" entre 0 et 1. Cette valeur sera utilise pour trouver la couleur correspondante dans le degrade. Le degrade est fait avec une interpolation lineaire entre les valeurs immediatement inferieures et superieures de "pf".

Il faut donc implementer une fonction pour chaque type de pattern. Nous supportons a cette date Onion, Gradient, Leopard et Mandelbrot. Voici, a titre d'exemple d'implementation, la fonction Leopard :

```
float Pigment : :doLeopard(Vector ipt)
{
  float value,tmp1,tmp2,tmp3;
  tmp1 = sin(ipt[0]);
tmp2 = sin(ipt[1]);
  tmp3 = sin(ipt[2]);
  value = SQR((tmp1 + tmp2 + tmp3) / 3.0);
  return value;
}
```

Les transformations peuvent etre applique en appliquant ce qui s'appelle un Warp. Le Warp est similaire a la methode utilise pour transformer les objets, il s'agit de transformer le point d'intersection par la matrice de transformation inverse et de calculer la couleur de texture en ce point.

4.12.2 Image Map

Voici comment on utilise les image_map dans POV. On doit d'abord specifier le type d'image. Les types supportes par zrcube sont le png, le jpeg, et le ppm.

```
plane { <0,1,0>, -3
 pigment {
   image_map { png "dirt.png" }
   rotate <90,0,0>
   scale <40,40,40>
 }
}
```

Il y a plusieurs types de mapping : le planaire, le spherique et le cylindrique. Ils sont declare dans le bloc image_map avec la mot clef "map_type" suivi d'un entier entre 0 et 2.

Comme les color map, on peut utiliser des transform pour modifier l'aspect de la texture. Ceci est particulierement important pour plaquer des textures sur des plans en mapping planaire, car il faut pouvoir tourner la texture avec rotate pour qu'elle soit projete sur l'axe du plan.

4.13 Antialiasing

4.13.1 Detection de countour avec filtre de moyenneur

La premiere technique utilise est un algorithme de lissage qui utilise un filtre moyenneur. Il s'agit de calculer les gradients entre les pixels connexes d'un pixel central. Si ces gradients sont au dessus d'une valeur seuile, la couleur du pixel central recoit la moyenne des couleurs des pixels connexes. Cette methode cree donc une perte d'information qui ne donne pas de resultants satisfaisants ; l'image est plus flou sans qu'elle soit reelement anti-aliasee.

4.13.2 SuperSample

La deuxieme methode essaye est le supersampling. Il s'agit de relancer quatre rayons autour du rayon initial du point i,j si la difference entre la couleur du pixel au gauche ou au dessus de i,j avec la couleur du pixel i,j est superieur a une valeur seuil. La difference de couleur est calcule en sommant les valeurs absolues des differences des composantes :

$$\triangle = abs(C1.rouge - C2.rouge) + abs(C1.vert - C2.vert) + abs(C1.bleu - C2.bleu)$$

Les quatre rayons relances sont choisies aleatoirement dans un cadre autour de i,j en utilisant la fonction random. On fait ensuite une moyenne des quatre couleurs renvoyes et cette moyenne est la nouvelle couleur de i,j.

Chapitre 5

La radiosite

5.1 Radiosite hierarchique

5.1.1 La radiosite

5.1.1.1 Idee

Imaginons une salle avec quatre murs rouges et un sol blanc. En toute logique , le sol devrait être un peu rouge, car selon le modèle physique de la couleur, les murs rouges émettent de la lumière rouge. Or l'algorithme de ray tracing ne prend pas en compte cette interaction entre les surfaces, le but de la radiosité est de modéliser ce comportement afin d'obtenir un modèle d'illumination le plus proche de la réalité possible.

5.1.1.2 Une premiere approche

Puisque chaque surface émet de la lumière , on doit la considérer comme une source de lumière, mais elle reçoit aussi la lumière emise par toutes les autres surfaces, on doit donc la considérer aussi comme un capteur de lumière. On est donc amenés à associer à chacune d'elles deux valeurs : la quantité de lumière reçue (son « illumination ») et la quantité de lumière en surplus qu'elle va émettre vers les autres (son « énergie radiative »).

Cependant, si l'on revient à un exemple concret, il apparaît évident qu'une surface, par exemple une table, ne reçoit pas une quantité uniforme de lumière. Dans le cas idéal, il faudrait diviser notre surface en surfaces de taille infinitésimales, ce qui reviendrait a représenter chaque surface sous la forme d'une intégrale et par conséquent a représenter l'ensemble des surfaces sous la forme d'une integrale. On appelle form factor la fonction donnant la quantite d'energie donnée par une surface i à une autre.

On peut regrouper ces concepts de « balance d'énergie » dans la relation de radiosité pour le point i et par rapport a son environnement env, en notant que la radiosite d'une surface est la quantite de lumiere reflechie par unite d'aire.

On appelle de plus *form factor* la fonction qui calcule l'interaction entre deux patches *i,j*, en fonction notamment de leur distance.

$$Radiosite(i) = Emission(i) + Reflectivite(i).\int_{i\,\epsilon\,env} Radiosite(j).FormFactor(i,j)$$

Avec :

Radiosite(i) $= B_i$ = Total de l'énergie quittant la surface. (homogène avec des $W.m^{-2}$)

Emission(i) = E_i = Le niveau d'énergie émis par la surface (homogène avec des $W.m^{-2}$)

Réflectivité(i) = p_i = La fraction de l'énergie qui est reflechie (sans dimension)

$\int_{env}$= intégrale sur l'ensemble des surfaces.

Formfactor(i,j) = $F_{i,j}$

Les algorithmes de radiosite classiques sont une approximation du modele physique, elle suppose que l'on travaille avec des surfaces dites *lambertiennes*, parfaitement diffuses, et que les surfaces sont suffisament petites pour les considerer comme ayant une radiosite constante : on ramene ainsi l'integrale sur l'environnement a une somme sur les petites surfaces composant cette environnement. Chacun de ces petits elements de surfaces sont appelles "*patch*", ou *surfel* en francais. Il est evident que plus ces patches sont petits, plus la qualite de l'approximation est grande et donc plus l'illumination calculee sera realiste.

Classiquement l'equation de radiosite est definie par :

$$Bi.A_i = E_i.A_i + p_i.\sum_{k=1}^{n} B_k.F_{k,i}$$

Ou

n est le nombre de patches

B_i est la radiosite du i-eme element ;

A_i l'aire du i-eme element ;

E_i la lumiere emise, autrement dit 0 si il ne s'agit pas d'une source de lumiere.

5.1.1.3 Calcul theorique du form factor

5.1.1.3.1 Approche intuitive La definition du form factor est delicate. En effet, celui-ci definit l'influence d'un patche sur un autre et ne depend que de la position relative des deux patches, cela doit donc etre un terme purement geometrique.

Intuitivement, on peut dresser une liste des parametres duquel il doit dependre :

- L'aire des patches : une grande surface aura bien entendu beaucoup plus d'influences qu'une petite
- La visibilite d'un patche par rapport a un autre : si un objet de la scene empeche un patch i d'etre visible d'un patche j, le form factor sera nulle, puisque leur influence mutuelle sera nulle.

- On doit toujours avoir $F_{i,i} = 0$ puisqu'un patche n'a aucune influence sur lui meme.
- On doit avoir $F_{i,j} = F_{j,i}$ puisque l'influence sur d'un patche i sur un patche j est la meme que celle du patche j sur le patch i.

5.1.1.3.2 Definition du form factor par Cohen et Greenberg En se basant sur le modele physique, Cohen et Greeberg proposere une definition du form factor qui est aujourd'hui la plus utilisee dans le modele de radiosite simple. C'est celle que nous avons choisit dans notre implementation.

$$F_{i,j} = \int_{A_i} \int_{A_j} \frac{(cos(\phi_1).cos(\phi_2))}{\pi.r^2}.H_{i,j}.dA_i.dA_j$$

où :
A_i = aire du patche de depart A_1
A_j = aire du patche d'arrivee A_2
r = distance entre les patches A_1et A_2
$H_{i,j}$ = 1 si A_1est visible de A_2, 0 sinon
ϕ_1 et ϕ_2 representent respectivement les angles entre les normales aux patches i et k et la droite qui les relies, en considerant ces deux surfaces comme planes, ce qui est necessairement le cas puisque l'on travaille avec des patches.

Puisque nous nous sommes ramene au cas ou la radiosite d'un patche est constante, c'est a dire au cas ou le form factor est constant pour chaque point du patche, on peut se ramener a l'equation :

$$F_{i,j} = \frac{(cos(\phi_1).cos(\phi_2))}{\pi.r^2}.H_{i,j}.A_i.A_j$$

L'un des fonctions importantes de ce form factor est la fonction $H_{i,j}$, la fonction de visibilite. En effet, elle peut etre particulierement couteuse en temps de calcul : son calcul consiste a tester si la droite *(i,j)* coupe l'un des objets de la scene, ce qui impliquerait selon une approche naive de faire autant de tests que d'objets dans la scene, et ce pour chaque patche !.

Cependant, on peut remarquer que ce calcul revient a :

1. Calculer l'equation du rayon joignant les patches i et j
2. Lancer ce rayon
3. Tester si l'objet coupe est le patche j, si oui alors renvoyer un, renvoyer 0 sinon.

On se rend compte que la partie difficile de ce calcul, l'etape 2 est en fait un probleme que nous avons deja traite lors du raytracing. On peut donc utiliser les fonctions codee pour le ray-tracing, en utilisant donc les nombreuses optimisations possibles, et plus particulierement en utilisant l'octree.

Resolution de l'equation de radiosite.

5.1.1.4 Resolution de l'equation de radiosite

5.1.1.4.1 Resolution theorique En principe pour tenir parfaitement compte des interactions entre les differents patchs, en considerant que tout les patchs ont exactement la mme aire, il faudrait resoudre un systeme lineaire representable par une matrice n carre avec n le nombre de patchs, notees K cette matrice est principalement constituees des form-factor entre les patch correspondand a la i-eme ligne et a la j-eme colonne de la matrice multiplie par la reflectance diffuse, ou plus simplement la lumiere renvoye par chaque patch. Cela decoule de l'equation simplifiee de la radiosite (en tenant compte de l'egalite d'aire entre les differents patch) :

$$Bi = E_i + p_i . \sum_{k=1}^{n} B_k . F_{k,i}$$

Cela amene donc a une egalite generale pour tout les patchs , sous forme matricielle :

$$(I - diag(P_1, ..., P_n).F).B = E$$

5.1.1.4.2 Resolutions pratiques

Introduction La simple generation de la matrice parait deja tres lourde en temps de calcul, et en besoin memoire pour des scenes de plusieurs milliers de patch (ce qui est necessaire si on veut obtenir une image realiste). Cette methode est evidement tres precise car elle resoud de maniere mathematique et sans aucune approximation. Toutefois Les demandes en temps de calcul de cette methode sont reellement immenses par rapport a ce l'on peut obtenir. Et bien souvent la difference ne pourra pas se voir a l'oeil nu avec des methodes plus approximative, mais qui prennent un temps de calcul nettement moins important.

