Crayon[1]

Jérôme Grimbert

2 juin 2005

[1]Document de conception pour Crayon, un remplaçant de POV-Ray en C++ compatible avec les architectures multiprocesseurs et multicores.

Table des matières

Table des figures

Liste des tableaux

Première partie

Objectifs

Chapitre 1

C'est quoi, Crayon ?

Les objectifs de Crayon sont :

- moteur de rendu multi-thread
- les capacités de POV-Ray 3.6 avec des patches en plus
- un code Orienté objet, en C++ propre, sans exceptions, utilisant les STL mais pas les verbes esotériques comme vsnprint...
- suppression de la notion de version du SDL (et donc pas de compatibilité garantie d'une version à l'autre)
- Pas d'I/O en format libre
- capacité d'exportation en format textuel et normalisé d'une scène
- un accès (en lecture) depuis le SDL aux valeurs et sous-objets en tant qu'expressions (bref pouvoir aller rechercher le rayon d'une sphére...)
- une syntaxe entierement explicite (et non plus dépendant de la position d'une valeur)
- les "patterned_finish"
- une collections de patterns homogène (plus de cas particulier pour checker/hexagon...)
- la lecture des gif
- une notion d'objet plus claire (hollow de POV-Ray est vraiment à casser)
- une licence libre ?

1.1 Modelisation de POV-Ray

En UML, ça donne a peu près 1.1.

Le modèle UML approximatif de POV-Ray montre quelques complications dues à l'historique des développements, en particulier la notion de layered_texture et de simple_texture fait un peu double emploi avec patterned_texture, si on considère la superposition de texture comme un motif comme les autres.

1.1.1 Evolution du modèle

Il est possible de rendre le modèle un peu plus homogène, en ajoutant un patterned_finish et en simplifiant la superposition de texture. Ce pose alors la question du double niveau patterned_texture vs patterned_pigment / patterned_finish / patterned_normal.

Si du point de vue du langage, il peut y a voir un intérêt à garder ce double niveau (factorisation de la description), elle n'est pas forcement utile du point de vue du code : certes, on risquerait d'évaluer trois motifs au lieu d'un seul dans certains cas, mais au prix d'une complexité accrue du code, puisque les motifs sont possibles à différents niveaux. (1.2)

1.1.2 Encore un peu plus loin

Tout comme l'objet material qui n'existe pas vraiment, en supprimant l'objet texture du modèle de données, on obtient un modèle un peu plus simple (1.3).

1.1.3 Combien de faces pour cette forme ?

Une question qui revient de temps en temps : la texture interieur d'une forme doit-elle forcement être identique à la texture extérieure ?

On pourrait éventuellement considérer celà au niveau du modèle de donnée (1.4).

Mais peut-être qu'un motif ferait tout aussi bien, sans alourdir systèmatiquement la consommation mémoire...

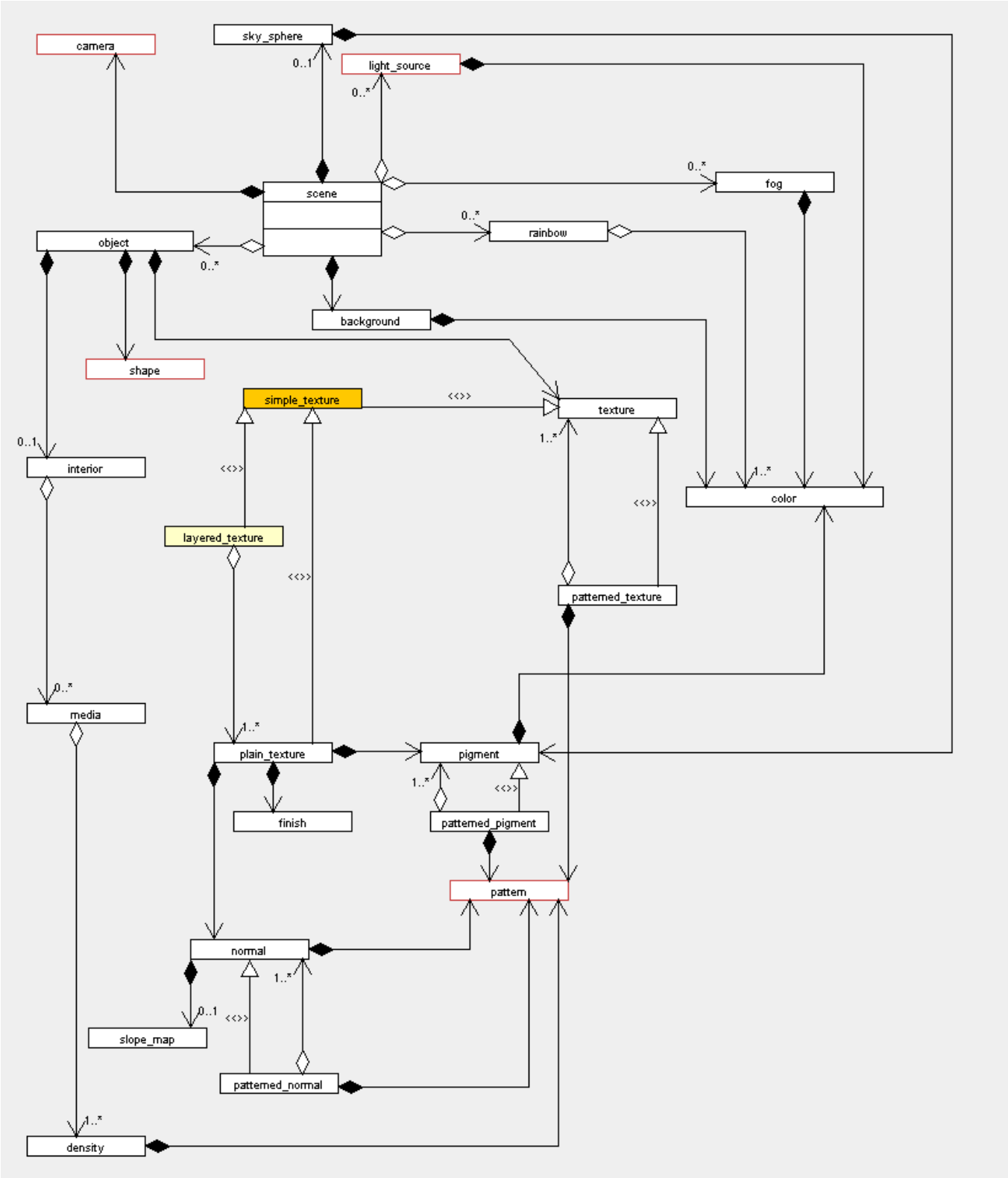

FIG. 1.1 – Une modèlisation approximative des données de POV-Ray

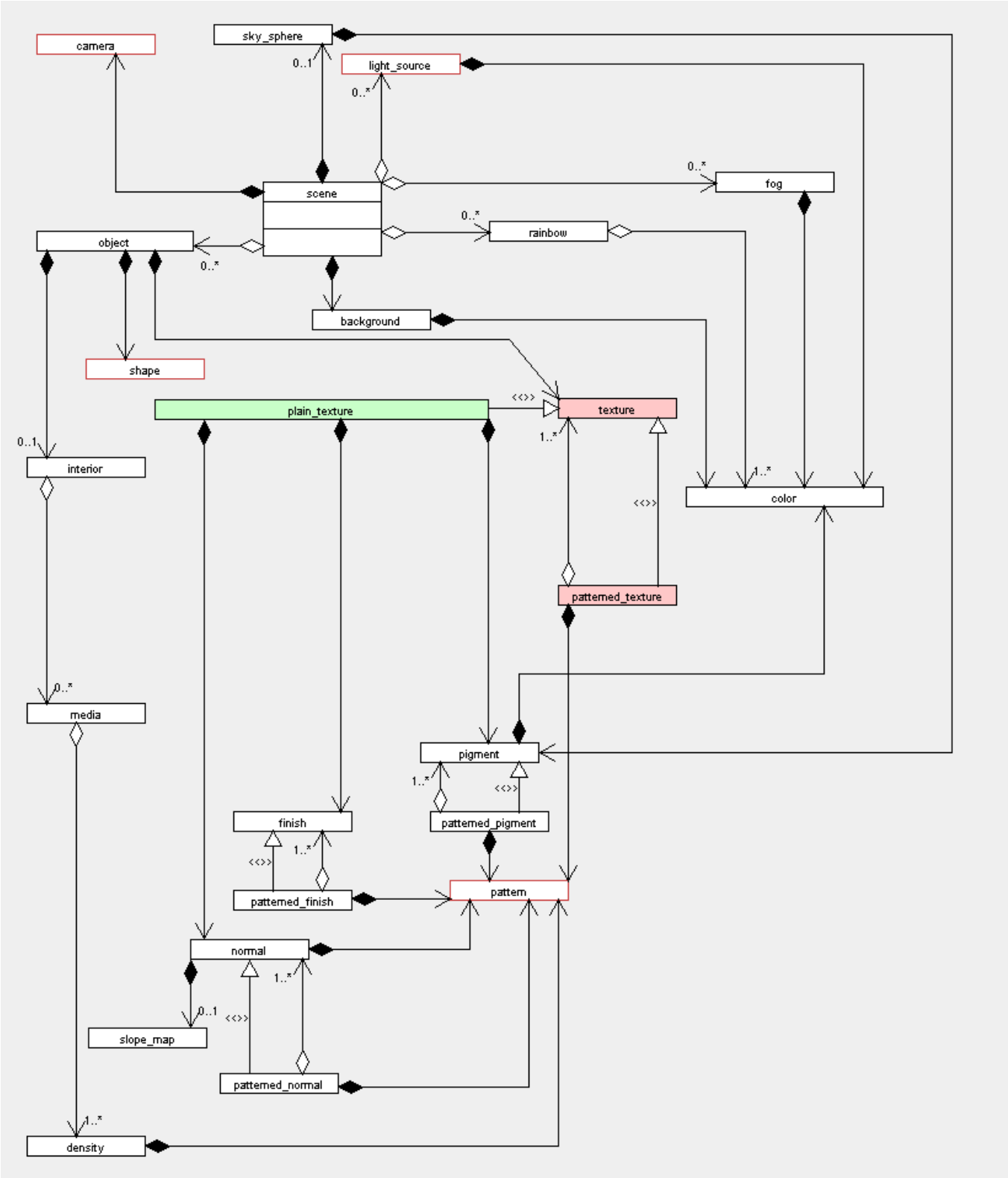

FIG. 1.2 – La superposition peut n'être qu'un motif comme les autres, et le finish peut être rendu homogène

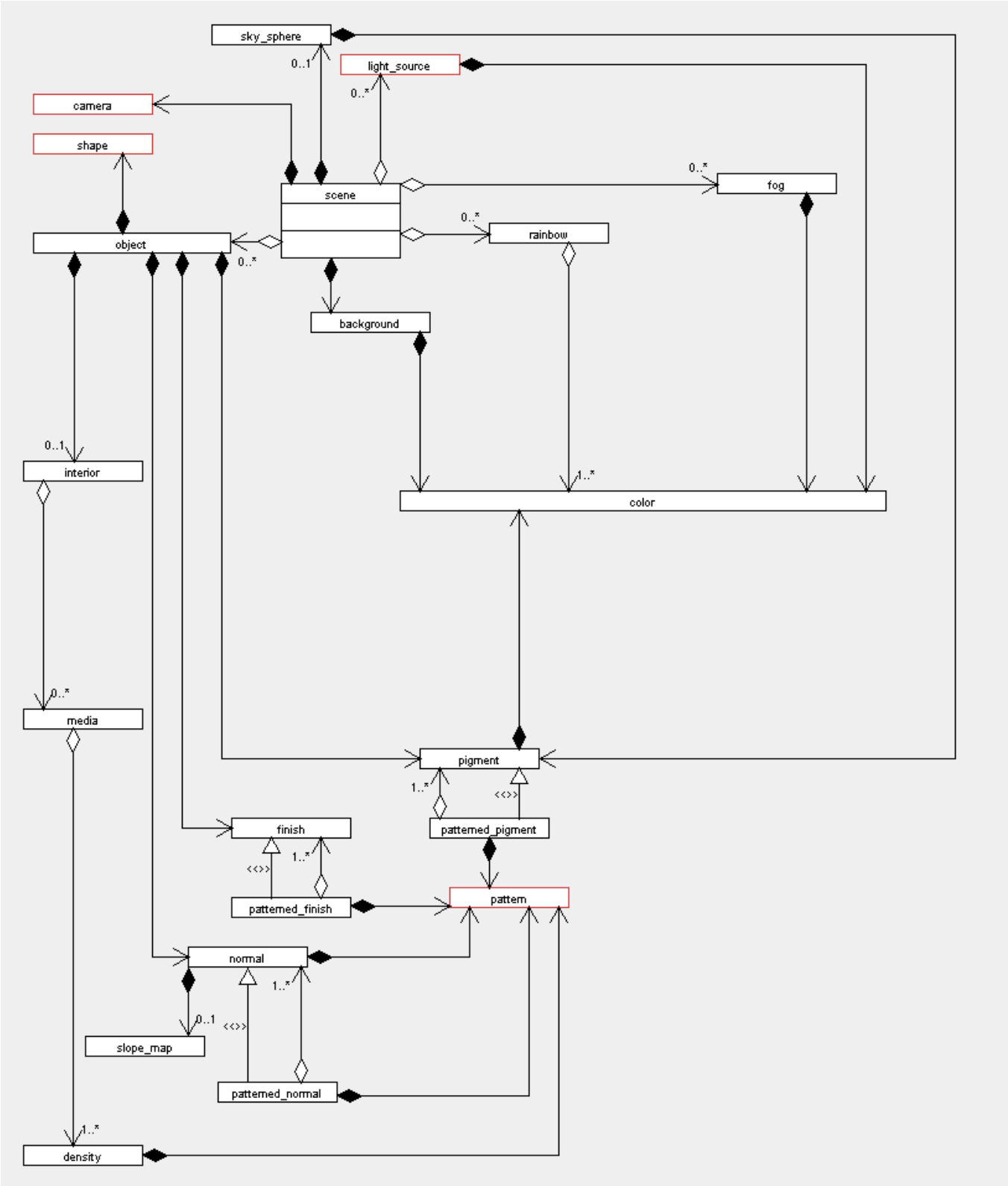

FIG. 1.3 – Simplification en supprimant l'objet texture

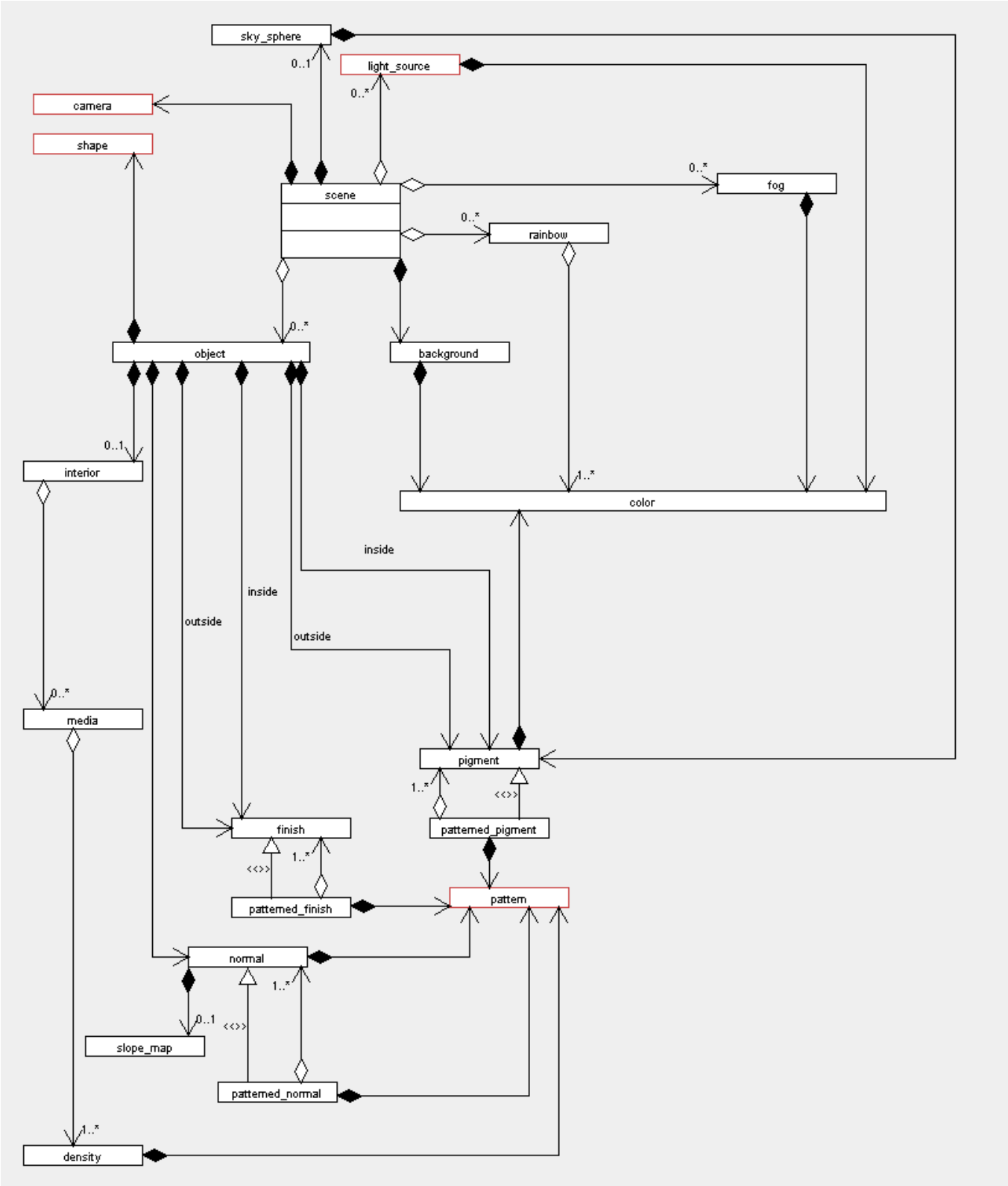

FIG. 1.4 – Séparation des faces dans le modèle de donnée

Chapitre 2

Un peu de qualité

2.1 Rappel de définitions ISO 9126

ISO-9126 n'étant disponible qu'en anglais, les termes originaux sont fournis entre parenthèses.

2.1.1 Capacité fonctionnelle (Functionality)

Ensemble d'attributs portant sur l'existence d'un ensemble de fonctions et leurs propriétés données. Les fonctions sont celles qui satisfont aux besoins exprimés ou implicites.

Pertinence (Suitability) Critère portant sur la présence et l'opportunité d'un ensemble de fonctions pour les tâches spécifiées (Attributes of software that bear on the presence and appropriateness of a set of functions for specified tasks).

Précision (Accurateness) Critère portant sur l'obtention de résultats ou d'effets justes ou convenus (Attributes of software that bear on the provision of right or agreed results or effects).

Interoperabilité (Interoperability) Critère portant sur la capacité du logiciel à interagir avec des ensembles spécifiés (Attributes of software that bear on its ability to interact with specified systems).

Conformité (Compliance) Critère du logiciel faisant qu'il adhère aux normes, conventions ou lois de régulations et autres obligations similaires (Attributes of software that make the software adhere to application related standards or conventions or regulations in laws and similar prescriptions).

Sécurité (Security) Critère portant sur la capacité à prévenir les accès indus, qu'ils soient accidentels ou délibérés, aux programmes ou aux données (Attributes of software that bear on its ability to prevent unauthorized access, whether accidental or deliberate, to programs or data).

Transparence (Traceability) Critère portant sur l'effort nécessaire à la vérification de l'exactitude du traitement des données en des points particuliers (Attributes of software that bear on the effort needed to verify correctness of data processing on requiered points).

2.1.2 Fiabilité (Reliability)

Ensemble d'attributs portant sur l'aptitude du logiciel à maintenir son niveau de service dans des conditions précises et pendant une période déterminée.

Maturité (Maturity) Critère portant sur la fréquence des pannes pour cause d'erreurs dans le logiciel (Attributes of software that bear on the frequency of failure by faults in the software).

Résistance (Fault tolerance) Critère portant sur la capacité du logiciel à maintenir un niveau de performance spécifié en cas d'erreurs logicielle ou de violation de l'interface spécifiée (Attributes of software that bear on its ability to maintain a specified level of performance in case of software faults or of infringement of its specified interface).

Récupération (Recoverability) Critère portant sur la capacité du logiciel à rétablir son niveau de performance et récuperer les données directement affectées par une panne, ainsi que le temps et l'effort nécessaire pour y parvenir (Attributes of software that bear on the capability to re-establish its level of performance and recover the data directly affected in case of a failure and on the time and effort needed for it).

Disponibilité (Availability) Critère portant sur la quantité de temps où le logiciel est disponible pour l'utilisateur lorsqu'il en a besoin (Attributes of software that bear on the amount of time the product is available to the user at the time it is needed).

Dégradabilité (Degradability) Critère portant sur l'effort nécessaire au rétablissement des fonctions essentielles après une panne (Attributes of software that bear on the effort needed to re-establish the essential functionality after a breakdown).

2.1.3 Facilité d'utilisation (Usability)

Ensemble d'attributs portant sur l'effort nécessaire pour l'utilisation et sur l'évaluation individuelle de cette utilisation par un ensemble défini ou implicite d'utilisateurs.

Intelligibilité (Understandability) Critère portant sur l'effort de la part de l'utilisateur pour reconnaître les concepts logiques du logiciel et les manipuler (Attributes of software that bear on the users effort for recognizing the logical concept and its applicability).

Apprentissage (Learnability) Critère portant sur l'effort de la part de l'utilisateur pour apprendre l'utilisation du logiciel (Attributes of software that bear on the userseffort for learning its application).

Exploitabilité (Operability) Critère portant sur l'effort de la part de l'utilisateur pour le traitement et le contrôle du traitement (Attributes of software that bear on the userseffort for operation and operation control).

Explicitation (Explicitiness) Critère portant sur l'information concernant l'état du logiciel (barre de progression, etc.) (Attributes of software that bear on the software product with regard to its status (progression bars, etc.)).

Personnalisation (Customisability) Critère portant sur la capacité de personnalisation du logiciel par l'utilisateur afin d'obtenir une réduction de l'effort d'utilisation et une augmentation de la satisfaction (Attributes of software that enable the software to be customised by the user to reduce the effort required for use and increase satisfaction with the software).

Séduction (Attractivity) Critère portant sur la satistisfaction des envies et préférences latentes des utilisateurs au travers de services, comportements et présentation, au delà des demandes explicites (Attributes of software that bear on the satisfaction of latent user desires and preferences, through services, behaviour and presentation beyond actual demand).

Clarté (Clarity) Critère portant sur la clarté de l'information expliquant à l'utilisateur les tâches que le logiciel peut accomplir (Attributes of softwate that bear on the clarity of making the user aware of the functions it can perform).

Serviabilité (Helpfulness) Critère portant sur la disponibilité de mode d'emploi expliquant à l'utilisateur comment interagir avec le logiciel (Attributes of software that bear on the availability of instructions for the user on how to interact with it).

Convivialité (User-friendliness) Critère portant sur la satisfaction des utilisateurs (Attributes of software that bear on the users' satisfaction).

2.1.4 Rendement (Efficiency)

Ensemble d'attributs portant sur le rapport existant entre le niveau de service d'un logiciel et la quantité de ressources utilisées, dans des conditions déterminées.

Rapidité (Time behaviour) Critère portant sur les temps de réponses et de calculs ainsi que les vitesses de traitements (Attributes of software that bear on response and processing times and on throughput rates in performing its function).

Consommation (Resource behaviour) Critère portant sur la consommation des ressources utilisées ainsi que leur durée d'utilisation durant le traitement (Attributes of software that bear on the amount of resource used and the duration of such use in performing its function).

2.1.5 Maintenabilité (Maintainability)

Ensemble d'attributs portant sur l'effort nécessaire pour faire des modifications données.

Diagnostique (Analysability) Critère portant sur l'effort nécessaire au diagnostique des insuffisances ou des causes d'erreurs, ou à l'identification des parties à modifier (Attributes of software that bear on the effort needed for diagnosis of deficiencies or causes of failures, or for identification of parts to be modified).

Variabilité (Changeability) Critère portant sur l'effort nécessaire pour effectuer une modification, supprimer une erreur ou changer d'environnement (Attributes of software that bear on the effort needed for modification, fault removal or for environmental change).

Stabilité (Stability) Critère portant sur le risque d'une modification a produire un effet inattendu (Attributes of software that bear on the risk of unexpected effect of modifications).

Testabilité (Testability) Critère portant sur l'effort nécessaire à la validation du logiciel modifié (Attributes of software that bear on the effort needed for validating the modified software).

Administration (Manageability) Critère portant sur l'effort nécessaire à établir ou rétablir le fonctionnement du logiciel (Attributes of software that bear on effort needed to (re)establish its running status).

Recyclable (Reusability) Critère du logiciel portant sur son potentiel à pouvoir être réutilisé entiérement ou partiellement dans un autre (Attributes of software that bear on its potential for compelete or partial reuse in another software product).

2.1.6 Portabilité (Portability)

Ensemble d'attributs portant sur l'aptitude de logiciel à être transféré d'un environnement à l'autre.