Des methodes iteratives convergentes ont ete developpees, afin de permettre en un temps raisonnable le calcul de radiosite. Parmi les methodes les plus connues on peut cite l'*iteration de jacobi*, l'*iteration de Gauss-Seidel*, et l'*iteration de Southwell*. Les deux dernieres sont les plus utilisees car elle convergent plus rapidement vers la solution finale, que l'iteration de Jacobi. On regroupe parfois ces iterations sous le termes d'*algorithmes de radiosiosite progressive,* car si on affichait la scene a chaque passe, on verrait la scene s'illuminer progressivement de facon de plus en plus precise.

Iteration de Southwell Notre methode de calcul est basee sur l'iteration de southwell, qui repose sur un principe simple mais efficace. Afin de tendre rapidement vers la solution voulue, On distribue prioritairement le plus gros de l'energie contenu dans la scene. En effet la plus grosse quantite de lumiere est souvent contenu dans un nombre limite de patch, qui sont traites en priorite.

En pratique nous avons besoin d'une part d'un vecteur ou liste X, qui contiend la radiosite de chaque patch ; et d'un vecteur ou liste R, qui contient ce que l'on appelle le residus, c'est a dire l'energie du patch qui n'a pas encore ete distribue a la scene. EPSILON est un parametre ajustable, qui indique la precision demandee a l'algoritme, plus EPSILON est petit, plus le resultat sera precis, et plus le resultat mettra du temps a se calculer.

Algorithme :

selectionner le patch G avec le plus grand residus
pour chaque patch X differents de G de la scene
*radiosite patch X = (radiosite du patch G) * (K_ G,X)*
*residus patch X = (residus patch X) + (residus du patch G) * (K_ G,X)*
fin pour
residus patch G = 0
fin tant que

Ici K est la matrice precedement citee : K = (I - diag(p_1, ..., p_n) F) avec p la reflectance de chacun de patch, et F le facteur de forme entre chacun des patch.

Si on selectionne le patch avec le residus le plus grand, puis pour chacun des autres patch de la scene on distribue l'energie contenu dans le patch initial. Puis on met a jour les residus afin de tenir compte de la nouvelle distribution d'energie dans la scene. Les patchs qui ont le plus d'energie a distribuer seront ainsi trs vite traites, et des les premieres iterations, il sera possible d'avoir une idee trs precise du rendu final.

Il apparait clairement qu'avec une telle methode, on se rapproche trs rapidement du resultat correct, par rapport a une methode sans selection prealable des patchs a traiter, qui ne font pas la moindre difference entre les patchs et qui converge aussi vers le resultat correcte, mais qui necessitent beaucoup plus d'iteration pour arriver au resultat correct.

Il faut enfin noter que d'apres cet algorithme, la scene sera au debut noire, et qu'elle sera a chaque iteration un peu plus lumineuse.

5.1.2 Radiosite hierarchique

5.1.2.1 Presentation

Comme nous l'avons vu precedemment, la methode la plus intuitive pour illuminer une scene en employant l'algorithme de Southwell est de diviser chaque surface en un nombre fini de patches. Un maillage regulier de chaque surface apparait comme la methode la plus simple, mais les temps de calcul sont rapidement tres important. En effet, les elements doivent etre assez petits pour etre de radiosite constante ; or diviser par deux leur longueur multiplie leur nombre par quatre et le nombre de facteurs de forme par seize. Le calcul de la radiosite selon cette methode a une complexite de $O(n^2)$, avec n le nombre de patches.

Afin de reduire le nombre de calcul de form factor, Harahan et Al proposerent en 1991 la radiosite hierarchique [9], s'inspirant d'une analogie avec le probleme

astronomique des N corps. Cette methode exploite le fait que les interactions lumineuses decroissent comme l'inverse du carre de la distance si bien que deux patches contigus peuvent etre regroupes lorsqu'on etudie leur interaction avec un troisieme suffisament eloigne : on calcule ainsi un facteur de forme au lieu de deux. La mehtode developpee par Harahan est base sur un arbre quaternaire (un quadtree), ou chaque niveau n'interagit qu'avec des noeuds du meme niveau.

La radiosite hierarchique ne subdibvise pas a priori les surfaces : les surfaces ne sont subdivisee que lorsque cela apparait necessaire, c'est a dire lorsque le form factor entre la surface et une des autres surfaces de la scene est trop important (dans ce cas, on suspecte une forte interaction, et donc la radiosite a peu de chance d'etre constante, donc on subdivise).

La construction de l'arbre est toujour un probleme de complexite $O(n^2)$, mais avec n le nombre de surfaces de depart, et non plus le nombre de patche, ce qui est tres largement inferieur au $O(n^2)$. De plus Hanrahan a montre que le nombre de paire de patche etait de complexite $O(n)$.

5.1.2.2 Implementation

5.1.2.2.1 Construction du quadtree La partie la plus compliquee de la radiosite hirarchique est la construction du quadtree. Harahan propose une procedure recursive d'affinage :

```
Refine(Patch *p, Patch *q, float Feps, float Aeps){
  float Fpq, Fqp;
  Fpq = FormFactorEstimate(p,q);
  Fqp = FormFactorEstimate(q,P);
  if(Fpq< Feps && Fqp < Feps)
    link(p,q)
  else
    if(Fpq > Fqp){
      if(subdiv(q,Aeps)){
         Refine(p,q->ne,Feps,Aeps);
         Refine(p,q->nw,Feps,Aeps);
         Refine(p,q->se,Feps,Aeps);
         Refine(p,q->sw,Feps,Aeps);
      }else{
         link(p,q);
    }else{
       if(subdiv(p,Aeps)){
         Refine(q,p->ne,Feps,Aeps);
         Refine(q,p->nw,Feps,Aeps);
Refine(q,p->se,Feps,Aeps);
         Refine(q,p->sw,Feps,Aeps);
     } else {
```

```
            link(p,q);
        }
    }
```

La fonction FormFactorEstimate(p,q) calcule un form factor approche (selon une limite superieure), plus rapide a calculer qu'un "vrai" form factor. Nous avons choisit un calcul base sur la notion d'angle solide. Nous y reviendrons dans la section concernant le form factor.

La fonction subdiv(p,Aeps) divise un patch p en quadtre patches, formant ainsi le quadtree. Si l'air du patche p est inferieur a Aeps, elle renvoie faux.

Enfin, Refine prend en argument deux patches, une aire minimum pour arreter de subdiviser, et un form factor minimal, Feps. Refine commence par estimer le form factor entre deux patches, puis soit subdivise le patche et affine ses fils (en relancant une recursion) ou arrete la recursion en enregistrant une interaction entre les deux patches.

Nous avons rajoute un test de visibilite : si P n'est pas visible de Q, alors Fpq = Fqp = 0, car deux patches non visibles l'un de l'autre n'interagissent pas. Cependant, ce test ne doit pas etre base sur les centres des deux patches, en effet, les centres peuvent ne pas se "voir" l'un de l'autre alors que certains de leurs cotes sont visibles l'un de l'autre. En consequence, notre test de visibilite teste la visibilite entre chacun des quatres coins des quadtree (cela fait donc seize tests) et en deduit un pourcentage de visibilite. On multiplie ensuite l'estimation du form factor par ce pourcentage.

5.1.2.2.2 Approximation du form factor Nous avons choisit d'approximer le form factor en calculant l'angle solide entre un patche P, de taille finie et un patche Q, de taille infinimement petite :

Soit D la distance entre les centres de P et Q.

Soient $\overrightarrow{N_1}$la normale a P et $\overrightarrow{N_2}$ la normale a Q.

Soient $a = \overrightarrow{N_1}.\overrightarrow{PQ}$ et $b = \overrightarrow{N_2}.\overrightarrow{QP}$ (a est a donc pour valeur le cosinus de l'angle forme par $\overrightarrow{N}$ et $\overrightarrow{PQ}$, de meme pour b).

Soit A l'aire du patche P.

Soit vis(P,Q) la fonction renvoyant le pourcentage de visibilite de P et Q.

Alors, si F est l'approximation du form factor :

$$F = \frac{a.b.A}{D^2.\pi}.vis(P,Q)$$

5.1.2.2.3 Calcul analytique du Form factor Soit $\overline{R_i}$ le vecteur unitaire du point P vers le vecteur $\overrightarrow{V_i}$, ou $V_{i_{i\epsilon<1,4>}}$ represente les coins de nos quadtree.

vis(P,Q) represente toujours le pourcentage de visibilite de P a Q.

$$F_{P.dQ} = -\frac{1}{2}\sum_{i=1}^{n}(arcos(\overline{R_i} \bullet \overline{R_{i+1}})) \bullet N.\overline{(\overline{R_i} \wedge \overline{R_{i+1}})} * vis(P,Q)$$

5.1.2.2.4 Resolution de l'equation de radiosite Une fois l'abre construit et les liens etablis, il suffit de parcourir l'arbre en profondeur pour trouver le patch avec la plus grande radiosite. Puis il faut suivre les liens pour repartir son energie sur les patches sur lesquels il interagit. On calcule alors l'interaction entre les deux patches en utilisant le form factor analytique.

5.2 Nurbs

5.2.1 Introduction

Nous avons vu dans la partie concernant la radiosite hierarchique que nous avions besoin de diviser en *patches*, la division n'etant pas uniforme. Le format d'entree (.pov) decrit des objets formels, ainsi, une sphere est decrite par les coordonnees de son centre et par son rayon.

Or il est difficile de diviser une telle surface en preservant son exactitude mathematique. De plus, zRcube ne rend pas que des spheres, et il peut donc etre tres interessant de ne pas avoir a ecrire des routines de subdivisions differentes pour chaque type d'objet.

Enfin, la scene sera rendue via un algorithme de ray-tracing apres l'etape de radiosite, le ray-tracing travaillant lui aussi sur des objets formels, garder une definition mathematiquement exacte des objets durant la premiere etape apparait la encore souhaitable.

Nous avons donc ecarte la solution basee sur une simple polygonalisation des objets. En effet, tout d'abord cette polygonalisation n'est qu'une approximation de l'objet mathematique, et par consequent l'operation de polygonalisation n'est pas bijective : il serait dans ce cas tres difficile de passer d'un objet decrit sous forme de polygones a un objet formel pour le raytracing, et donc d'utiliser les calculs d'illumination lors de l'etape de ray-tracing.