Adaptabilité (Adaptability) Critère portant sur les possibilités d'adaptations a des environnements spéficiés différents sans autres actions ou nécessités que celles fournient par le logiciel dans ce but (Attributes of software that bear on the opportunity for its adaptation to different specified environments without applying other actions or means than those provided for this purpose for the software considered).

Installation (Installability) Critère portant sur l'effort nécessaire à l'installation du logiciel dans l'environnement spécifié (Attributes of software that bear on the effort needed to install the software in a specified environment).

Compatibilité (Conformance) Critère portant sur l'adhésion du logiciel aux normes et conventions relative à la portabilité (Attributes of software that make the software adhere to standards or conventions relating to portability).

Remplacement (Replaceability) Critère portant sur les possibilités et les efforts pour utiliser le logiciel en lieu et place d'autres logiciels dans son environnement (Attributes of software that bear on opportunity and effort using it in the place of specified other software in the environment of that software).

2.2 Evaluation des critères

Les valeurs s'étale de 0 (sans importance, à ignorer) à 5 (très important).

2.2.1 Capacité fonctionnelle

Critère	
Pertinence	4
Précision	5
Interopérabilité	2
Conformité	0
Sécurité	4
Transparence	1

TAB. 2.1 – Capacité fonctionnelle

2.2.1.1 Pertinence

Il n'est pas question de sortir des spécifications. Il n'y aura pas de simulateur de vol caché à l'intérieur !

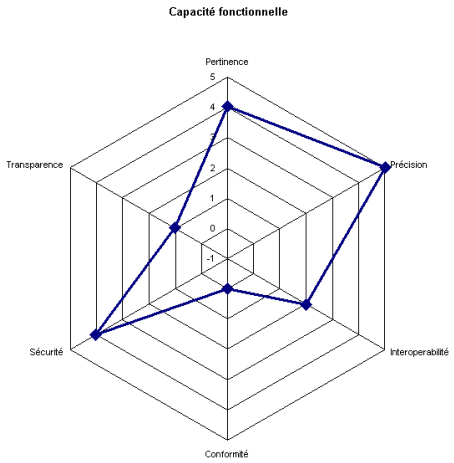

FIG. 2.2 – Capacité fonctionnelle

2.2.1.2 Précision

Il faut obtenir un logiciel qui produit ce qu'on a demandé.

2.2.1.3 Interopérabilité

Il y aura forcement une rupture de langage, mais il est envisageable de conserver une similitude au niveau de la ligne de commande. On peut rêver d'une interface avec Moray par exemple. L'importation d'objet dans des formats publiques est aussi envisageable.

2.2.1.4 Conformité

A l'heure actuelle et en droit français, le brevet ne s'applique pas au logiciel. Il est donc tout à fait possible d'utiliser des choses comme la compression GIF. Les problèmes d'exportabilité ne seront pas pris en compte.

2.2.1.5 Sécurité

L'écriture de fichiers (autre que les images), devrait se limiter à un répertoire particulier. L'écriture même des images pourrait également être limitée à un répertoire.

2.2.1.6 Transparence

Utile pour la mise au point. Le langage, en permettant un accès en lecture à la structure de données, ainsi que l'exportation en format standardisé devrait suffire à couvrir ce besoin. Au-delà, je crains qu'il ne faille recourir aux debuggers.

Critère	
Maturité	3
Résistance	5
Récupération	2
Disponibilité	5
Dégradabilité	0

TAB. 2.2 – Fiabilité

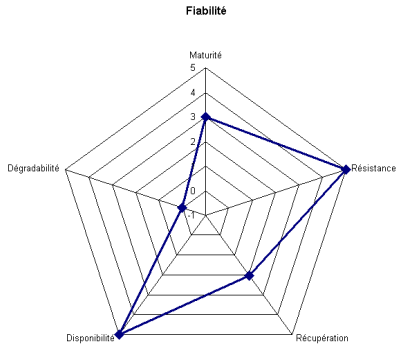

FIG. 2.3 – Fiabilité

2.2.2 Fiabilité

2.2.2.1 Maturité

C'est un objectif, il est peut-être un peu idéaliste même si un cycle de développement itératif en W devrait beaucoup aider en la matière.

2.2.2.2 Résistance

Les causes d'erreurs devraient en théorie se limiter à des erreurs dans le texte en entrée, puis à des soucis aux niveaux des ressources.

2.2.2.3 Récupération

La reprise d'une image en cours est importante, même si la remise en route peut être longue (nouvelle analyse de la scène, reconstruction de la structure de données et reprise de l'image en cours)

2.2.2.4 Disponibilité

J'ai un peu de mal avec ce critére.

2.2.2.5 Dégradabilité

Pas de fonctionnement en mode dégradé prévu.

2.2.3 Facilité d'utilisation

Critère	
Intelligibilité	5
Apprentissage	4
Exploitabilité	1
Explicitation	1
Personnalisation	1
Séduction	1
Clarté	4
Serviabilité	3
Convivialité	3

TAB. 2.3 – Facilité d'utilisation

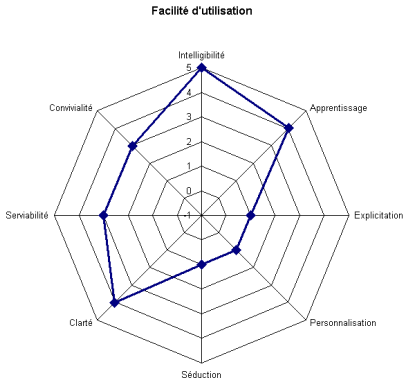

FIG. 2.4 – Facilité d'utilisation

2.2.3.1 Intelligibilité

Indispensable pour le langage.

2.2.3.2 Apprentissage

On profitera du développement en W pour augmenter au fur et à mesure la documentation avec des travaux dirigés.

2.2.3.3 Exploitabilité

C'est essentiellement une ligne de commande, avec la possibilité d'interrompre le logiciel (pour reprendre plus tard ou pas).

2.2.3.4 Explicitation

On se limitera à une ligne de statistiques, et éventuellement à l'affichage de l'image en cours.

2.2.3.5 Personnalisation

Dans les formats d'image le supportant, on pourra éventuellement mettre un commentaire défini par l'utilisateur.

2.2.3.6 Séduction

On reste à l'écoute des utilisateurs, tant que ça ne sort pas des spécifications.

2.2.3.7 Clarté

La documentation est primordiale, et le logiciel est au moins capable de donner la totalité de la syntaxe qu'il supporte.

2.2.3.8 Serviabilité

Absolument indispensable, un bon mode d'emploi.

2.2.3.9 Convivialité

Les messages d'erreurs, seules choses produites en dehors des images, devront être les plus claires possibles.

2.2.4 Rendement

Critère	
Rapidité	5
Consommation	1

TAB. 2.4 – Rendement

FIG. 2.5 – Rendement

2.2.4.1 Rapidité

Absolument indispensable, l'objectif étant si possible de monter en puissance aussi bien que possible, ainsi que de tenir l'ordre des temps de rendus actuels de POV-Ray.

2.2.4.2 Consommation

On fera attention aux très grosses structures, mais on préférera la vitesse à une occupation mémoire réduite pour l'implémentation des objets raisonnables.

2.2.5 Maintenabilité

Critère	
Diagnostique	1
Variabilité	3
Stabilité	4
Testabilité	5
Administration	2
Recyclable	0

TAB. 2.5 – Maintenabilité

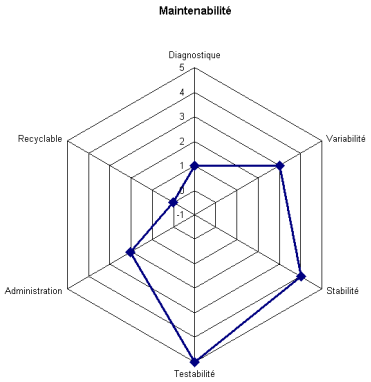

FIG. 2.6 – Maintenabilité

2.2.5.1 Diagnostique

L'utilisation et le respect d'un découpage orienté objet devrait aider, mais il est probable que l'on continue avec un debugger.

2.2.5.2 Variabilité

Toujours difficile à prédire, l'identification d'objet tel que camera, formes et motifs devraient néanmoins aider grâce au mécanisme d'héritage.

2.2.5.3 Stabilité

Si on respecte bien le découpage en objet, les propagations devraient être limitées

2.2.5.4 Testabilité

Il faut au moins une scène d'illustration pour chaque chose, donc ça devrait fournir une base de non-régression au fur et à mesure des développements.

2.2.5.5 Administration

Le plus difficile, ça va être d'obtenir le premier qui marche (une caméra, une source lumineuse, un motif, une forme et un format de fichier en sortie), ensuite c'est un développement en W, avec ajouts successifs de fonctionnalités.

2.2.5.6 Recycable

Sans importance. Mais ce n'est pas une raison pour écrire n'importe comment.

2.2.6 Portabilité

Critère	
Adaptabilité	5
Installation	4
Compatibilité	5
Remplacement	1

TAB. 2.6 – Portabilité

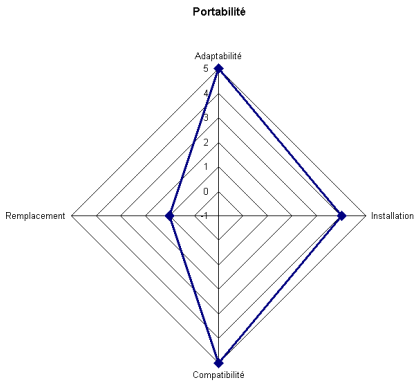

FIG. 2.7 – Portabilité

2.2.6.1 Adaptabilité

Le developpement se fera avec un compilateur C++ et make. L'ensemble des bibliothèques utilisées fera partie du source fourni.

2.2.6.2 Installation

Un binaire, une documentation et un ensemble de fichiers d'include et d'exemples (et éventuellement, les sources).

2.2.6.3 Compatibilité

Développement en C++, avec utilisation des STL mais surtout pas des verbes C trop récent. La référence du compilateur est le GNU, celle de la libC, le C89.

2.2.6.4 Remplacement

On rêve de prendre à terme la place de POV-Ray (et les autres ?), même si le langage ne sera pas compatible.

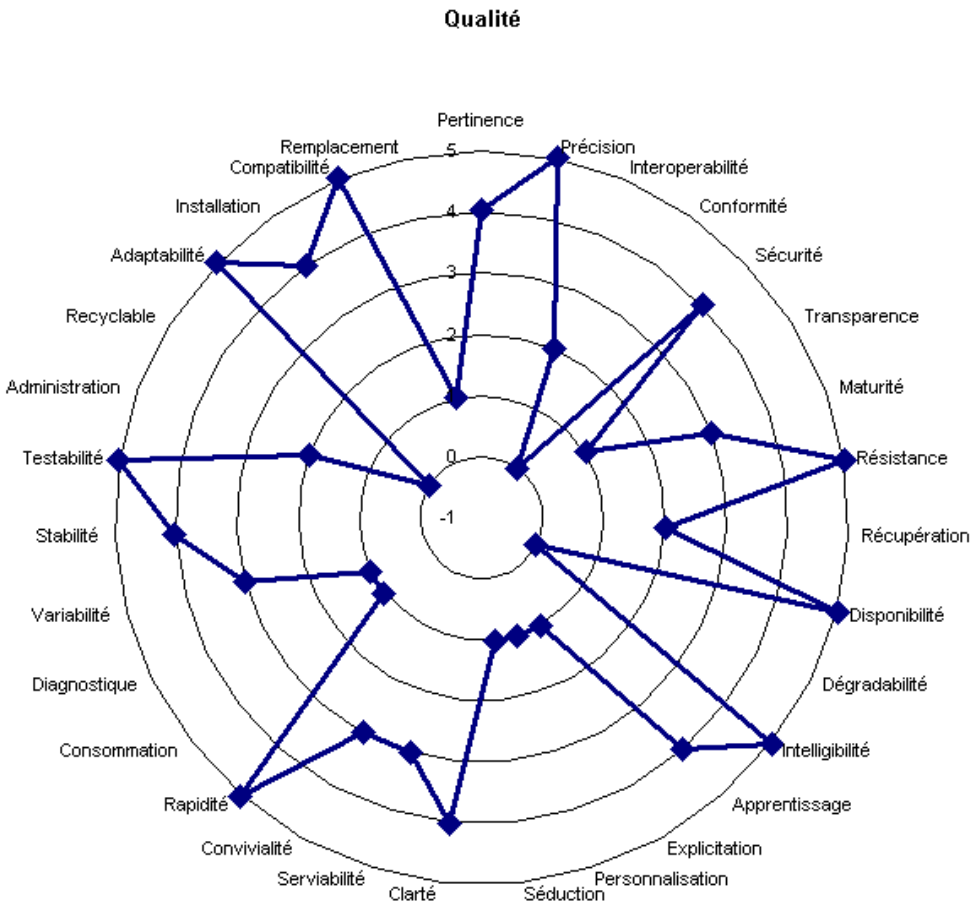

FIG. 2.1 – Vision globale des valeurs des critères

Chapitre 3

Réflexions diverses

3.1 Le Gamma

C'est franchement la plaie.