Mais surtout, l'etape de subdivision hierarchique serait dans le cas de polygones particulierement difficile. Imaginons une sphere faite avec peu de polygones, supposons que l'on ai besoin de subdiviser la partie haute de la sphere. On serait amene a diviser un - ou des- polygones, c'est a dire une - ou des- surfaces planes en un ensemble de polygones formant une surface concave, une meilleure approximation de la sphere. En dehors du probleme de la realisation technique, surmontable, on imagine assez difficilement une sphere avec certaines surfaces tres mal approximees par un petit nombres de polygones et certaines bien approximee par un nombre important de ces derniers.

En raison de tous ces defauts de version polygonalisees de objets, nous nous sommes tournes vers une representation mathematique basee sur les surfaces de NURBS. Cette representation permet mathematiquement de representer n'importe quelle surface en utilisant un jeu de donnee tres reduit. En raison de la possibilite d'une definition parametrique des surfaces, cette representation permet une division facile et exacte.

Nous presenterons ici progressivement les notions necessaires a la comprehension et donc a l'implementation des surfaces de NURBS. Nous passerons sur

la plupart des demonstrations mathematiques afin de se concentrer sur les resultats utiles a l'implementation, celles-ci peuvent etre trouvees dans [4]. Notons enfin que les NURBS sont un puissant outil pour la modelisation, mais que nous ne nous attacherons ici qu'a decrire les proprietes utiles dans notre application specifique.

5.2.2 Preliminaires

5.2.2.1 Polynome de Berstein et modele de Bezier

5.2.2.1.1 Polynome de Berstein La defintion d'une courbe de Bezier repose sur le polynome de Berstein, polynomes utilises par ce dernier au debut du XXeme siecle pour l'approximation polynomiale des fonctions.

Definition : Soit n un entier non nul. Pour tout i variant de 0 a n ($i \epsilon$[[0,n]]), on definit le polynome de Berstein de degre n et d'indice i note B_n^i par la formule :

$$B_n^i(t) = C_n^i t^i (1-t)^{n-i}$$

ou C_n^i est le coefficient standard du binome
$\frac{n!}{i!(n-i!)}$ et ou $t \epsilon \Re$, la plupart du temps considere sur [0 ;1].

5.2.2.1.2 modele de Bezier On definit a l'aide de ce polynome l'equation d'une courbe de Bezier.

Definition Soit n un entier strictement positif donne, et dans dans $\Re^3$, une suite de $n+1$ points quelconques $P_0, P_1, ..., P_n$.La courbe de Bezier associee a ces donnees est l'arc decrit par le point M(t) defini par la formule

$$\sum_{i=0}^{i=n} B_i^n(t) \overrightarrow{OP_i}$$

Les points $P_0, P_1, ..., P_n$ sont appeles "points de definition" (ou points de controle). La figure 1 presente une courbe de bezier construite avec quatre points de defintions. Notons que la courbe passe par P_0 et P_3.

Les courbes de Bezier permettent de definir elegamment et simplement de nombreux types de courbes, elles sont d'ailleur utilisees dans la plupart des logiciels de dessin 2D. Cepedant, l'analyse de la definition precedente montre que le degre n de l'equation - et donc du polynome de Berstein - est egal au nombre de points de definition. Or, le calcul de la valeur de ce polynome en t augmente rapidement avec le degre, on se limite donc tres souvent a quatre points (comme c'est le cas sur la figure).

Or nous avons besoin de definir des surfaces - et donc des courbes - quelconques. On peut envisager de mettre plusieurs courbes de Bezier bout a bout,

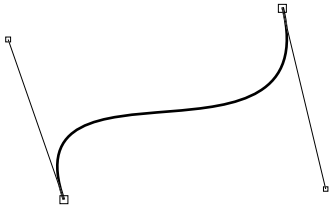


FIG. 5.1 – Un exemple de courbe de Bezier

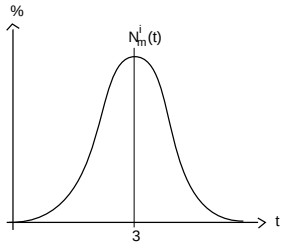


FIG. 5.2 – Un exmple de fonction de base $N_i^m(t)$

mais cette solution nuit a la generalite (rappelons que l'on souhaite une definition unique de nos surfaces). De plus, les courbes de Bezier ne permettent pas de decrire toutes les courbes.

Afin de repondre a ces exigences, on est amene a generaliser les courbes de Bezier, et a definir des courbes que l'ont a appele "B-Splines".

5.2.2.2 Courbes B-Spline

L'idee directrice est de batir une famille de fonction avec de nombreux parametres permettant un choix plus vaste de courbes que le permet modele de Bezier. Le modele des B-splines inclue les fonctions de Bezier, mais permet de definir aussi d'autres courbes, par exemple ne passant par aucun point de definition.

5.2.2.2.1 Fonctions B-splines (*Basis Functions*) La fonction $N_m^i(t)$ est appele la fonction de base, ou fonction B-Splines. Le "B" de "B-spline" signifiant "Base". Cette fonction determine a quel point un point de controle P_i influence la courbe pour une valeur t. La valeur de cette fonction est un nombre reel, par exemple 0.5, ainsi un point particuler peut etre, par exemple, defini par 25%de la position d'un point de controle, plus 50% d'un autre et 15% d'une dernier. L'equation d'une courbe B-spline doit donc specifier la fonction de base pour chaque point de controle.

La figure 5.2 montre un exemple de ce a quoi ressemble une fonction de base : elle a une valeur maximale a un certain point defini -le point de controle

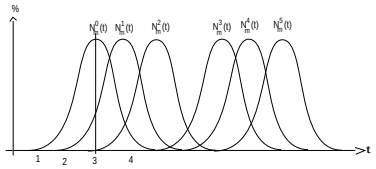


FIG. 5.3 – Un ensemble de fonction de base

- et decroit au fur et a mesure que l'on s'eloigne de ce point.

Puisque chaque point de controle a sa propre fonction de base, une courbe B-spline avec , par exemple, cinq points de controle aura cinq fonctions de base, chacun couvrant une partie de l'intervalle de variation de t. A t=2.3 sur la figure 5.3, le point de controle P_0 a une influence d'environ 20%, tandis que P_1 en a une d'environ 70% et P_2 environ 5%. Lorsque t parcours l'intervalle [0,7], chaque point de controle influence progressivement la courbe : plus t est proche du maximum de la fonction de base correspondant au point de controle, plus la courbe sera influencee par ce point.

Vecteur nodal (*knot vector*) Notons que sur la figure 5.3, chaque courbe a exactement la meme forme et est non-nulle sur un intervalle de meme taille. Dans un soucis de generalisation, on cherche a faire varier la largeur des intervalles et la hauteur maximale des courbes.

La solution est de definir une suite de reels qui coupe l'intervalle de variation de t en plusieurs sous-intervalles. En variant la taille relative des intervalles, l'intervalle d'influence de chaque point variera.

Cette suite de reels est appelee "*vecteur nodal" (knot vector),* les reels sont appeles *valeurs nodales.*

De par son utilite, cette suite doit etre non decroissante.

Lorsque les valeurs nodales non repetees sont egalement espacees, le vecteur nodal est dit "uniforme", il est dit "non uniforme" dans le cas contraire.

Lorsque des valeurs sont repetees, on parle de *noeud multiple.*

Fonction de base On peut maintenant definir la fonction de base. On utilise la definition recursive proposee par Cox et De Boor.On se fixe un vecteur nodal v, k le nombre de valeures nodales, soit m le degre de la fonction (l'indice i verifiant $0 \leq i \leq k-m-1$, ce qui impose $m \leq k-1$).

$$N_0^i = \left\{ \begin{array}{c} 1, \, si \, v_i \leq t \leq v_{i+1} \\ 0, \, sinon \end{array} \right)$$

$$N_m^i = \frac{t - v_i}{v_{i+m} - v_i} N_{m-1}^i(t) + \frac{v_{i+m+1} - t}{v_{i+m+1} - v_{i+1}} N_{m-1}^{i+1}$$

On fixe de plus la convention : si l'un des denominateur s'annule (c'est le cas lors de noeuds multiples), alors on annule le terme correspondant.

Cas des valeurs nodales multiples Soit le vecteur nodal (0,0,1,1), on trouve facilement :

Au degre 0, seule N_0^1 est non nulle.

Au degre 1, $N_1^0(t) = 1 - t$ sur [0,1[, nulle ailleur

et $N_1^1(t) = t$ sur [0,1[, nulle ailleur.

On reconnait dans ces fonction les polynomes de Berstein :

$N_1^0 = B_1^0$ et $N_1^1 = B_1^1$ sur le meme segment [0,1[.

Ce resultat se generalise, mais nous ne ferons pas la demonstration ici.

5.2.2.2.2 Courbes B-splines D'une maniere analogue au modele de Bezier, on pondere des vecteurs construit avec des points de controle par des fonctions B-splines, qui remplacent le polynome de Berstein du modele de Bezier.

Notations Le degre de la courbe est note m, c'est le degre commun des fonctions splines utilisees.

Le nombre des B-splines intervenant est note n+1, n un entier naturel. Cela correspondra aussi au nombre des arcs de la courbe (que l'on appelle aussi *segments de courbe*), ce qui signifie que l'intervalle dans lequel varie le parametre doit etre la reunion de n+1 intervalles non vides $[v_j, v_{j+1}]$.

Le vecteur nodal forme de k reels $[v_0, v_1, ..., v_k]$ donne l'intervalle $[v_m, v_{k-m}]$ de variation du parametre. Le nombre de valeurs nodales distinctes contenues dans cet intervalle est le nombre de segments, donc n+1.

Dans le cas ou toutes ces valeurs nodales sont simples, on a $k-m-m = n+1$ d'ou $k = 2m+n+1$. Le premier intervalle correspond a $[v_m, v_{m+1}]$ et le n+1-ieme a $[v_{m+n}, v_{m+n+1}]$.

Les points de controle sont notes P_i, ou i prend les valeurs entieres de 0 a $m + n$.

Definition Soit un entier m choisi pour le degre de la courbe (dans notre cas, on choisira generalement 2 ou 3), et un vecteur nodal $(v_0, v_1, ..., v_k)$ de k+1 valeurs nodales pour lesquels on suppose que $k > 2m$ et $v_{k-m} > v_m$. Une suite (P_i) de m+n+1 points de l'espace, ou n+1 est le nombre de valeurs nodales distinctes contenues dans $[v_m, v_{k-m}]$. La courbe B-spline de degre m est l'ensemble des points M de l'espaces definis par la formule :

$$\overrightarrow{OM}(t) = \sum_{i=0}^{i=n+m} N_m^i(t).\overrightarrow{OP_i}$$

Le parametre reel t doit parcourir l'intervalle $[v_m, v_{k-m}]$.

Sur $[v_m, v_{k-m}]$, la somme des fonctions N_m^i est egale a 1. De plus, on demontre aisement grace a cette propriete que la courbe est independante par rapport au repere.

La figure 5.4 presente un exemple de courbe B-spline.

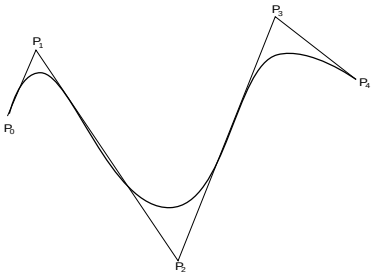


FIG. 5.4 – Un exemple de courbe B-spline

5.2.2.2.3 Utilisation : choix du vecteur nodal et du degre Pour generer une courbe "lisse" du type de celle de la figure ? ?, on utilisera generalement un vecteur nodal uniforme.

Un vecteur nodal du type (0,0,1,1) permet de tracer une courbe de Bezier (car comme nous l'avons vu, dans ce cas, N=B).

Pour que la courbe passe par les extremites, il suffit de doubler les valeures nodales des extremites, par exemple : (0,0,1,2,3,3).

Pour des cas courant, on ne depassera que rarement le degre 3.

5.2.3 Nurbs (Non Uniform Rationnal B-Splines)

Le modele des B-splines rationnelles non uniforme a notamment ete propose par Coons (1967) et Versprille (1975). Dans ce modele, les fonctions rationnelles remplacent les fonctions polynomiales, c'est donc une generalisation des B-splines, ces dernieres etant incluses dans les NURBS.

En plus des caracteristiques des B-splines, les NURBS apportent deux principales nouvelles proprietes : l'invariance par les transformations projective -tres utile pour la modelisation interactive, mais sur laquelle nous ne nous attarderons pas - et surtout la modelisation des coniques autre que la parabole.

Nous allons donc enfin pouvoir modeliser toutes les formes.

5.2.3.1 Courbes Nurbs

5.2.3.1.1 Definition On garde les notations employees lors de la definition des B-splines, et ajoute une suite de reels $p_0, p_1, ..., p_{m+n}$ tous non nuls, le plus souvent positifs.

On definit alors une courbe NURBS comme l'ensemble des points $M(t)$ de l'espace definis par :

$$\overrightarrow{OM}(t) = \sum_{0}^{m+n} F_m^i(t).\overrightarrow{OP}_i$$

avec

$$F_m^i(t) = \frac{N_m^i(t).p_i}{\sum_{j=0}^{j=n} N_m^i.p_i}$$

ou $t\epsilon[v_m, v_{k-m}[$

Les reels $p_0, p_1, ..., p_{m+n}$ sont appeles poids, il permettent de modifier l'influence du point correspondant sur la courbe

5.2.3.1.2 Modelisations de sections coniques On appelle section conique la courbe correspondant a l'intersection d'un cone avec un plan. L'angle selon lequel le plan coupe le cone determine si le resultat est un cercle, une ellipse, une parabole, ou une hyperbole.

Dans une definition stricte ces deux dernieres courbes sont infinies, cependant on se restreindra aux arcs coniques (c'est a dire des morceaux de coniques), les courbes infinies etant en general peu utile dans la modelisation d'objets.

Puisque les courbes coniques sont quadratiques, nous pouvons les representer par des NURBS quadratiques, c'est a dire des NURBS de degre deux. Afin de determiner les parametres necessaires a la definition de cette NURBS, on peut employer la methode suivante (que l'on admettra, la preuve sortant du cadre de ce rapport) :

- La courbe est definie par trois points de controle. Le premier et le dernier points sont le point de depart et le point d'arrivee de l'arc, la position du troisieme point determinant la forme de la courbe.
- Les poids des premiers et derniers points sont 1.0
- Pour le point de controle central : un poids inferieur a 1.0 genere une ellipse, un poids egal a 1.0 genere une parabole et un poids superieur a 1.0 genere une hyperbole.
- Le vecteur nodal correspondant est {0.0, 0.0, 0.0, 1.0, 1.0, 1.0}

L'arc conique que l'on utilise le plus souvent est probablement l'arc de cercle. Puisqu'un cercle est une ellipse particuliere, la methode est un cas particulier de la methode precedente.

- Le triangle forme par les points de controle doit etre isocele
- Soit $2.\theta$ la mesure de l'arc, le poids du point de controle central est egal a $cos(\theta)$

5.2.3.1.3 Exemple : modelisation d'un cercle On utilise une NURBS de degre 2, en definissant les points de controles (les points sont definis en coordonnees homogenes : x,y, poids), le triangle P_1, P_3, P_5 etant isocele :

$P_0 = \{1, \frac{\sqrt{(3)}}{2}, 1\}$	$P_4 = \{3, \frac{\sqrt{(3)}}{2}, 1\}$
$P_1 = \{0, 0, 0.5\}$	$P_5 = \{0, \sqrt{(3)}, 0.5\}$
$P_2 = \{1, 0, 1\}$	$P_6 = \{1, \frac{\sqrt{(3)}}{2}, 1\}$
$P_3 = \{2, 0, 0.5\}$	

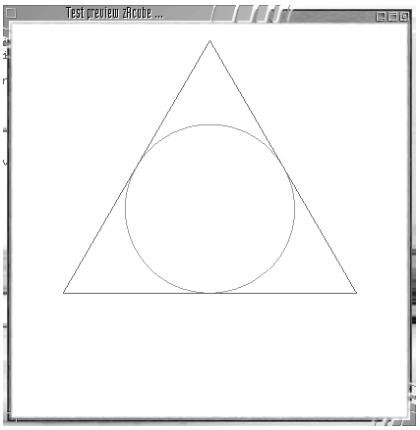


FIG. 5.5 – Un cercle defini par une courbe NURBS de degre 2 et ses points de controle

et un vecteur nodal du type $v = \left\{0, 0, 0, \frac{1}{3}, \frac{1}{3}, \frac{2}{3}, \frac{2}{3}, 1, 1, 1\right\}$

On peut de la meme maniere definir un cercle a l'interieur d'un carre (on a dans ce cas quatres arcs a la place de trois et neuf points a la place de six), d'un pentagone, etc...

5.2.3.2 Surfaces de Nurbs

5.2.3.2.1 Definition

Produit tensoriel La technique du produit tensoriel permet de construire une surface en "multipliant" deux courbes. Dans notre cas, nous voulons creer une surface a partir de courbes NURBS, cette methode revient donc a multiplier la fonction de base de la premiere courbe avec celle de la deuxieme, puis a utiliser le resultat de ce produit comme fonction de base pour une matrice de point de controle. Les surfaces ainsi generees (le principe s'applique a d'autres courbes) sont appellees *surfaces par produit tensoriel.*

Definition On peut donc definir une surfaces de NURBS, en conservant les notations precedemment utilisees pour les courbes NURBS :

$$\overrightarrow{S}(u,v) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{p} N_m^i(u).N_m^j(v).\overrightarrow{P_{i,j}}.p_{i,j}}{\sum_{i=0}^{n}\sum_{j=0}^{p} N_m^i(u).N_m^j(v).p_{i,j}}$$

Notons que l'on a donc besoin d'une matrice $M_{n,p}$ de points de controles et donc d'une matrice $M_{p_{n,p}}$ de poids associes.

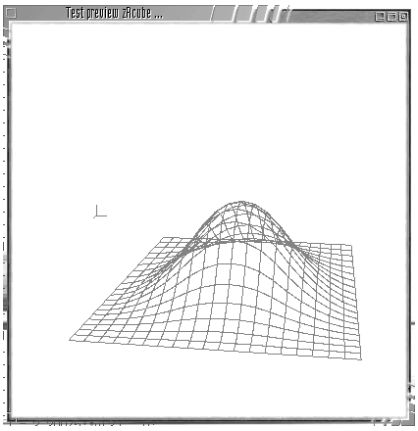


FIG. 5.6 – Une surface de NURBS simple

5.2.3.2.2 Exemple La figure 5.6 montre une surface de nurbs simple de degre 2 generee par la matrice (en coordonnees homogenes) :

$$M = \begin{pmatrix} (0,0,0,1) & (1,0,0,1) & (2,0,0,1) \\ (0,1,0,1) & (1,1,2,1) & (2,1,0,1) \\ (0,2,0,1) & (1,2,0,1) & (2,2,0,1) \end{pmatrix}$$

avec un vecteur nodal uniforme, par exemple $v = \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6\}$

On voit rapidement que l'ensemble des vecteurs de la matrice sauf celui du centre sont dans le meme plan puisque la coordonnee z est nulle. Ces points doivent donc former un rectangle. Le vecteur central a quant a lui une coordonnee z=2, on obtient donc la figure :

5.2.4 Implementation et utilisation

5.2.4.1 Implementation

Armes de ces notions mathematiques, nous avons pu commencer a tracer des nurbs, d'abord en deux dimensions, afin de valider les fonctions de base, en dessinant par exemple le cercle de la figure 5.5 puis en trois dimensions.

5.2.4.1.1 Architecture adoptee Le diagramme UML de la figure 5.7 presente l'architecture des classes que nous avons adoptee pour implementer les nurbs, les surfaces de nurbs et les objets en nurbs.

Nous avons d'abord cree la classe *Nurbs*, dont on derive une classe *Uniform-Nurbs*, dont la seule difference est qu'un vecteur nodal de type uniforme est automatiquement genere. Cette classe permet de definir une courbe NURBS en deux dimension a partir d'une liste de point de controle, d'un vecteur nodal et

d'un degre. Cette classe possede une methode *evaluate(float t)* qui renvoie le vecteur correspondant au parametre t, ce qui est tres utile pour par exemple utiliser les valeurs dans une autre classe.

Nous avons ensuite cree la classe *Surface*, qui permet naturellement de definir une surface NURBS a l'aide d'une matrice de point de controle, d'une matrice de poids, d'un vecteur nodal et d'un degre. Cette classe possede elle aussi une methode *evaluate(float u, float v)* qui renvoie le vecteur correspondant aux parametres *u,v*.

Ces deux classes possedent aussi une methode *display()*, compilee seulement si openGL est detecte sur la machine, qui permet d'afficher la surface/courbe dans le previewer openGL. Cette methode est principalement implementee afin de permettre le debugging : un dessin de la surface est bien sur bien plus clair pour valider notre code qu'une suite de chiffre.