Le Gamma sert normalement à une chose : permettre l'adaptation des courbes de réponses d'une image en fonction de l'écran.

Les tubes cathodiques ont en effet une courbe de réponse qui peut être caractérisé par une équation où I est l'intensité lumineuse, et V la tension appliqué, γ étant le fameux Gamma.

$$I \simeq V^{\gamma}$$

La correction de Gamma consiste à modifier les valeurs informatiques pour que l'affichage correspond à une impression (en supposant que l'impression d'un gris 50% avec des encres noires sur du papier blanc donnent bien un gris 50%, ce qui est une autre histoire, surtout quand la qualité du papier varie).

La correction consiste à appliquer $V_c \leftarrow V_s^{\frac{1}{\gamma}}$

Pour rendre les choses encore plus compliqués, les coefficients γ, outre qu'ils sont approximatifs, sont également potentiellement différents pour chacuns des trois canons dans un tube couleur. Donc ce n'est pas un γ qu'il peut y avoir, mais trois !

Et comme si ça ne suffisait pas, il est désormais commun d'avoir des cartes graphiques qui font elles-même une correction de gamma (ce qui fait que l'image qui s'affiche subit les déformations successives de la chaine vidéo). Est-il vraiment nécessaire de continuer à s'embêter la vie dans un logiciel comme Crayon ?

On peut certes effectuer une correction sur les valeurs avant de les enregistrer mais il faut bien prendre en compte qu'il convient de bien séparer alors :

1. Le γ utilisé pour faire une correction des valeurs en sortie du moteur de rendu. Par défaut, pas de correction me semble le plus simple, donc valeur à 1 !
2. L'éventuelle correction à appliquer pour afficher l'image à l'écran, il s'agit du γ de la chaine graphique entre l'applicatif et le devant de l'écran. Un système étaloné devrait présenter normalement un γ de 1, si la carte graphique corrige effectivement l'écran.
3. La valeur de γ qu'il faut inscrire dans certains formats d'image (PNG, par exemple), et qui peut, ou pas, être utilisé pour corriger l'affichage.
4. La valeur de γ qu'il faut réellement appliquer pour un format d'image ? Est-ce bien utile ? Est-ce que les combinaisons de 1 et 3 ne suffisent pas déjà ?

Il ne s'agit là que de la chaine en sortie. En entrée, lorsqu'une image est utilisée, il peut également être nécessaire :

1. de tenir compte (ou pas !) de la valeur de γ que porte le fichier (si le format d'image contient cette information, faut-il en tenir compte ?)
2. d'appliquer une correction de γ supplémentaire.

Enfin, dernière nouvelle : le tube cathodique est mort. Les écrans LCD n'ont plus du tout une courbe de réponse en γ, mais généralement la carte graphique compense (ou cherche à compenser) ce problème. Tout ceci pour m'inciter à penser que les corrections de γ devrait le plus souvent être ignorées (avec des valeurs par défaut de 1, par exemple).

3.2 La taille des images

Les scènes sont généralement prévues pour une vision donnée.

POV-Ray a toujours mis les informations concernant la taille de l'image à l'extérieur de la scène, puisqu'une scène peut aussi bien se rendre en 320x240 qu'en 960x720.

En fait, la camera dispose d'un rapport hauteur/largeur, et l'image finale aussi.

Cependant lors de la conception de la scène, une chose est souvent implicite : l'aspect des pixels de l'image, qui est

égale au rapport des deux précédents rapports. Si celui-ci vient à changer, les choses se compliquent :

- Soit le champs de vision change (plus large, ou plus étroit)
- Soit le champs de vision reste la même, mais l'image subit une déformation afin de fournir la résolution demandée

Il me semblerait sain que, tout comme une image utilisée en entrée, la définition de la caméra définissent l'image de sortie sur un carré [0,1], et que les spécifications de tailles de l'image projettent ce carré sur le fichier de sortie, chaque coin du carré venant dans un coin du fichier en sortie.

Cependant, il pourrait être sympathique qu'au lieu de spécifier à la fois la hauteur et la largeur, on puisse spécifier l'un des deux et le pixel-ratio désiré. On obtiendrait ainsi l'image telle qu'elle a été conçue (sans en manquer un bout, ni voir des choses qui n'était pas sensé être vu).

Je pense en particulier à des synthèses d'images pour VCD et DVD : les pixels ratios ne sont pas carrés du tout, mais la mise au point sur un moniteur informatique est certainement plus pratique si l'on peut visualiser les ébauches avec les pixels ratios du moniteur, plutôt que de devoir mettre l'image en production pour pouvoir la percevoir comme elle le sera réellement plus tard.

L'autre cas, c'est la réalisation d'un fond d'écran : pour un écran 4 :3, que la résolution soit de 1024x768 ou 1280x1024 (pixel ratio différents), la surface de l'écran à couvrir sera finalement la même. Il ne devrait donc pas y avoir de changement du champs de vision, non ?

Il faudrait donc que la scène comporte, en dehors des informations pour la caméra qui permettent de définir un carré [0,1], un display_ratio[1], tandis que l'utilisateur donnerait son pixel_ratio [2], ainsi que l'une des dimensions de l'image souhaité. Crayon calculerait alors l'autre dimension de l'image automatiquement[3].

L'équation à respecter en théorie est $W \cdot P = H \cdot D$. Les ratios sont toujours un rapport $\frac{Largeur}{Hauteur}$. W est la largeur de l'image, H son hauteur, P est le pixel_ratio et D le display_ratio.

[1] 1,333 pour une vision Télé 4 :3 par exemple

[2] 1 :1 pour un pixel carré d'un écran 640x480 4 :3 par exemple

[3] Pour une image DVD 720x576, l'utilisateur donnerait par exemple -W720 -P.9375, la scène comportant un rapport 4 :3 fournirait une image de 720x576 pour un affichage 4 :3. Il est peut-être souhaitable de laisser l'utilisateur spécifier la hauteur et la largeur de l'image, en lui indiquant les deux ratios, surtout si l'image en sortie est anamorphosée.

Width	Heigth	Pixel Ratio	1/P	Display
640	480	1	1	1,3333
800	600	1	1	1,3333
1024	768	1	1	1,3333
1152	864	1	1	1,3333
1280	768	0,8	1,25	1,3333
1280	1024	1,0666	0,9375	1,3333
1600	1200	1	1	1,3333
1920	1200	0,8333	1,2	1,3333
480	576	1,6	0.625	1,3333
480	480	1,3333	0,75	1,3333
720	576	1,0666	0,9375	1,3333
720	480	0,8888	1,125	1,3333
720	576	1,424	0,7022	1,78
720	480	1,1866	0,8426	1,78

TAB. 3.1 – Exemples d'images classiques

3.3 Les couleurs en filtrage et réflection

Avec POV-Ray, il n'y a qu'un seul coefficient qui s'applique à la couleur d'un rayon réfléchi ou réfracté pour obtenir la participation en couleur au rayon incident.

Cependant, afin de gérer les réflections métalliques, le mot-clef metallic est devenu disponible dans le bloc de reflection. En sa présence, la couleur du rayon réfléchi est multiplié par la couleur à l'endroit de la réflection (outre que le montant de la participation de la réflection peut être modifié par un nombre flottant et que le modèle de cette réflection utilise le modele de Fresnel, dans lequel la couleur du rayon n'est pas modifié aux angles de réflection totale).

Il me reste cependant un gros problème : comment faire une surface noire réfléchissante ?

De même, le filtrage se limite à un seul nombre flottant : comment faire une surface verte qui ne laisse passer que le rouge ?[4]

Je pense que tout comme le pigment qui donne la contribution de la couleur en fonction de la lumière ambiente, le filtrage devrait avoir lui aussi la possibilité de spécifier

[4] Je sais que c'est à la limite une chose quasi-impossible dans la réalité, bien que dans le cas d'une couleur considérée comme un spectre et non plus comme un triplet RGB, il est tout à fait possible que la couleur apparente d'une matière et sa capacité de filtrage permettent d'avoir un objet vert (donc réfléchissant le spectre vert, absorbant le reste) dont le filtrage du spectre est tel que une source spectrale specifique située derrière se retrouve filtrée avec uniquement la partie rouge du spectre en sortie.

En fait, j'ai essentiellement un Ageratum derrière la tête, et son effet bien connu en photographie : c'est une fleur bleue qui est rose sur les photographies !

Bon, j'avoue, je n'ai pas qu'un Ageratum dans la tête, j'ai tout une composition florale !

non pas un seul nombre flottant, mais une couleur complète (voir un motif complexe de filtrage), sans que ça ne viennent interférer avec la couleur ambiente.

De même, pour la réflection, la couleur de la réflection métallique devrait pouvoir être indépendante de la couleur ambiente.

Heureusement, comme les couleurs de filtrage et réflection ne sont utiliser que pour multiplier la couleur d'un rayon existant, il n'est pas nécessaire de conserver le nombre flottant actuel, puisqu'en fait la spécification de la couleur prend sa place.

Autre petit gag de la réalité : les récentes peintures polychromes de certaines voitures de luxe. Elles peuvent être vertes vues d'un coté, et bleues d'un autre point de vue (avec un fini métallique, bien entendu, sinon c'est assez facile à simuler). Cela n'a rien de bien difficile à codifier dans un motif utiliser pour la réflection.

Ce qui m'anène à penser que le modèle de Fresnel n'est en fait qu'un cas particulier de motif de réflection/réfraction, et que par conséquent il n'est pas nécessaire de le coder en dur dans le moteur de rendu, bien au contraire.

La réflection de base s'obtiendrait simplement en spécifiant un gris à la place de l'actuelle valeur. La réflection métallique actuelle est obtenue en utilisant un motif Fresnel avec une duplication des formules des motifs du pigment ambient et la réflection variable actuelle est une simplification de la réflection métallique qui n'utilise que des gris en lieu et place des couleurs.

Cependant, il existe deux variantes de réflections variables, l'une avec et l'autre sans Fresnel, ce qui m'incite à penser qu'il y a en fait certainement trois modules possibles pour la réflection :

Simple le calcul est indépendant de l'angle de réflection

Variable le calcul est dépendant de l'angle de réflection, mais à un effet linéaire sur la couleur de la lumière

Fresnel le calcul est dépendant de l'angle de réflection, et utilise l'indice de réfraction pour avoir un comportement non-linéaire sur la couleur de la lumière.

On notera que le cas Variable est en fait modèlisable très facilement avec un simple motif basé sur l'angle d'incidence.

Au passage, on notera qu'il est nécessaire pour un rayon de conserver sa valeur d'indice de réfraction courante, afin que Fresnel puisse calculer correctement son comportement.

Il y a peut-être encore d'autres modules possible pour la réflection, comme par exemple un module anisotropique.

3.4 Azurant et ageratum

Une idée pour plus tard, quoique : au lieu de faire une simple multiplication terme à terme des vecteurs couleurs (un rayon que multiplie un pigment donne un autre rayon), pourquoi ne pas remplacer la couleur des pigments par une matrice et effectuer une multiplication vectorielle ?

Si on suppose un instant que la couleur d'un rayon n'est plus un simple triplet RGB mais un vecteur plus large contenant par exemple des informations UV[5], il serait alors simple de modèliser une scène avec de la lumière noire.[6]

Il faut donc considèrer plusieurs aspects à la couleur d'un rayon :

- un aspect interne, lui permettant d'interagir avec les pigments d'un objet
- un aspect externe, qui fournit un joli quadruplet RGBA si utile pour les pixels d'une image.

D'ailleurs, est-il bien raisonnable de continuer à utiliser le même terme et le même objet pour les rayons et les pigments d'un objet ?

Les rayons sont assimilables dans l'absolu à un vecteur à N dimensions[7], et les pigments (en fait les interactions avec la matière) peuvent s'assimiler à des matrices carrés de dimensions N, dont actuellement seule la diagonale n'est jamais nulle[8].

Rien n'empécherait de remplacer la couleur d'un rayon par une valuation plus fine d'un échantillonnage d'un spectre de longueur d'onde[9], indépendament de la conservation des pigments en l'état.

De même, le changement de la représantation des pigments serait tout à fait possible en conservant des couleurs de rayon inchangé.

Ce qui doit nous amener à deux choses :

- Les sources lumineuses doivent recevoir des spécifications de couleur de rayons, et non de pigments

[5] UV, pour Ultra-Violet, et non pas UV-mapping.