A l'aide de cette classe *Surface,* nous avons defini la famille d'objet *Curve**. L'objet pere est curveObject, qui contient un pointeur vers un tableau de *Surfaces* (il est ainsi possible de definir un objet a partir de plusieurs surfaces). On derive alors de cet objet les objets *POV* de base et l'on cree ainsi les classe *CurveSphere,CurveCone* et *CurveTriangle.* Ces objets derivent naturellement aussi des objets *Sphere,Cone* et *Triangle* qui correspondent aux objets formels (ceux utilises par le ray-tracing) afin d'heriter des membres de base (par exemple pour la sphere : le rayon et le center).

Nous reviendrons plus tard sur les classe de *Quadtree.*

5.2.4.1.2 Difficultes recontrees

Nurbs en deux dimensions La plupart des difficultes que nous avons recontrees sont liees a une mauvaise comprehension des differents parametres .

Ainsi, si comprendre les points de controle ne nous a pas pose de difficultes, le vecteur nodal est longtemps reste mysterieux pour nous, la plupart de nos documentation posant la formule sans tenter de l'expliquer.

L'intervalle de variation du parametre *t* (*le domaine parametrique),* nous a pose beacoup de problemes aussi : beaucoup d'articles que nous avons lu se contredisaient a ce sujet, et les tests sur des cas particulier n'etaient pas toujours revelateurs des bugs.

Nurbs en trois dimensions Une fois que les NURBS en deux dimensions furent maitrisees, nous nous sommes attaques aux NURBS en trois dimension, la difficulte principale fut ici de comprendre comment devait etre organise la matrice de point de controle pour decrire la surface voulue (notamment une sphere). Notre solution sera presentee dans la section suivante.

5.2.4.1.3 Methode adoptee pour la modelisation des surfaces

Apres de nombreux tests et essais, nous sommes arrives a la conclusion que l'on pouvait voir chaque ligne de la matrice comme un ensemble de point de controle

FIG. 5.7 – Diagramme UML de l'implementation des NURBS et des surfaces de NURBS dans zRcube

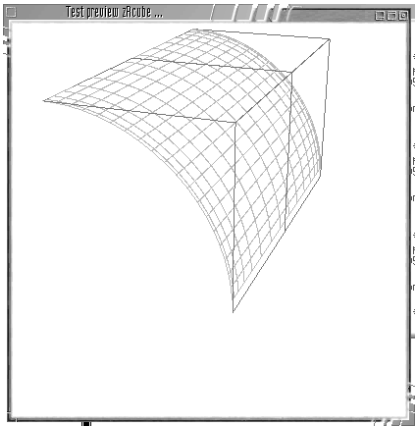


FIG. 5.8 – Un morceau de cylindre et ses points de controle

decrivant une "tranche de la surface". En mettant bout-a bout des "tranches" de ce type, on pouvait facilement decrire une surface.

La figure 5.8 montre un morceau de cylindre et ses points de controles :

Dans ce cas, la matrice est composee de neufs points. La premiere ligne correspond aux trois points les plus a gauche, la seconde aux trois en haut a droite et la derniere les trois en bas a droite. La courbure est obtenue en modifiant les poids des points de la second ligne. On a donc la matrice de points de controle (en coordonnnees homogenes) :

$$\begin{pmatrix} (0,1,1,1) & (0,1,0,1) & (0,1,-1,1) \\ (1,1,1,\frac{\sqrt{(2)}}{2}) & (1,1,0,\frac{\sqrt{(2)}}{2}) & (1,1,-1,\frac{\sqrt{(2)}}{2}) \\ (1,0,1,1) & (1,0,0,1) & (1,0,-1,1) \end{pmatrix}$$

avec un vecteur nodal : $\{0,0,0,1,1,1\}$

On pourra encore une fois noter la compacite de la description !

5.2.4.1.4 Definition d'une sphere en NURBS Cette definition de surface semble assez intuitive, mais nous devons etre capable de transformer tous les objets de bases en surfaces NURBS, et nous savons que c'est mathematiquement possible. Nous devions donc etre capable de decrire une sphere en tant que surface de NURBS.

Apres de nombreux essais, nous avons abouti a une definition basee sur la rotation de points de controle definissant un cercle (ce n'est probablement pas la seule maniere de definir une sphere avec des courbes NURBS) :

– Soit une matrice M a n colonnes, et m lignes. On utilisera des NURBS de degre 2.

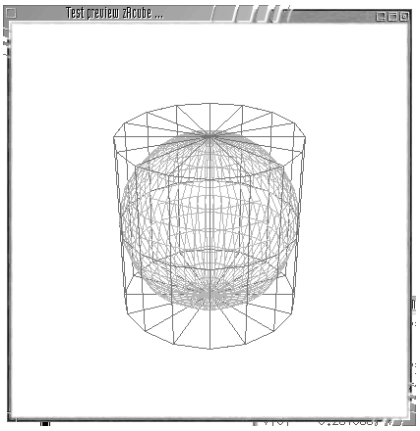


FIG. 5.9 – Une sphere definie par une surface de NURBS et ses points de controle

- On commence par placer les points de controle definissant un cercle (voir par exemple la figure ? ?) accompagnes des poids correspondants. On utilise une version definissant un cercle a base de n points de controles. Ces points definiront la premiere ligne de la matrice.
- On effectue une rotation de ces points de $\frac{2.\Pi}{m}$ afin de definir n nouveaux points de controles qui constitueront la ligne suivante dans la matrice.
- On recommence la derniere operation m fois pour obtenir les m lignes de la matrice.

La sphere de la figure 5.9 a ete cree a partir d'une matrice de 9 lignes et 9 colonnes (soit 81 points). La matrice minimum est obtenue en utilisant une versions a six points pour defininir un cercle, cette matrice aura donc pour taille 6x6, soit 36 points.

Mais cette matrice n'est pas forcement la meilleure pour toutes les utilisations. En effet, elle revient a inscrire une sphere dans un pyramide, ce qui n'est pas tres intuitif, notamment si l'utilisateur peut etre amene a "tirer" certain point de la sphere pour la transformer. On preferera probablement dans ce cas un representation a base de carres ou d'octogones.

5.3 Utilisation des nurbs pour la radiosite

Utilisation pour la radiosite progressive hierarchique

Nous avons vu precedemment que la raison pour laquelle nous avons implemente les surfaces NURBS etait que nous voulions sudbviser de maniere generale des objets formels tout en gardant une definition mathematiquement exacte de ses objets afin de creer un *mesh adaptatif.*

5.3.1 Definition du quadtree

Reprenons la definition d'une surface de Nurbs :

$$\overrightarrow{S}(u,v) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{p} N_m^i(u).N_m^j(v).\overrightarrow{P_{i,j}}.p_{i,j}}{\sum_{i=0}^{n}\sum_{j=0}^{p} N_m^i(u).N_m^j(v).p_{i,j}}$$

Puisque u et v varient entre respectivement dans des intervalles I_1 et I_2 (definis par leurs vecteurs nodaux respectifs), si on les fait varier dans un intervalle plus petit, on ne parcourera que partiellement la surface. Ainsi, si par exemple, on divise I_1en deux intervalles distincts, on obtiendra deux morceaux de surfaces, ce qui revient a "couper" la surface en deux.

Pour definir un nouveau noeud du quadtree, il suffit donc de couper les intervalles I_1et I_2 en deux morceaux chacun, on definit ainsi quatre sous-intervalles pour (u,v) qui constituent les bornes des fils.

Il est tres important de noter que la definition des "sous-surfaces" est toujours mathematiquement exacte, et ce bien que lors de l'affichage on approxime souvent ces surfaces par des carres. En consequence, on peut subdiviser a l'infini sans perdre de precision et sans avoir a revenir a la definition de l'objet (c'est a dire que l'on a pas, comme dans un modele polygonal, a transformer une surface plane en plusieurs sous surfaces planes en se ramenant a la definition de l'objet).

Une application simple de cette methode est visible sur toutes les copies d'ecran de surfaces de NURBS : pour tracer le maillage, on prend juste des valeurs particulieres pour (u,v), quatre valeurs definissent un carreau, ce qui revient a la construction d'un mesh tres simple.

Notons enfin que dans notre application, le mesh genere n'est absolument pas uniforme comme sur les figures presentees, les surfaces n'etant subdivisee que lorsque cela est necessaire lors du precalcul du mesh en appliquant la methode de la radiosite hierarchique.

5.3.2 Lissage de Gouraud

Le probleme Puisque nous ne pouvons avoir des elements infiniments petits, le rendu effectue avec la methode decrite precedemment donnerait des petits carres de couleurs. Nous voulions des degrades de couleur pour donner un aspect lisse aux couleurs, c'est a dire eliminer les discontinuites de couleurs.

Pour effectuer un lissage de Gouraud, nous avons besoin de connaitre les couleurs aux quatre coins des quadtrees. Ce n'est pas une operation si aisee que cela peut paraitre : s'il est facile de connaitre la couleur des freres d'un noeud, il peut etre necessaire de remonter jusqu'a la racine de l'arbre si l'un des quadtree voisin n'est pas dans la meme branche que celui que l'on cherche a lisser.

Notre solution Nous avons finalement trouve une solution assez simple basee sur un precalcul des couleurs de coins.

Pour cela, nous avons derivee la classe Vector pour creer une classe Vector-Rad. C'est un vecteur standard auquel on rajoute une propriete de couleur et

un entier N. On definit les coins des quadtree a partir de VectorRad a la place de Vector.

On prend de plus garde lors de la construction des quadtrees a ne pas avoir de vertices en doubles : si sur une surface deux quadtree partagent des vertices, ils doivent utiliser le meme VectorRad. Pour cela, chaque surface possede une liste de vertices : lorsque, lors d'une division, on est amene a declarer un nouveau vertex, on commence par verifier qu'il n'est pas dans la liste, s' il n'y est pas, on l'ajoute. Dans tous les cas, le quadtree stocke l'indice de ses coins dans la liste et non pas directement les vertices correspondants.

On realise ensuite un parcours en profondeur a partir de la racine de l'arbre correspondant a chaque surface. Pour chaque feuille, on ajoute a chacun des quatres coins du noeud la couleur du patche, et on incrememte N. Puisqu'on a pris garde a ne pas avoir de vertices en double, un meme VectorRad peut avoir ete "visite" entre une et quatres fois. Il suffit de diviser par N pour obtenir la moyenne des couleurs de ses voisins.

Grace a cette methode, on reussit donc a transformer un probleme global -connaitre la couleur de ses voisins - en un probleme local -connaitre la couleur de ses coins -.

Le lissage de Gouraud est ensuite une interpolation de couleur entre les coins. Dans le previewer openGL, elle est realisee directement par openGL, nous reviendrons dans la derniere partie sur la maniere de realiser cette interpolation dans le cadre du raytracing.

5.3.3 Implementation

Revenons au diagramme UML de la figure 5.7.

Classe Quadtree Puisqu'un quadtree est une "sous-surface", c'est donc une surface particuliere. Il apparait donc logique de declarer la classe Quadtree comme une classe derivee de la classe Surface, et on ajoute quatres parametres determinant les intervalles de variation de u et v.

Classe d'objets Puisque les objets sont des surfaces, et sont "decoupees" sous formes de quadtree, la surface de l'objet, celle pour laquelle les parametres u et v permette de parcourir la totalite de la surface , correspond a la racine d'un quadtree. On definit donc les classes d'objet en version Nurbs (CurvedSphere, CurveCone ...) en tant qu'un tableau de pointeur sur des Quadtree.

Etapes : On peut donc resumer les differentes etapes menant au calcul de la radiosite :

1. Conversion des objets en surface de NURBS
2. Construction de mesh et des liens a l'aide de la procedure d'affinage de Harahan
3. Iteration de Southwell pour determiner les couleurs des patches

Chapitre 6

Combinaison des differents elements

6.1 Parallelisation du Raytracing

6.1.1 Principe

La parallelisation du calcul de lancer de rayon a ete facilite grace a l'utilisation de la librairie reseau precedement developpee. Le probleme sur la methode de transmission etait donc resolu. Ne restait plus qu'a choisir une methode de repartition de calcul. Le projet a ete fonde avec l'hypothese que les machines qui se repartiraient le calcul seraient de puissances equivalentes. La repartition est donc faite de maniere equivalente par le serveur pour chaque machine cliente.

Pour selectionner les pixels a rendre sur les clients, le sereur attribut a chacun un nombre qui va servir de modulo. Dans la situation ou il y a trois clients, le premier client recoit le nombre 1 et 3 ce qui lui indique qu'il doit effectuer les calculs pour 1 pixel sur 3. Le second client va lui aussi rendre un pixel sur trois mais en commencant par le second et le troisieme client en commencant par le troisieme. A la fin des calculs, chaque client aura effectue le meme nombre de calcul a 1 pixel pres au maximum. Bien entendu, dans l'hypothese ou le groupe d'ordinateur qui se reparti les calculs n'est pas homogene, le temps de calcul sera augmente et les clients les plus rapides devront attendre les plus lents.

6.1.2 Application

Pour limiter la perte de temps en transferts de donnees sur le reseau, les paquets doivent etre de taille optimum et doivent contenir le maximum d'informations. Le serveur doit quant a lui etre de puissance non negligeable afin de pouvoir traiter toutes les donnees qu'il recoit via le reseau. Afin de realiser nos calculs en reseau, nous avons realises deux nouveaux programmes derives fortement du programme principal de calcul.

L'un est une version serveur qui va ouvrir un port afin de permettre aux clients de se connecter et de dialoguer et l'autre est la version cliente. Chacun de ces deux programmes prennent les options en lignes de commande ce qui permet de les lancer en console. Il y a deux methodes pour lancer un calcul parallele avec zrcube : la premiere consiste a creer un serveur sur une machine et a connecter les clients a la main sur ce serveur. Cette methode est fastidieuse des que l'on depasse 5 machines. La seconde methode consiste a utiliser l'interface principale qui permet de lancer les programmes clients sur les clients via ssh. Cette methode fonctionne a condition d'avoir des acces ssh sur toutes les machines.

6.1.3 Resultats

Les resultats obtenus en parallelisant les calculs du raytracing sont dependants de la scene rendue. Dans le cas ou l'on souhaiterait profiter pleinement du partage de calcul, il faut que le temps de calcul de chaque pixel soit au minimum deux fois superieur a son temps de transfert. Il devient donc evident que le meilleurs resultats se ressentiront sur des scenes tres complexes en hautes resolutions.

Type de Machine	cubes2.pov	complex.pov
monoposte celeron 360Mhz	172 s	56 s
monoposte celeron 500Mhz	120 s	40 s
2 clients (celerons 500Mhz)	147 s	27 s
3 clients (celeron 500Mhz)	85 s	43 s
monoposte bi-celeron 500Mhz	66 s	22 s

6.2 Combiner raytracing et radiosite

6.2.1 Selection du patch de radiosite pour le raytracing

Le but principal du projet etant de combiner deux algorithmes tres performants pour l'image de synthese. Il fallait que l'information d'illumination globale fourni par la radiosite puisse etre recuperee par le module de raytracing de zrcube. Pour cela il fallait pouvoir faire le rapport entre les resultats d'intersection avec les surfaces raytracees, et les patchs de la radiosité.

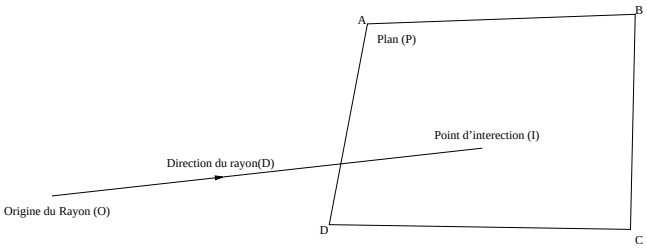
D'abors le raytracing determine pour chaque pixel l'objet qui doit etre affiche, puis quand l'objet en question est determine il demande au module de radiosite l'intensite lumineuse correspondante. Pour le determiner précisement le raytracing passe le rayon qui a ete utilise pour determiner l'intersection, le point d'intersection precisement calcule, et un pointeur sur l'objet en question.

La premiere etape consiste a savoir quel patchs correspondent a l'objet en question afin de ne pas tester la totalité des patchs de la scéne, c'est une optimisation relativement simple qui permet de gagner enormement sur les tests, mais qui necessite de conserver un pointeur sur l'objet d'origine dans chacun des quadtrees de la radiosite. Ainsi pour chacun des Quadtrees de l'objet, un

test d'intersection par rapport au plan est realise afin de determiner le point d'intersection entre le plan du patch et le rayon. Durant ce test d'intersection plusieurs cas triviaux sont traites, afin de rejeter le plus tôt possible les patchs qui ne conviennent pas. Puis une fois que le point d'intersection a ete calcule, on verifie que le point trouve se situe de la zone delimitee par les 4 cotes du patch, cela se fait a l'aide de produit scalaire avec les cotes du patch.

6.2.1.1 Test d'intersection entre un plan et une droite :

Au debut nous avons un plan qui est defini par les 4 points du patch. Il faut d'abors retrouver son équation afin de pouvoir le traiter de facon analytique, un plan est traditionellement decris par $N_x.x + N_y.y + N_z.z + d = 0$



$\overrightarrow{N} = \begin{pmatrix} N_x \\ N_y \\ N_z \end{pmatrix}$ etant la normal du plan, d se deduisant ainsi : $d = \overrightarrow{N}.\overrightarrow{W}$, $\overrightarrow{W}$ etant un point quelquonce du plan.

Le rayon qui a pour equation : $\overrightarrow{R} = \overrightarrow{O} + t.\overrightarrow{V}$ avec $t \geq 0$, $\overrightarrow{O}$ etant l'origine du rayon et $\overrightarrow{V}$ etant le vecteur directeur de la droite portant le rayon, ou plus simplement la direction du rayon.

Puis on calcul le produit scalaire de la normale au plan et du vecteur directeur du rayon : $\overrightarrow{N}.\overrightarrow{V}$ si le resultat est nul on en deduit que le rayon est parallelle au plan, pour eviter d'avoir a traiter ce cas particulier on considere que le test d'intersection echoue.

Ensuite on peut determiner l'intersection de la droite definie par le rayon et du plan en calculant le parametre t tel que :

$N_x.(O_x + t.V_x) + N_y.(O_y + t.V_y) + N_z.(O_z + t.V_z) + d = 0$

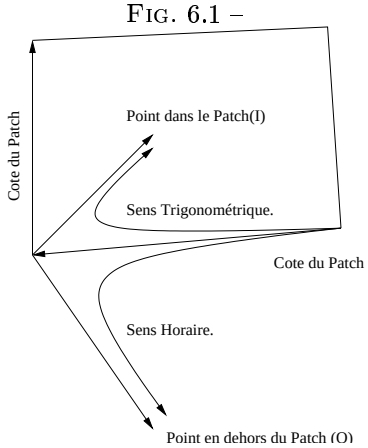
La solution de ce systeme est :

$t_o = -\frac{-(N_x.Ox + N_y.O_y + N_z.O_z + d)}{\overrightarrow{N}.\overrightarrow{V}}$

Au passage nous verifions le signe du numerateur, afin de ne pas prendre en compte les faces arriere, cela se justifie par le fait que $t \geq 0$.

On retrouve simplement les coordonnee cartesienne du point d'intersection I : $\overrightarrow{I} = \begin{pmatrix} O_{x+}t_o.V_x \\ O_y + t_o.V_y \\ O_z + t_o.V_z \end{pmatrix}$

FIG. 6.1 –



6.2.1.2 Determiner l'appartenance d'un point d'intersection au patch

Maintenant que nous avons determiné les points d'intersection entre le plan et le rayon il faut savoir si le point d'intersection est effectivement dans le patch ou pas. Cela peut se determiner de façon precise avec des produits vectoriels et des produit scalaires :

Pour cela on utilise une propriete bien connue du produit scalaire :
$\overrightarrow{U} \times \overrightarrow{V} = -\overrightarrow{U} \times \overrightarrow{V}$

Ainsi selon que le point d'intersection entre le plan et le rayon se trouve dans le patch ou en dehors du patch, le sens d'au moins un produit vectoriel avec les vecteurs obtenu a partir des cotes du patch change. On le voit bien sur le shema ci dessous, en fonction du placement du point P, l'orientation des vecteurs changent.

Ainsi, on fait un produit scalaire entre un cote du patch et le point d'intersection :
$\overrightarrow{U} = \overrightarrow{AB} \times \overrightarrow{AP}$

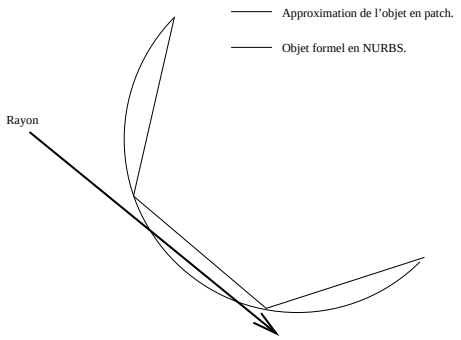
Puis on realise un produit scalaire entre $\overrightarrow{U}$ et $\overrightarrow{N}$ la normale au patch et d'aprés le signe de ce produit on peut débuire l'orientation de $\overrightarrow{U}$. Et si $\overrightarrow{U}$ n'est pas dans le meme que la normale, cela signife que P est en dehors du patch. On realise le meme test pour chaque cote succesivement.