[6] Donc de simuler le comportement des azurants, le produit des lessiviers qui rend votre chemise plus blanche que blanche. Voilà pour la première partie du titre. L'ageratum est une fleur bleue, mais qui prise en photo est rose, parce que les sensibilités de l'oeil et de la pellicule ne sont pas identiques (enfin, ça dépend des pellicules, et je ne parle pas des diapos)

[7] trois pour du RGB, faut-il un quatrième pour gérer le A ? j'ai quelques doutes sur ce sujet

[8] D'où l'actuel stockage sous la forme d'un vecteur, puisque c'est cette seule information qui est intéressante.

[9] Ce qui n'implique nullement, loin de là, que la réalisation des images de diffractions en soient pour autant possible plus facilement. Seule l'utilisation des photons permet d'obtenir une véritable figure de diffraction.

Au passage, on notera que les photons doivent conserver des informations de couleurs de rayons, ce qui est un moindre mal, surtout si on remplace les pigments par des matrices de dimensions élévées.

– le calcul de multiplication d'un pigment par un rayon est une fonction friend des deux classes[10]

3.4.1 Media

Je ne sais pas que mettre pour les médias.

Les emitting sont certainement semblables à des sources lumineuses.

Les deux autres types (scattering et absorption) sont plus flous : scattering va utiliser une autre source lumineuse, il lui faudrait donc en théorie un pigment, et j'ai l'impression qu'aborption gagnerait à utiliser une couleur de source lumineuse.

D'un autre coté, la différence entre scattering et absorption est principalement que scattering trace d'autres rayons, alors qu'absorption continue le rayon en ligne droite.

3.4.2 Plusieurs rayons

Autant je ne conçois pas de problème à avoir plusieurs types de pigments[11], autant il faut bien être conscient d'une chose : la couleur du rayon de la camera est la somme des couleurs provenant des lumières diffusées, de l'interaction de l'éclairage ambient avec le pigment de l'intersection, des rayons réfractés et réfléchis[12].

Les rayons réfractés et réfléchis héritent à priori de la classe du rayon incident, une addition n'est donc pas trop difficile à envisager. Mais les informations de diffusions et d'ambience arrivent en droite ligne des spécifications des sources lumineuses.

Par conséquent, à moins de se restreindre à une seule classe concrête de rayon (ce qui est toujours possible, après tout), il est nécessaire :

– que la caméra indique quelle classe de rayons elle reçoit[13]
– qu'il existe une fonction d'addition de rayons pour chaque paire ordonnée de combinaisons des classes dérivées du rayon[14]
– Le media de type absorption nécessite probablement également une fonction de soustraction[15], à moins d'être implémenté comme un pigment[16].

[10] Ce qui implique, pour se garder la porte ouverte pour des évolutions des couleurs de rayons et de pigments, que les classes couleurs et pigments soient des classes abstraites dont l'une des déclinaisons de chacune corresponde à l'implémentation de POV-Ray.

Ce qui implique également que le nombre de fonction friend à écrire pour la multiplication est le produit des possibilités réelles : c'est toujours sympathique à savoir avant de se lancer dans l'exploration des variations possibles.

La fonction friend est bien entendu entièrement virtuelle pour les classes abstraites.

[11] Par exemple, un vecteur de dimension 1 pour Gray, de dimension 3 pour RGB et de dimension 4 pour CMYB, voire même de dimension N pour SpectreN, ou même une matrice pour gérer les azurants.

[12] Et encore, je passe sur les photons et la radiance.

[13] Enfin, « émet », puisqu'on fait du rendu en marche inverse.

[14] Autrement dit, le carré du nombre de classe de rayon. Je trouve que ça fait beaucoup !

[15] Là encore, il y aurait autant de fonctions de soustraction que de fonctions d'additions !

[16] Ce qui à la réflexion semble être la meilleure solution. Rien n'empêche d'avoir une classe pigment qui corresponde exactement au même vecteur que la couleur d'un rayon.

Deuxième partie

Conception

Chapitre 4

Modèle de donnée

En 4.1, on peut observer le modèle de donnée de Crayon. Les classes en fond coloré sont abstraites[1], le fond jaune est pour une classe dont les sous-classes sont décrites, le fond rose est pour une classe dans les sous-classes ne sont pas décrites.

Vis-à-vis de POV-Ray, on remarquera l'apparition de classes abstraites (Infini et Effet, par exemple), qui simplifie un peu le nombre de type d'objet de la scene. Au passage, on peut découvrir avec émerveillement que l'utilisation de background et sky_sphere sont exclusifs l'un de l'autre : c'est normal, les deux cherchent à s'occuper des rayons de la caméra n'ayant atteint aucun obstacle.

Si l'on voulait rajouter des Lens-Flares, par exemple, il s'agirait probablement d'une sous-classe d'effet. Encore faudrait-il avoir une idée de l'algorithme à utiliser, mais c'est une autre histoire.

En fait, suite à la section 3.4, le modèle de donnée de Crayon devrait plutôt ressembler à 4.2. Outre la séparation de Color et RayColor[2], la notion de pigment de POV-Ray sur un objet est découpée en quatre :

1. Le pigment de la surface (celui de POV-Ray), utilisé pour les lumières diffuses et la lumière ambiente[3]
2. Le pigment pour la réflexion. POV-Ray utilise celui de la surface affecté d'un coefficient de reflexion. Il est plus souple, je trouve, de pouvoir spécifier ce qu'on veut exactement
3. Le pigment pour la réfraction. C'est l'ancien filtrage de POV-Ray, la direction du rayon est déterminé par l'indice de réflexion définit dans Interior à partir du rayon incident.
4. Le pigment pour la transmission. C'est l'ancienne transmission de POV-Ray, la direction du rayon est identique au rayon incident.

Pour les coupeurs de cheveux en quatre, j'avoue avoir utiliser un raccourci dans les schémas : la plupart des listes (comme dans patterned_*, star, rainbow) sont plutôt des maps au sens POV-Ray que de simple listes. La différence est que chaque entrée dans la liste se retrouve affecté d'un coefficient, l'interprétation de ce coefficient dépendant de l'utilisation de la map.

Star et Rainbow nécessite de pouvoir interpolé deux RayColors, donc outre une fonction probablement nécessaire d'amplification[4] des valeurs d'un RayColor, il sera nécessaire d'avoir une fonction d'interpolation tout comme on doit déjà avoir une fonction d'addition. En fait, il est peut-être possible de n'avoir qu'une seule fonction d'addition pondérée :

- Dans le cas d'une addition normal, les poids sont tous égaux à l'unité
- Dans le cas d'une interpolation, la somme des poids est égale à l'unité

Vu la croissance de la cardinalité des fonctions de ce genre en cas de croissance des sous-classes de RayColor, n'avoir qu'une seule fonction à l'interface est certainement préférable.

Il manque dans 4.2 la variabilité de la méthode d'échantillonnage des médias qu'on trouve dans 4.3, où RayColors et Color ont également été francisés afin d'essayer de rendre plus facile leur compréhension. La notion d'onde est ce qui se rapproche certainement le plus de RayColors, bien que la notion de spectre puisse également correspondre, si l'on considère la décomposition d'une onde en son spectre. Cependant, la notion de spectre est trop limitative, puisqu'on peut difficilement y faire entrer d'autres informations, comme par exemple la polarisation. L'onde

[1] ça se voit un peu mieux que l'italique.

[2] Il me manque un jeu de termes précis en français pour ces deux-là : le second est réellement un ensemble d'informations lumineuses, tel qu'un ensemble de fréquence de longueurs d'onde avec leur amplitude respective, ou plus simplement un triplet RGB, voire HSV ou même uniquement une intensité de gris, tandis que le premier est une caractéristique intrinsèque de l'interaction de la matière avec la lumière.

[3] Faut-il aussi séparer ces deux notions ? J'ai un gros doute. En effet, si on dédouble encore cette information, pourquoi ne pas aussi permettre un pigment par source lumineuse, plutôt qu'une globalisation pour les lumières diffuses ? Je trouve qu'il y a une limite aux trucages, et qu'il n'est pas nécessaire de dédoubler cette information. Si on veut vraiment faire un pigment différent en fonction de la source lumineuse, il suffira d'utiliser une matrice comme pigment, et des sources lumineuses ayant des vecteurs orthogonaux (et un rien de courage pour coder les deux objets Color et RayColor correspondant, mais au moins ça sera simple à mettre en place)

[4] ou réduction, tout est dans le coefficient multiplicateur.

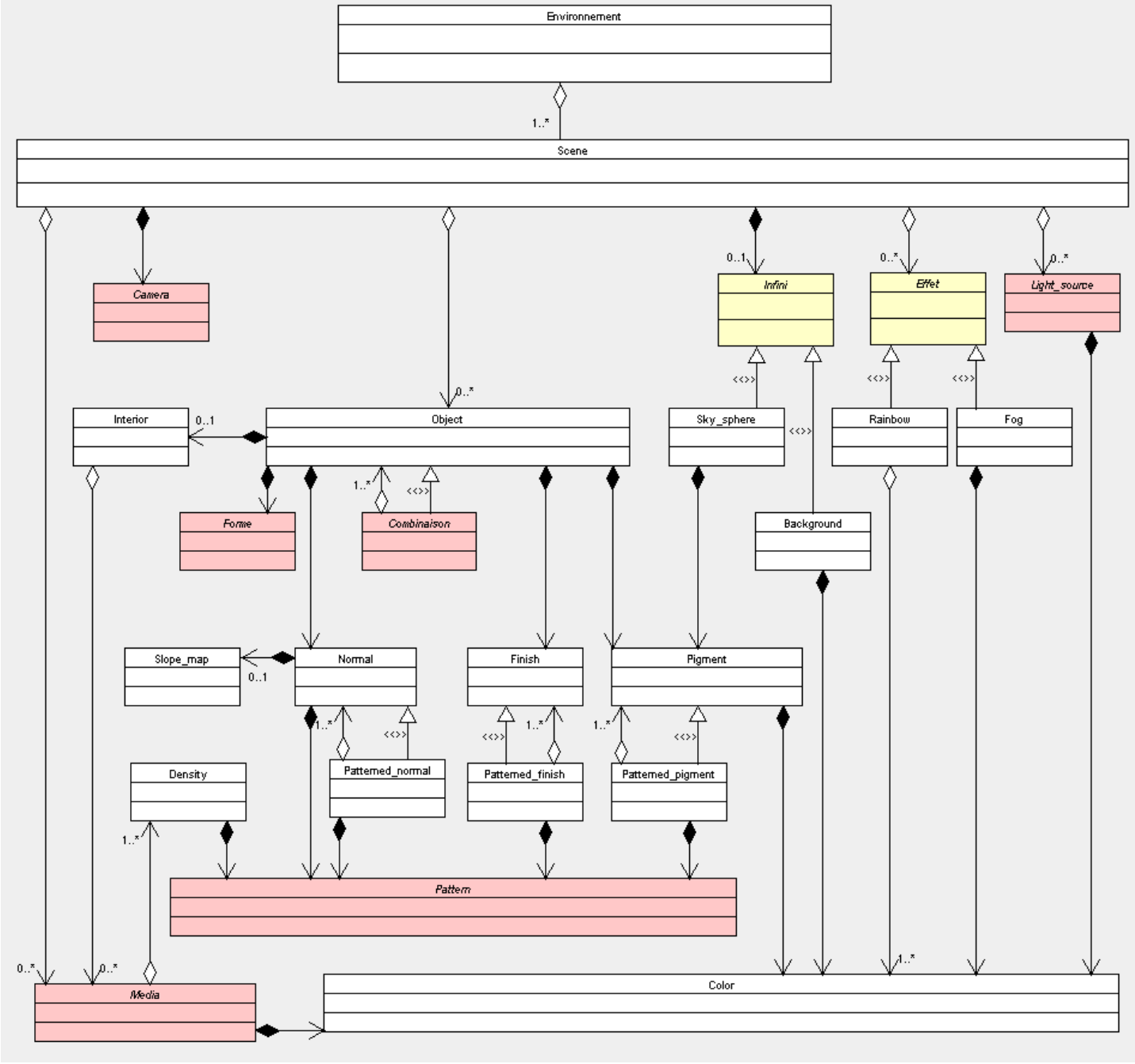

FIG. 4.1 – Modèle de donnée de Crayon

est un concept un peu trop vaste, mais le moins restrictif après le spectre.