Si apres tout ces test le point n'a pas ete rejete, cela signfie qu'il existe bien un point d'intersection entre le rayon et le patch.

6.2.1.3 Les problemes d'approximation

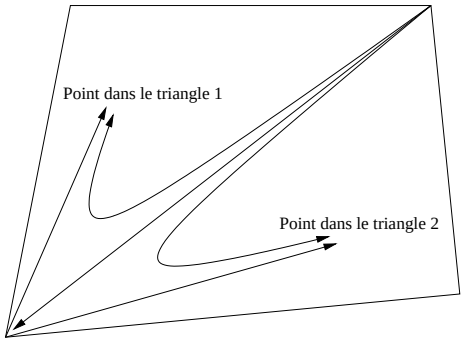
Toutefois pour etre sur que le patch selectionné n'est pas occulté par un autre patch de meme objet, on calcul la distance entre le point d'intersection trouvé et l'oringine du rayon. et on selectionne parmis tout les patchs qui sont coupés par le rayon celui qui est le plus proche.

Cette methode a l'avantage d'etre precise lorsque le maillage est non uniforme, mais il pose certains probléme lorsque les objets ne s'approximent pas bien avec des quads, en effet dans certains cas il est impossible de trouver une intersection avec un patch de l'objet a cause de petites imprecisions du a l'approximation de l'objet en patchs :



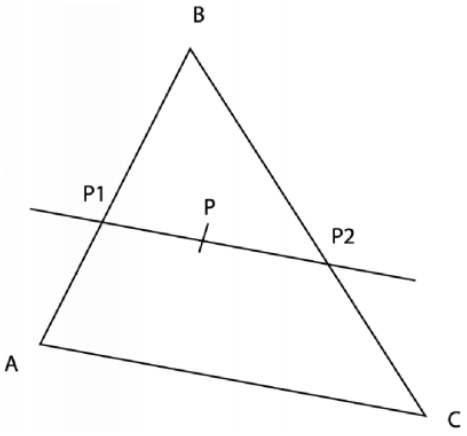
Cela se produit typiquement sur des objets tels que les spheres, qui ont tendance à nécéssiter enormément de patch pour etre approximer de facon correcte. Agmenter le nombre aurait pu résoubre le probleme, mais cela aurait considérablement augmente le temps de calcul de la radiosite sans pour autant apporter un gain sensible sur la qualite, cette solution etait donc exclue. Une autre solution envisagee etait d'utiliser un ε afin de prendre en compte l'imprecision des calculs, mais en pratique cela se revelee complique, car cela depend du degree de tesselesation des objets, de la precision des calculs, et de la taille de la scéne. Pour résoudre ce probléme nous utilisons le point d'intersection fournis par la raytracer, qui prend en compte l'équation des objets, et qui est donc beaucoup plus précis qu'une simple intersection avec des polygônes. Puis on fait un test de distance entre le point d'intersection et le centre des patchs. A noter que ce que nous appelons centre des patchs n'est pas calcule en faisant la moyenne de chacuns des cotes du patchs, mais est evalue grace aux NURBS, ce qui permet de gagner en precision. Ainsi en cas de doute on prend le patch le plus proche du point d'intersection ce qui permet de donner une couleur correcte sur les bords des objets.

Il y a un dernier probleme a resoudre avant de passer au lissage proprement dit, la separation du patch en triangle et la selection du bon triangle, en effet pour simplifier et surtout accelerer les calculs le lissage se fait sur des triangles. Cette selection se fait de maniére analogue a la methode utilisee pour savoir si un point se trouvee ou non dans le patch. On realise un produit vectoriel avec le point d'intersection et en fonction de son sens on en deduit le triangle approprie :



6.2.2 Lissage Gouraud

Une fois le patch delimite en deux triangles et que le triangle possedant le point d'intersection determine, il faut trouver la couleur de ce point. Pour cela nous utilisons un procede de lissage Gouraud implementer en 3d. Le gouraud classique est applique sur les scanlines horizontales du triangle une fois que celui ci a ete projete sur l'ecran. Etant donne que nous avons besoin de la couleur pour le ray-tracing, on ne peut pas projeter le triangle, et donc nous avons du trouver une methode 3d.



Pour trouver la couleur au point P, il faut d'abord trouver les couleurs aux points P1 et P2 et interpoler. Le probleme principale est donc de trouver P1 et P2, les points d'intersection entre la droite parallele a AC passant par P. Pour trouver P1, nous avons les deux equations vectorielles de droites suivantes :

$$\begin{aligned} P1 &= P + t * \overrightarrow{CA} \\ P1 &= A + t_2 * \overrightarrow{AB} \end{aligned}$$

Ce qui nous donne un systeme de 3 equations a 2 inconnues :

$$\begin{aligned} Px + CAx * t &= Ax + ABx * t_2 \\ Py + CAy * t &= Ay + ABy * t_2 \\ Pz + CAz * t &= Az + ABz * t_2 \end{aligned}$$

En resolvant ce systeme, nous trouvons trois couples de solutions (t, t_2). Seuls les $t_2$sont montres ici :

$$\begin{aligned} t_2 &= \frac{-(Ax * CAy - Ay * CAx + CAx * Py - CAy * Px)}{ABx * CAy - ABy * CAx} \\ t_2 &= \frac{-(Ax * CAz - Az * CAx + CAx * Pz - CAz * Px)}{ABx * CAz - ABz * CAx} \\ t_2 &= \frac{-(Ay * CAz - Az * CAy + CAy * Pz - CAz * Py)}{ABy * CAz - ABz * CAy} \end{aligned}$$

Avant de trouver P1, il faut donc trouver une solution t_2 qui n'a pas un denominateur nul. Une fois ceci fait, on peut calculer P1, et le procede est exactement le meme pour P2, en remplacant les A par des C et les AB par CB.

Une fois ces deux points trouves, il faut maintenant determiner leur couleur. Les procedes sont aussi identiques pour c1 et c2, donc nous allons que detailler c1.

Il faut calculer 3 gradients pour AB : un pour chaque composante de la couleur, rouge, verte, bleue. Appelons ces gradients rg,vg,bg, et cA,cB les couleurs des points A et B.

$$\begin{aligned} rg &= \frac{cB.rouge - cA.rouge}{\left\| \overrightarrow{AB} \right\|} \\ vg &= \frac{cB.vert - cA.vert}{\left\| \overrightarrow{AB} \right\|} \\ bg &= \frac{cB.bleu - cA.bleu}{\left\| \overrightarrow{AB} \right\|} \end{aligned}$$

Une fois ces gradients calcules, on obtient c1 facilement :

$$\begin{aligned} c1.rouge &= cA.rouge + rg * \left\| \overrightarrow{(p1 - A)} \right\| \\ c1.vert &= cA.vert + vg * \left\| \overrightarrow{(p1 - A)} \right\| \\ c1.bleu &= cA.bleu + bg * \left\| \overrightarrow{(p1 - A)} \right\| \end{aligned}$$

La couleur c2 est calcule de la meme facon, et la couleur finale de meme, en interpolant entre c1 et c2 entre p1 et p2 pour avoir la couleur en p.

6.3 Interfaces

6.3.1 Interface GTK

6.3.1.1 Description

L'interface generale du projet a ete realisee en GTK (The Gimp ToolKit http ://www.gtk.org). Le choix de cette librairie est du au fait qu'elle avait deja ete utilisee lors du developpement de la librairie reseau. L'interface finale regroupe en quelques clicks la totalite des options disponibles des differents programmes qui ont ete realisees. Elle se devait d'etre simple et intuitive.

Elle est composee de trois parties qui sont respectivement : le menu general, la toolbar et l'espace de communication avec l'utilisateur.

– Le menu general :

C'est le menu classique que l'on retrouve sur toutes les applications en general. Il permet les fonctions de bases comme ouvrir un fichier ou encore quitter l'application. Les parametres de l'application comme les differents chemins d'acces aux applications sont reglables via le menu preferences. Enfin la section aide de ce menu propose une petite explication pour utiliser l'aplication sans avoir a lire la documentation generale du programme.

– La toolbar :

Les quatres fonctions principales du programme sont accessibles graces aux quatres boutons de la barre d'outils. Le premier bouton reprend la fonction d'ouverture de fichier de description de scene. Apres avoir selectionner un fichier, un bouton preview permet de lancer le programme de previsualiation de la scene en OpenGL. La fonction principale du programme est accessible par le bouton start qui entraine le calcul de la scene avec les options choisies. Enfin, il est possible de revoire le dernier rendu effectuer grace au bouton view.

– L'espace utilisateur :

Il est compose de quatres onglets qui separent par categories les options disponibles. Le premier onglet contient un editeur de texte qui affiche le contenu du fichier de description de la scene. Le second regroupe les differentes options du raytracer. L'utilisateur peut regler les dimensions de l'image de sortie, le nom de l'image, l'activation de l'antialias et les reglages internes a l'antialias. L'onglet radiosite contient l'option d'activation de la radiosite ainsi que deux parametres de reglages internes a la radiosite. Enfin, la section network permet de choisir si le rendu va etre calcule sur une ou plusieurs machines. Elle contient aussi un tableau ou l'utilisateur peut rentrer la liste des machines qui vont aider au calcul de la scene.

6.3.1.2 Realisation

Le programme gratuit glade (http ://www.glade.gnome.org) permet en theorie de construire facilement des interfaces dont le code source est genere en C. En realite, ce logiciel ne permet pas d'entrer le code correspondant aux differentes actions comme d'autres outils de developpement visuel comme Visual C++ pourrait le faire. Il genere uniquement le code permettant d'afficher les

differents Wigets (objets visuels) au lancement du programme compile. Un autre inconvenient de ce programme est qu'il genere des centaines de lignes de codes d'appels aux differents Widgets qui rende vite le code illisible.

L'appel aux differents fonctions est realise grace a des brachements entres les signaux emis par les Widgets et les fonctions. Par exemple, pour assigner le signal "clicked" du bouton "button1" a la fonction start_application(), il faut entrer :

```
gtk_signal_connect(GTK_OBJECT(button1), \
"clicked", GTK_SIGNAL_FUNC(start_application), NULL) ;
```

A l'interieur de ces fonctions, il est possible d'acceder a l'objet qui emet le signal et a un autre objet passe en parametre. Dans bien des cas, d'autres appels aux Widgets sont necessaires et ne sont possibles que si les Widgets sont declares en variables globales, ce qui deteriore la qualite du code. Ainsi, chaque petite fonction devient tres difficile a realiser si l'on ne veut pas mettre les Widgets en variables globales. Ces problemes ajoutes a une masse de code illisible rend la creation d'une grosse interface particulierement difficile.

6.3.2 Viewer GTK

Le viewer d'image integre au projet est l'interface aux fonctions realisees dans notre librairie pic. Ce viewer permet de voir les images du type :

- png
- ppm
- jpg

Il utilise respectivement les librairies libpng, libppm et libjpeg. L'interface de ce viewer n'est autre qu'un buffer de pixel rgb qui est rempli en chargeant le fichier que l'utilisateur veut regarder et affiche ensuite a l'ecran dans une simple fenetre. Ainsi, pour visualiser une image, il suffit juste de lui preciser en argument l'image et il detecte les parametres de l'image et l'affiche. Il a l'avantage d'etre tres facile d'utilisation et surtout tres rapide pour afficher.

6.3.3 Previewer OpenGL

Il est souvent difficile de modeliser une scene pov car on est oblige de lancer regulierement des rendus pour verifier que l'on a bien place les objets, que l'angle de la camera est satisfaisant ...

Pour repondre a ce besoin, nous avons ecrit un previewer en openGL, qui permet de verifier rapidement que l'on a place les objets correctement. Il permet de plus de tester l'illumination de la scene. Le screenshot disponible dans le manuel d'utilisation montre les differents boutons : le rollout "display" regroupe les fonctions d'affichage, le rollout "settings" regroupe les reglage d'illumination, enfin, la partie basse de l'interface permet d'acceder aux boutons de modification de la camera. L'affichage 3d est realise en openGL, les widgets sont geres par la librarie GLUI.

On a donc les boutons (leur effet est detaille dans la documentation du logiciel) :

- Radiosite : lance la radiosite
- Subdivide : lance la subdivision
- Objects : permet d'afficher les objets en fil de fer
- Axis : permet d'afficher les axes (tres pratique pour changer la position des objets).
- Uniform : permet de passer en subdivision uniforme.
- Iterations : nombre d'iterations de Southwell
- Energie limit : limite d'energie de Southwell
- Min Area. : aire limite pour la subdivision
- Subdiv. Level : niveau de subidvision (pour la subdivision uniforme).
- Update : met a jour l'image en fonction des parametres
- Rotation et translation : permet de deplacer la camera.

Outre de remplir ses fonctions de previsualisation, le previewer a surtout ete un precieux outil pour tester la radiosite et la subdivision : par exmple on se rend beaucoup mieux compte d'une mauvaise subdivision si elle est affichee en trois dimension que si elle est affichee sous forme d'une liste de chiffres ...

Conclusion

Apres six mois de developpement, zRcube atteint les objectifs que nous nous etions fixes : gerant les principales fonctions de pov, la qualite de rendu etant similaire a celle de ce dernier sur des images raytracees, de plus, et contrairement a pov, zRcube est capable de combiner au rendu raytracing un calcul de l'illumination globlale base sur un algorithme de radiosite et il est capable de rendre sur un reseau.

On peut donc resumer ce qu'est actuellement capable de faire zRcube :

- Parsing de scenes pov : Le parser est code en utilisant les outils flexx++ et bison++, il est donc facile a mettre a jour et est particulierement performant, il integre de plus un preprocesseur. Nous avons ajoute deux extensions au language de description de POV (#zfor et energy) afin de tirer parti au maximum des possibilites de zrcube.
- Rendu raytracing : le rendu en raytracing gere la plupart des fonctions de pov : reflection, refraction, textures, color maps... Il prend de plus naturellement en compte l'ensemble des options associees (indices de reflection et de refraction par exemple).
- Anti-aliasing : les scenes sont anti-aliasee grace a une methode basee sur le super-sampling.
- Rendu parallele : Les rendus en raytracing peuvent utiliser la puissance de calcul d'un reseau grace a une architecture client/serveur. Ils peuvent de plus utiliser plusieurs processeurs si la machine est multi-processeurs.
- Illumination en radiosite : l'illumination de la scene est calcule grace un algorithme de radiosite progressive hierarchique basee sur l'iteration de Southwell. Les surfaces de sont pas polygonalisees mais converties en NURBS afin de conserver l'exactitude mathematique des objets formels.
- Combinaison du raytracing et de la radiosite : zRcube est capable de combiner le realisme de l'illumination en radiosite avec le raytracing grace a un algorithme en deux passes.
- Previsualisation rapide de la scene en openGL : permettant de plus de tester les meilleurs reglages d'illumination et de regler la camera en permettant a l'utilisateur de se deplacer dans la scene.
- Interface GTK : afin de simplifier l'utilisation de zRcube, les lignes de commande sont doublees d'une interface gtk permettant de lancer facilement de rendus sur le reseau, de lancer des previsialisations et plus generalement de regler toutes les options de zRcube.

Toutes ces fonctionnalites ont permis a zRcube de se faire une petite place parmis les moteurs de rendu, ainsi les webmasters de povray.co.uk et de rendermania.com ont ecrit un petit article sur ce projet, nous avons de plus recu des mails du monde entier de graphistes ayant teste notre logiciel : il a ete telecharge par pres de 200 personnes, notre site web comptabilisant plus de 750 visites.

ZRcube est un gros projet, et les principaux points sur lesquels nous avons appris touchent probablement a la maniere de gerer un tel projet, nous forcant a travailler au maximum en groupe. L'adoption de CVS nous a beaucoup aide dans la gestion et la mise a jour de nos sources. Nous avons de plus bien sur beaucoup appris sur les methodes de rendu realiste d'images de synthese, d'abord en implementant certaines dans zRcube, mais aussi en lisant des theses et des rapports de recherche a leur sujet lors de nos recherche bibliographiques.

En regard de notre bien meilleur connaissance de ces techniques, nous avons maintenant de nombreuses idees pour accelerer le rendu et ameliorer la qualite de l'image. Une bonne approche pourrait etre d'utiliser la methode du Raytracing de Monte Carlo developee cette annee par Gregory Ward Lawson qui permet de prendre en compte les interactions lumineuses comme le fait la radiosite mais evite les problemes lies a la necessite de subdiviser le surfaces en elements finis pour calculer la radiosite. Il semble que la plupart des moteurs de rendu professionels soient en trains d'implementer cette methode. D'autres extensions de zRcube pourraient s'averer interessantes, on peut notamment noter le modele de radiosite etendue, sense permettre de prendre en compte les refraction et reflection en radiosite, ce qui permettrait de se passer de la passe de raytracing.

Bibliographie

[1] Informatique graphique, methodes et modeles. *B. Peroche,D. Ghazanfarpour,D. MIchelucci & Marc Roelens*

[2] Algorithmes pour l'infographie. *David F. Rogers*

[3] Modeles de Bezier, des B-splines et des NURBS. *Gilbert Demengel & Jean-Pierre Pouget.*

[4] Using NURBS surface in real-time Applications. *Dean Macri (http ://www.gamasutra.com/features/19991117/macri_ 01.htm*).

[5] NURBS Curves : A Guide for the Uninitiated. *Philip J. Schneider* (*http ://developer.apple.com/dev/techsupport/develop/issue25/schneider.html*).

[6] Le rendu par lancer de rayons : *http ://raphaello.univ-fcomte.fr/IG/RayTracing/lancer.htm*

[7] Ray tracing news : *http ://www.acm.org/tog/resources/RTNews/html/*

[8] A Rapid Hierarchical Radiosity Algorithm. *Pat Hanrahan, David Salzman, Laury Aupperle*

[9] Parallel hierarchical Radiosity for Complex Buildings Interiors (rapport de recherche de l'INRIA). *D. Menevaux, Kadi Bouatouch*

[10] Coarse-Grained Parallelism for hierarchical Radiosity using Group Iterative Methodes. *T. Al Funkhouser*

[11] Hierarchical Radiosity : A Simple, Practical, Robust, and Efficent Implementation. *Kenneth E. Hoff*

[12] Programmer avec les threads unix. *Charles Nothrup*

Annexe : charte de code

Project : zRcube
File : Charte
Date : Time-stamp : <21-Nov-2000 02 :11 :29 mandor>
Version : 0.1

1/Noms d'identificateurs :
-noms de fonctions, de classes, de variables, et commentaires en anglais.
-les noms de classe et de type commencent par une majuscule
-les noms de fonction commencent par un minuscule
-pas d'underscore dans les noms de fonction et de variables
-> utiliser les les majuscule, exemple : displayObjects() ;

2/En-tete de fichiers

```
/*************************************************************************\
 *                  Project : zRcube                                     *
 *                  Filename : charte.cpp                                *
 *                  Version : 0.1                                        *
 *                  Authors : JB Mouret                                  *
 *                  Last change :                                        *
 *                  Time-stamp : <21-Nov-2000 01 :11 :04 mandor     *
 * Version history : 11/21/00 : first release                            *
 * Notes :                                                               *
\*************************************************************************/
```

3/Organisation des classes :
Exemple : definition de la classe "classtype"

```
#ifndef CLASSTYPE_H
#define CLASSTYPE_H
class ClassType {
public :
        //Interfaces
        ....
        //Functions
        ....
```

```
        //Vars
        ....
private :
        //Functions
        ....
        //Vars
};
#endif
```

4/Noms de fichiers :

Chaque classe est repartie dans deux fichiers :

-un fichier avec le suffixe ".h" contenant la definition de la classe

-un fichier avec le suffixe ".cpp" contenant le corps de la classe

Les fichiers portent le nom de la classe correspondante (et portent donc une majuscule) :

- *.cpp : <Nom_classe>.cpp
- *.h : <Nom_classe>.h

5/Interfaces :

Les interfaces sont construit d'un prefixe, suivit du nom de la variable avec une majuscule.

Le prefixe pour obtenir la valeur d'un variable est "get", celui pour en changer la valeur est "set" :

<Type T> get<Id_var>(void);
void set<Id_var>(<Type> T var);

exemple :

pour acceder a la variable reflectedRay, de type Ray : Ray getReflectedRay(void);

pour changer la valeur de la variable reflectedRay : void setReflectedRay(Ray var);

5/Divers

-Les constantes sont definies dans le .h

-Specifier les void et les int meme si ils ne sont pas necessaires.

-Dans la mesure du possible, mettre une rapide description de chacune des fonctions dans le .h

-Pour les tableaux (listes statiques), utiliser le template que nous coderons, de meme pour les listes chainees.