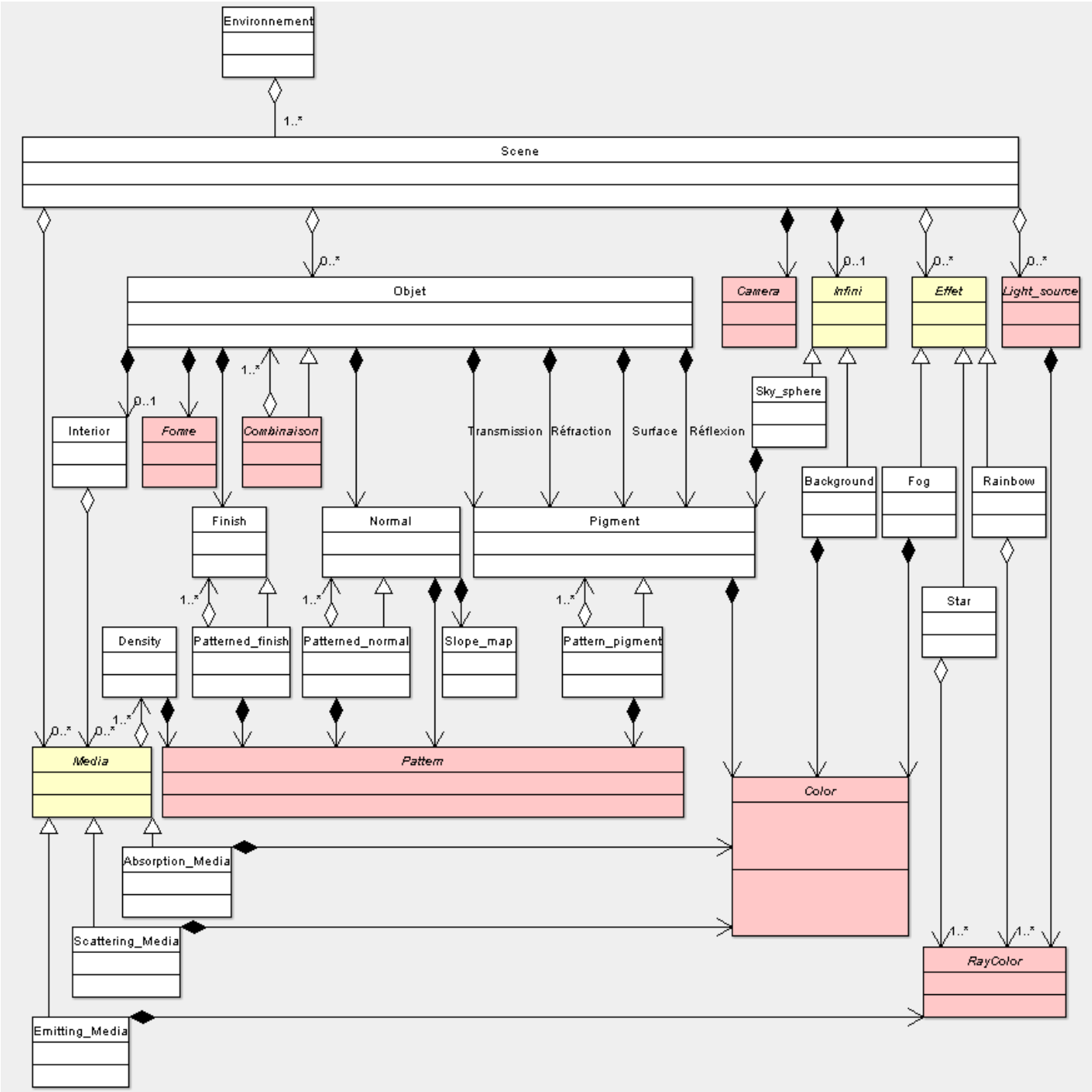

FIG. 4.2 – Modèle de donnée raffiné

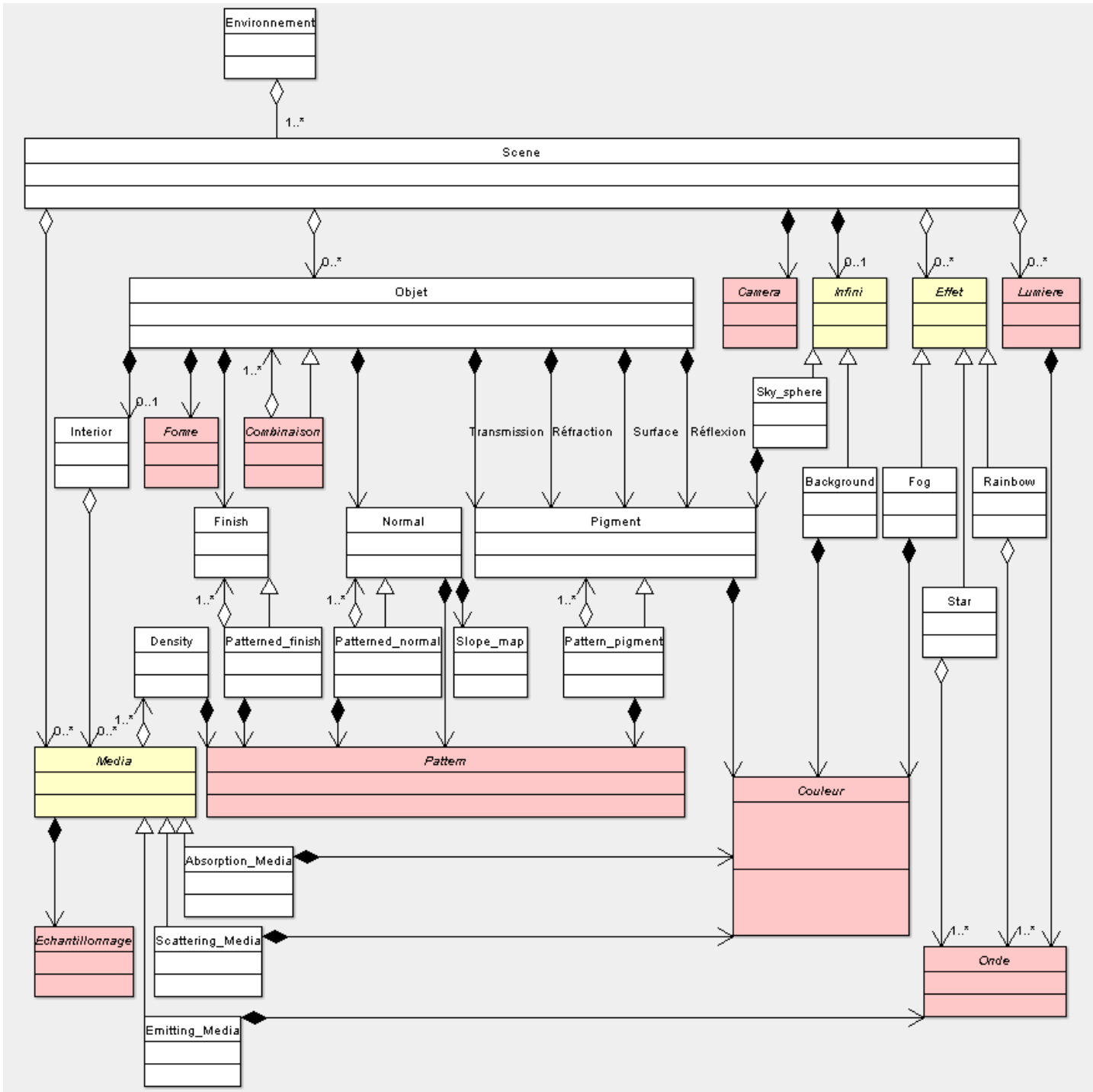

FIG. 4.3 – Modèle de donnée étendu

Chapitre 5

Rétroanalyse des options de POV-Ray

Ca va être un peu long, mais c'est certainement indispensable.

5.1 Using Window version of POV-Ray

5.1.1 I/O Restriction

> L'idée de base des I/O restrictions est d'essayer de protéger l'utilisateur des scènes qu'il a pu télécharger d'endroits divers et qui pourraient chercher à créer ou à modifier des fichiers qu'ils ne devraient pas.

Il y a différents niveaux (exclusifs) possibles d'interdiction :

- rien
- l'écriture seule
- la lecture et l'écriture

Il existe deux listes des répertoires autorisés qui doivent être défini dans le fichier pvengine.ini de l'installation :

- une liste des répertoires autorisés en lecture (Input)
- une liste des répertoires autorisés en écriture (Output)

Il n'y a pas que les répertoires cités qui sont autorisés, mais également leur sous-répertoires.

Il y a également une option pour autoriser le répertoire de la scène, mais sans autoriser pour autant ses sous-répertoires.

Il est aussi possible d'interdire le démarrage de programmes exterieurs (avant et après les rendus).

La question qui se pose, en quoi interdire la lecture est-il importante ?

Je serais assez partisan d'une approche n'autorisant l'écriture que dans un seul répertoire (sans sous-répertoire). Et je ne vois pas de raison d'autoriser la désactivation de cette restriction.

Concernant le lancement d'autres programmes, je doute de l'utilité de conserver cette capacité : c'était utile lorsque le langage était faible, mais si les scènes ont vraiment besoin de faire ce genre de manipulation, je pense que Crayon ne doit pas devenir le coeur du système en cherchant à être un ordonnanceur, mais simplement un moteur de rendu d'image. Conserver cette capacité est aussi en opposition latente avec l'objectif de portabilité : les programmes à lancer seront certainement compilés (sinon, on pourrait les avoir dans le langage des scènes), donc forcement non-portable (à moins que l'une des actions soient de compiler les programmes, mais alors on rentre dans une usine à gaz en terme d'agencement de la scène : autant que Crayon ne soit qu'une feuille dans l'arbre d'appels des programmes, et que l'utilisateur utilise la mécanique qu'il souhaite pour enchaîner les rendus et les générations de scènes.

> Certes il n'y a pas 36 styles d'OS qui permettent d'utiliser ça efficacement, et les plus répandus pour l'instant en sont pratiquement incapables. Mais si l'on fait une animation, on peut avoir besoin de générer un include à chaque image. On peut imaginer sortir ces appels du .ini, mais c'est au prix de devoir gérer la boucle d'anim en externe. A voir... et à compléter par quelqu'un qui fait vraiment des anims ;-)[1]

Après tout, il est possible de conserver la capacité de lancement des programmes externes. Faut-il alors avoir une sécurité permettant d'interdire de lancer des programmes externes ? La question que je me pose, c'est la possibilité d'une exploitation d'une faille du système, mais si cette faille existe, est-ce qu'une simple interdiction suffirait à s'en protéger ? Je crois que le plus simple consistera sans doute à conserver la mécanique utilisant system(), mais également à l'interdire en présence d'une option de la ligne de commande.

[1] François Dispot, 14 janvier 2005

5.1.2 Utilisation d'un répertoire spécifique pour les images rendues

Intéressante alternative, source sans fin de confusion. Il faudrait trancher définitivement pour l'un ou l'autre comportement :

– l'image rendue est mise systématiquement dans un répertoire donné
– l'image rendue est mise dans le même répertoire que son source

Le premier a les inconvénients :

1. d'écraser des images dont les sources portent le même nom
2. l'utilisateur oublie généralement où est ce répertoire
3. l'absence de sous-répertoire (ce qui redonne le premier inconvénient avec d'autant plus de facilité)

Le second a les inconvénients :

1. de parsemer les rendus dans l'arborescences des sources (y compris dans le cas des animations, où un fichier source se retrouve noyé parmi des milliers d'images rendues.
2. de devoir autoriser le répertoire de la source en écriture (donc d'interdire de rendre un CD de scène sans les recopier sur le disque dur par exemple)

L'idée du répertoire dédié me plait le plus, même si j'utilise POV-Ray dans l'autre configuration habituellement, à condition de pouvoir rajouter la notion de sous-répertoire spécifiable par l'utilisateur ou la scène.

Par contre, il pourrait être utile de pouvoir modifier le réglage de ce répertoire sans devoir réinstaller.

> Question de goût... l'idée d'un répertoire fixe m'horripilerait profondément... à moins qu'il soit fixe mais défini pour chaque image.[2]

A la réflection, je crois que je me laisserai aller à finalement définitivement choisir le répertoire courant plutôt qu'un répertoire fixe, ça s'implifiera d'autant la spécification du répertoire puisqu'il n'y aura plus de spécification de répertoire ! Ce qui me gênait principalement, c'était la perte d'espace disque à cause de l'arborescence de fichiers rendus et jamais nettoyé.

5.1.3 Options de la ligne de commande

Il y a pas moins de quatre endroits où l'on peut mettre les options :

1. dans POV-Ray.INI du répertoire RENDERER
2. dans POV-Ray.INI du répertoire courant (de la scène ?)
3. dans un fichier *.INI spécifique à la scène
4. sur la ligne de commande

Et bien entendu, chaque endroit est capable de supplanter ces prédécesseurs dans le réglage des options.

5.1.3.1 Special Windows

Il y a des spécialités pour Windows disponibles sur la ligne de commande :

/**DEMO** effectue un rendu de démonstration.

/**EXIT** effectue le rendu demandé puis arrête le programme (alors que en son absence, le programme reste à la fin du rendu).

/**NORESTORE (ou /NR)** empêche la restoration des sessions d'éditions précédentes.

/**RENDER** effectue le rendu

/**EDIT** ouvre les fichiers dans l'éditeur

Je ne vois aucune raison de les conserver, puisque Crayon n'est qu'un moteur de rendu, et non plus un éditeur intégrant un moteur de rendu.

5.1.4 Fenêtres

De l'ensemble de l'interface, il ne devrait, au mieux resté qu'une seule chose : la fenêtre de rendu. Et encore, je trouve que ça consomme beaucoup de ressources.

> La fenêtre de rendu a un gros avantage : elle permet de suivre le rendu. Et un gros inconvénient : elle est liée au process de rendu. Pour moi ce serait un gros plus de pouvoir la fermer tranquilement, me déloguer et revenir plus tard en pouvant la rouvrir. J'avais proposé ça une fois à l'époque ou povray.unix était encore fréquentable car non envahis par des allemands de la pov team. Un finlandais que je ne nommerai pas a trouvé ça sacrilège. Il doit avoir raison, il est dans la TAG. Aurai-je plus de succès cette fois ? [3]

L'idée est intéressante, surtout dans le cadre d'un portage UNIX : implémenté Crayon sous une forme similaire à un démon qui travaille même sans personne de connecter à la machine.

Le coup de pouvoir rouvrir la fenêtre est aussi intéressante, mais souffre dans l'état actuel d'une spécification implicite

[2] François Dispot, 14 janvier 2005

[3] François Dispot, 14 janvier 2005

trop réduite : Quel est l'adresse de l'affichage à utiliser ? La réponse implicite actuelle est bien évidement l'écran local ! Mais dans un cadre UNIX (et non plus limité à un environnement Linux à la maison), l'adresse de l'affichage pourrait bien être différente. En outre, il est possible que plus d'un affichage veuille avoir accès à l'image en cours de rendu, ça ne devrait pas poser de difficulter.

Il faut, je pense, séparer la gestion de Crayon en cours de rendu de l'existence d'une ou plusieurs fenêtres d'affichage. La gestion de Crayon nécessite certainement une spécification de son interface plus complète et plus souple que la simple lecture du « clavier ». La gestion de Crayon pourrait par exemple déclencher l'ouverture d'une fenêtre de rendu.

Tout ce coté à un aspect terriblement proche du système et nécessitera certainement d'avoir un thread dédié pour la gestion de Crayon.

Peut-être faudra-t-il utiliser un protocole comme UDP ou TCP pour la gestion de Crayon : il faudrait que le protocole soit indépendant du système, c'est pour ça qu'une interface de gestion basé sur des signaux ne me plaît guère et en outre dans un processus multithread, le traitement des signaux n'est pas déterministe dans sa séléction du thread à interrompre. Il faudrait que le programme pour la gestion soit le plus simple possible, voire même si il pouvait être fournit en standard avec la plupart des systèmes.

Stricto-sensus, si on utilise un port UDP ou TCP pour la gestion de Crayon, il faudrait se faire affecter le port par l'IANA, mais je ne me sens pas de recommencer cette manoeuvre.

J'ai tendance à avoir une préférence pour UDP, mais c'est juste parce que ça évite de devoir maintenir et gérer des connexions multiples.

5.1.5 Menus

C'est encore plus simple, il n'y en a plus !

5.1.6 Rapport d'anomalies

De très bonnes choses dans cette section (Bug Report), et j'aurais même tendance à pousser les minima pour avoir obligatoirement une scène illustrant le problème. Cette scène qui incorporera dès lors l'ensemble des tests de non-régressions, donc prévenir le candidat à la soumission d'erreurs qu'il doit céder définitivement ses droits sur la scène en question.[4]

L'architecture pour recevoir les rapports d'anomalies reste à concevoir (et à réaliser) !

[4] Ce qui met le score en Conformité de Crayon à un minimum de 1. C'est beaucoup trop, je trouve.

5.1.7 Considérations sur la vitesse

Une note spécial concernant les priorités du moteur de rendu, mais à part ça, il n'y a que de très bonnes informations dans cette section (Speed considerations).

Faut-il laisser l'utilisateur agir sur les priorité des threads au travers de l'interface de crayon ? J'ai quelques doutes sur le sujet.

On peut, et on devrait, par contre, l'autoriser à jouer sur le nombre de thread de rendu, au moins de manière statique (pas de changement en cours de rendu, quoique ça soit relativement facile à faire).

5.2 Introduction and Tutorial

L'introduction qui arrive dans la seconde partie, je ne suis pas sûr que ça soit une super idée.

Le tutorial est gentil, mais j'aimerai bien voir les résultats au fur et mesure. En fait, j'aimerai bien voir une illustration pour chaque objet. La classification des objets entre 2.2 et 2.3 me semble être principalement due à l'historique de POV-Ray, une refonte serait certainement bien venue.

En fait, une refonte lourde en utilisant une approche comme Information Mapping est vraisemblablement nécessaire. Il y a beaucoup de choses dans la documentation de POV-Ray, mais il faut la connaître pour s'y retrouver !

5.3 POV-Ray Reference

L'organisation me semble meilleure, grâce à la classification.

Chapitre 6

Quels autres patches retenir ?

Un autre tour d'horizon indispensable. J'espère n'avoir oublié personne.

6.1 clothray

http ://tofbouf.free.fr/clothray/index.html

C'est une extension pour simuler des tissus. Elle ne supporte que des morceaux rectangulaires et utilise une simulation à base de connexions maillées implicitement pour alléger le volume des données et l'algorithmie. En sortie, on obtient un fichier de mesh prêt à être inclus.

Il s'agit d'un ensemble de points, chaque point étant relié à ses 24 plus proches voisins par des ressorts. En calculant les forces qui s'exercent sur chaque point à chaque itération (forces externes et internes), on en déduit la nouvelle vitesse et la nouvelle position de chacun des points formant le tissu. Des tests de collision avec l'environnement sont également effectués à chaque itération.

Il y a un problème connu : il n'y a pas de tests de collisions internes lors des itérations, et donc on obtient parfois des morceaux de tissus qui s'interpénétrent.

Petit détails : les objets interagissant avec le tissu doivent être regroupé dans une union. Certes ça évite les longs calculs inutiles face à des objets complexes hors de portée, mais je pense que la simplicité serait de pouvoir s'abstenir de fournir l'union des obstacles et que la scène en cours soit alors utilisée.

Il est possible d'avoir des contraintes mais elles sont limités à axes parallèles aux axes de référence. Il est par exemple impossible de spécifier une tringle de guidage qui soit en biais.

6.1.1 Améliorations à prévoir

- un test de collision interne vraiment efficace
- Davantages de libertés pour les contraintes
- Pouvoir utiliser la scène en cours comme ensemble des obstacles en l'absence d'obstacle explicite

6.1.2 En provenance du site Web

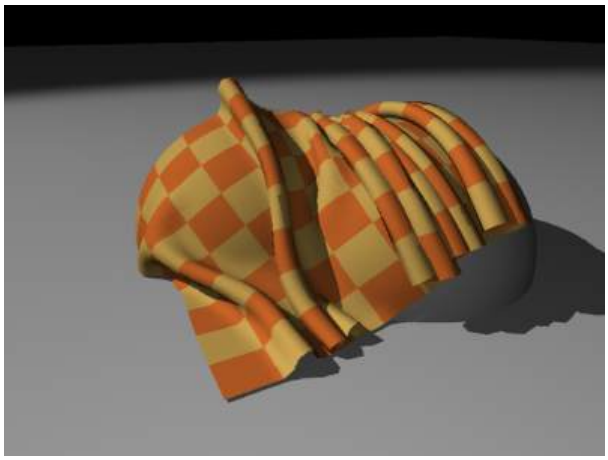

FIG. 6.1 – Illustration de Clothray

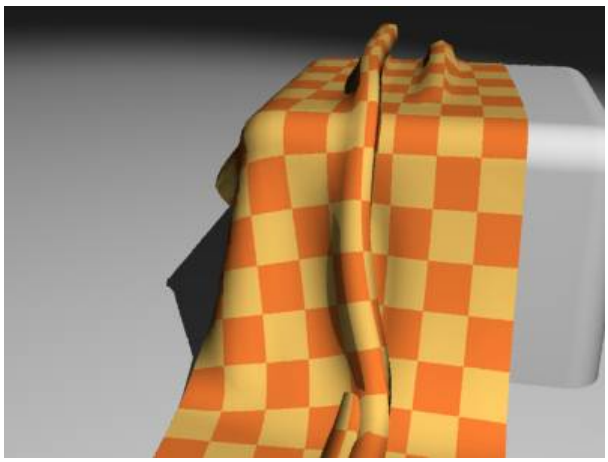

FIG. 6.2 – En l'absence de test de collision interne, il peut y avoir des soucis, ici en bas.

Au niveau du script, cette nouvelle fonctionnalité est implémentée sous la forme d'un nouveau bloc de déclaration :

```
simcloth {
[ environment OBJECT ]
```

[friction FLOAT]
[gravity VECTOR]
[wind { PIGMENT }]
[viscosity FLOAT]
[neighbors 0|1]
[internal_collision on|off]
[damping FLOAT]
[intervals FLOAT]
[iterations INTEGER]
input STRING // seul parametre obligatoire
[output STRING]
[mesh_output STRING]
[smooth_mesh on|off]
[uv_mesh on|off]
}

6.1.2.1 environment

Cela permet de déclarer un objet qui va servir d'environnement. Il est fortement conseillé de le déclarer avant le bloc simcloth (pour des raisons évidentes de lisibilité). Bien que n'importe quel type d'objet puisse etre utilisé, ceux qui disposent d'un intérieur bien défini réagiront bien mieux aux tests de collisions.

6.1.2.2 friction

Il s'agit d'un coefficient de perte au contact de l'environnement. Une valeur nulle signifie que le tissu sera très fortement ralenti, alors qu'une valeur de 1 permettra au tissu de glisser sur les objets (mais pas de rebondir...). La valeur par défaut (si un environnement est défini) est 1.0.

6.1.2.3 gravity

Permet de spécifier la direction de la gravité. La gravité est nulle par défaut.

6.1.2.4 wind

Un pigment est utilisé pour définir la direction et la force du vent en chaque point de l'espace. En un point donné, le vent est défini par les composantes r, g et b du pigment en ce point. On peut utiliser une déclaration explicite de pigment, ou bien un identifiant déclaré auparavant. N'oubliez pas que les composantes d'une couleur peuvent prendre n'importe quelle valeurs (négatives, valeur absolue supérieure à 1, ...)

6.1.2.5 viscosity

Détermine l'influence du vent et des frottements de l'air sur le tissu. La valeur par défaut est 0 (pas d'influence).

6.1.2.6 neighbors

Spécifie le nombre de voisins auxquels chaque point est relié par un ressort. neighbors 0 correspond à 8 voisins (calcul plus rapide, mais plus approximatif), et neighbors 1 (valeur par défaut) correspond à 24 voisins (calcul plus lent, résultat plus réaliste).

6.1.2.7 internal_collision

Active la détection des collisions tissu contre tissu. Ce problème est géré par l'ajout de ressort entre deux points du tissu qui se rapproche l'un de l'autre. Une instabilité du systeme peut apparaitre plus facilement et il faut généralement réduire le paramètre intervals (voir plus bas). Désactive par défaut (off).

6.1.2.8 damping

Coefficent de frottement plus général, il permet essentiellement de limiter l'accumulation des approximations du modèle physique qui pourraient conduire à son instablilité. En bref, un valeur inférieure à .95 (valeur par défaut) est très fortement recommandée.

6.1.2.9 intervals

Représente le delta de temps entre chaque itération. A laisser impérativement faible (enfin, ca dépend d'autres paramètres comme damping, ou le coefficient de raideur du tissu, mais bon....). Par défaut, 0.05 est utilisé.

6.1.2.10 iterations

C'est tout simplement le nombre d'itérations à effectuer (une seule, par défaut).

6.1.2.11 input

Nom du fichier .cth de départ. Seul paramètre obligatoire (ca se comprend...). Pour la description du format de ce fichier, voir plus bas.

6.1.2.12 output

Nom du fichier .cth d'arrivée. Si ce paramètre est spécifié, le résultat de la simulation sera sauvegardé. Utile pour une animation, ou une simulation en plusieurs étapes.

6.1.2.13 mesh_output

Permet de générer un fichier (du nom spécifié) contenant les triangles correspondants aux points du tissu a l'issu de la simulation. Cela permet de sauvegarder le résultat d'une simulation dans un format utilisable par une autre version de POV (qui aurait de nouvelles fonctionnalités).

6.1.2.14 smooth_mesh

Active ou désactive la création de smooth_triangle si l'option mesh_ouput est activée. Désactivée par défaut.

6.1.2.15 uv_mesh

Active ou désactive l'ajout de coordonnées UV lors de la création des triangle (ou smooth_triangle) si l'option mesh_ouput est activée. Les coordonnees UV s'étendent de $< 0, 0 >$ à $< 1, 1 >$. Désactivée par défaut.

6.1.2.16 Description du format de fichier .cth

Le format du fichier est très simple : la première ligne décrit les dimensions de l'étoffe (nombre de points n1 et n2 dans les deux dimensions, longueur nominale entre deux points voisins nlng), et son coefficient de raideur ks. Ensuite, chaque ligne décrit la position dans l'espace et la vitesse de chaque point. Toutes les valeurs (3 pour la position, 3 pour la vitesse) sont séparées par des virgules, et chaque ligne se termine également par une virgule.

Vient ensuite la définition des contraintes sur certains points. Les contraintes sont des coefficients (1 pour chaque axe) qui vont pondérer les coordonnées du vecteur vitesse d'un point donné du tissu. On pourra, par exemple, freiner un coin du tissu suivant un axe, le stopper complètement sur un autre axe, alors qu'il sera totalement libre sur le dernier. Une contrainte est définie par un entier (l'index), et les 3 coefficients représentant la contrainte (respectivement sur les axes x, y, z), tous sur la même ligne et séparés par des virgules. L'index est le rang du point (en commencant à 0) dans l'ordre de déclaration des points. Les contraintes peuvent etre déclarées dans n'importe quel ordre.

Le tissu est en fait représenté par deux tableaux de vecteurs Points[n1][n1] et Vitesse[n1][n1], et ses contraintes Indexn, contn. Le fichier aura donc cette allure :

n1, n2, nlng, ks,
Points[0][0], Vitesse[0][0],
Points[0][1], Vitesse[0][1],
...
Points[0][n2-1], Vitesse[0][n2-1],
Points[1][0], Vitesse[1][0],
Points[1][1], Vitesse[1][1],
...
Points[1][n2-1], Vitesse[1][n2-1],
Points[2][0], Vitesse[2][0],
...
...
Points[n1-1][n2-1], Vitesse[n1-1][n2-1],
Index1, cont1[x], cont1[y], cont1[z],
Index2, cont2[x], cont2[y], cont2[z],
Index3, cont3[x], cont3[y], cont3[z],

6.2 Flou de mouvement

http ://membres.lycos.fr/stalex/motion%20blur.htm

6.2.1 En provenance du site Web

Ce patch n'affecte que les objets. Les caméras et les lumières ne peuvent obtenir d'effet de brouillage en utilisant cette méthode.

6.2.1.1 Introduction

La façon la plus courante de créer ce flou de mouvement dans POV-Ray est de compiler de multiples copies d'une image distribuée sur un temps et un espace donnés, et d'ensuite grouper ces images en faisant leur moyenne. Malheureusement, cela demande beaucoup de temps et en plus d'utiliser d'autres softwares.

Une autre technique serait de créer beaucoup d'objets semi-transparents. Mais cette technique souffre d'une variété de problèmes, tel que pouvoir voir à travers les objets qui ne sont pas rendus correctement.

Nous avons ici une nouvelle solution pour appliquer rapidement ce que nous désirons à un objet. Cette solution est basée sur les suggestion de Ron Parker.

6.2.1.2 Initialiser le flou du mouvement

Pour l'initialiser, ajouter la ligne suivante à votre bloc global_settings :

Motion_blur <sample_count>,<clock_delta>

Sample_count est le nombre d'images temporaires qui seront échantillonnées.

Clock_delta est l'espace de temps pendant lequel se mettent en fonction les unités de la variable clock (cf. l'animation dans la doc). Cet intervalle de temps sera centré autour de la valeur de l'horloge pour l'image actuelle.

Vous ne pouvez pas spécifier de motion_blur dans les global-settings après que vous ayez créer un objet utilisant le flou du mouvement.

6.2.1.3 Créer une objet utilisant le flou du mouvement

Syntaxe :

motion_blur{ <objet>
<modificateurs_d'objet> }

Exemple :

motion_blur{
sphere{ 0,1 material{ mon_matériel}}
translate clock*x }

Note : Le contenu de l'objet motion_blur (tout ce qu'il y a entre les deux accolades) sera parsé plusieurs fois (le nombre de parsing sera égal au nombre d'échantillonnage de temps).

6.2.1.4 Les limitations

- Utiliser le flou du mouvement avec des modifications {} (concernant les scènes persistantes) peut donner des résultats inattendus.
- Un objet utilisant le flou du mouvement contient beaucoup de copies de l'objet brouillé (une pour chaque échantillonnage de temps). Pour cette raison, ajouter cette sorte de flou du mouvement peut utiliser beaucoup de mémoire. Faites attention à cela (rappelez-vous cependant que de multiples copies d'objets mesh n'utilise presque pas de mémoire).
- Le flou du mouvement ne fonctionne peut-être pas bien avec les photons et la radiosité, quoique cela fonctionnait bien au moins dans mes scènes.

6.2.1.5 Comment cela fonctionne ?

Cette technique de flou du mouvement fonctionne en créant de multiples copies de l'objet brouillé. Cela le place dans un objet très similaire à celui de l'union en CSG. (En fait, l'idée d'origine était d'en faire un cas spécial de l'union en CSG).

Chaque objet à l'intérieur de l'objet motion_blur reçoit un temps de repère.

Ce temps identifie à quel échantillonnage de temps l'objet appartient. Les objets immobiles ont un temps de repère égale à zéro. Quand un rayon est tracé, le temps de repère du ray-traceur est initialement remis à zéro. Si un rayon touche un objet motion_blur (qui a un temps de repère non nul), le temps de repère du ray-traceur est fixé à celui de l'objet. A ce point, le ray-traceur intercepte seulement les objets immobiles ou les objets mobiles dont le temps de repère est le même que celui du ray-traceur. Quand le ray-traceur à terminé de tracer le rayon, il vérifie s'il a touché un objet brouillé (il vérifie si son propre temps de repère n'est plus de zéro). Si c'est le cas, il retrace le rayon pour tous les autres temps de repère. Une technique similaire est utilisée pour les tests d'ombre. Notez que ce rééchantillonnage peut survenir à n'importe quel endroit des rayons d'intersection, et pas seulement au niveau de leur racine.

6.2.2 Améliorations et questions à envisager

- L'utilisation de clock est une surcharge qui peut être génante dans le cadre d'animation, ne faudrait-il pas mieux utiliser une autre variable ?
- Est-il nécessaire de dupliquer l'objet en mémoire ? Ne pourrait-on pas fournir une version simplifiée et moins gourmande dans laquelle l'objet de change pas (donc une seule copie mémoire) et où seule la matrice de transformation de l'objet (donc son déplacement) est échantillonée. La simplification ne pourrait-elle aussi s'appliquer dans le calcul des ombres : la couleur d'un rayon d'un objet flou est simplement la moyenne des couleurs des rayons échantillonées, l'objet flou, même opaque, acquérant une transmission partielle en fonction inverse du nombre d'intersections. Il n'est peut-être pas utile d'essayer de synchroniser différents objets flous entre eux.
- Si l'on désire avoir autre chose qu'une transformation simple d'un objet (comme par exemple une sphère bleue se transformant en tore vert ou en cube rouge), il faudra dupliquer effectivement le sous-objet pour chaque échantillon, mais il n'est peut-être pas nécessaire de conserver cette notion de repère de temps.
- Il faudrait, plutôt que d'utiliser un global_setting, pouvoir spécifier le floutage objet flou par objet flou, donc dans les paramêtres de l'objet flou. Il faudrait pouvoir indiquer les valeurs initiale et finale ainsi que le nombre d'échantillons, le bloc de l'objet opaque en mouvement étant effectivement évalué pour chaque échantillon : un peu comme un #while implicite, ce qui implique que l'interface du parseur permettent à n'importe quel objet de marquer une position dans le flux de texte pour pouvoir y revenir par la suite, et aussi qu'un objet puisse modifier une variable scalaire qu'il déclare implicitement.
- Est-ce vraiment bien que le nom de la variable scalaire ne soit pas changeable par l'utilisateur ?
- Ne faudrait-il pas deux variantes de floutage : l'une qui échantillone les objets, et l'autre qui ne prend qu'un objet et échantillonne une simple transformation (enfin, simple, c'est une matrice tout de même).

motion blur. Je n'ai pas utilisé ça depuis un bon

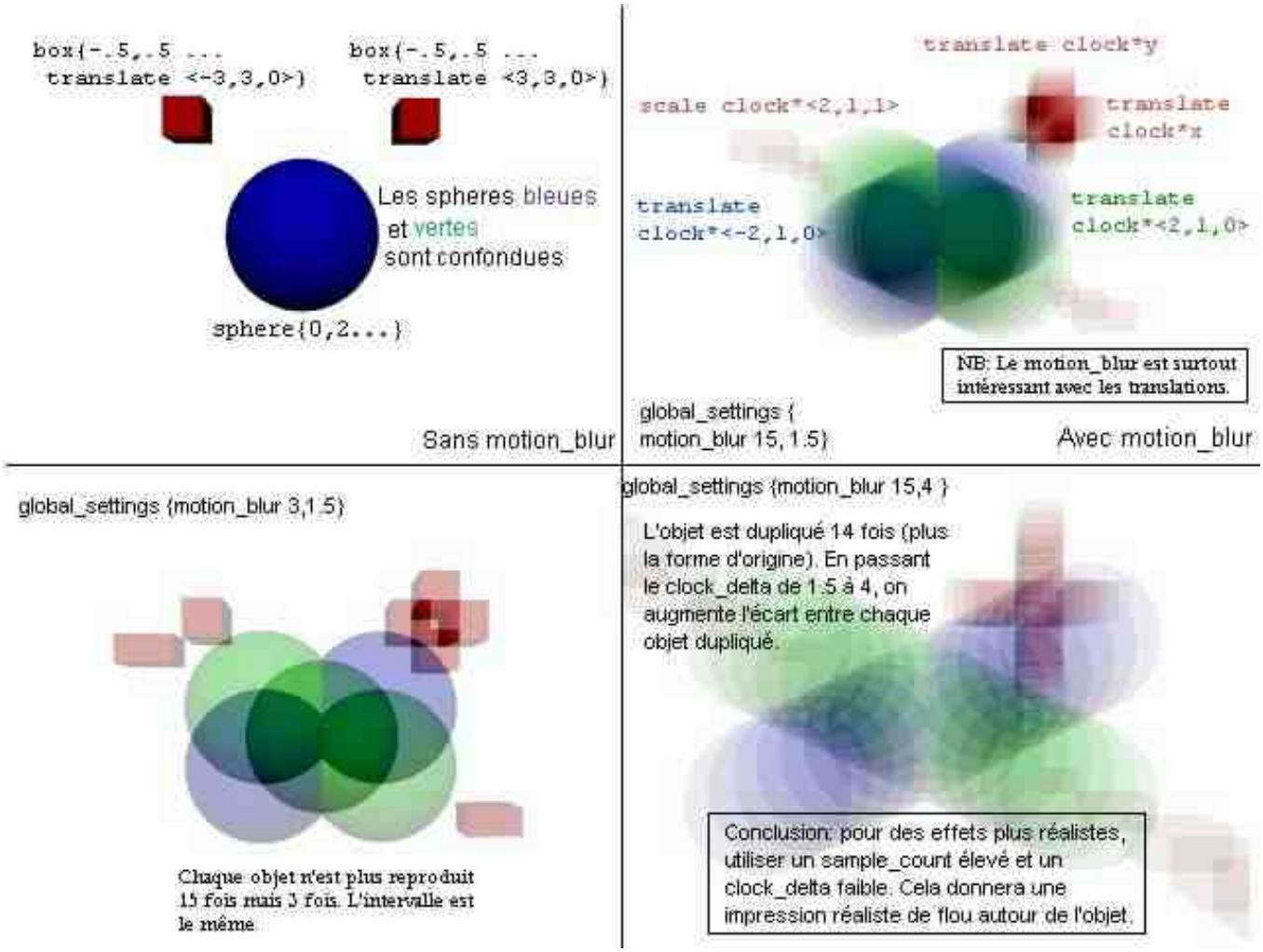

FIG. 6.3 – Flou de mouvements

> moment mais il me semble que ça se passait assez mal quand des lumières se déplaçaient, puisqu'elles étaient recopiées. Si ça marche mieux, tant mieux. J'en avais parlé sur p.u.p mais un finlandais que je ne nommerait pas a répondu qu'il était absurde de vouloir déplacer des lumières. Il doit avoir raison, il est membre de la TAG.[1]

Déplacer les lumières, c'est une idée intéressante, et je reconnais ne pas y avoir penser. Est-ce vraiment absurde ? Je ne sais pas, mais il est certain que ni l'implémentation de Povray, ni celle (ou celles) de Crayon ne seraient compatibles avec cette idée. Il existe cependant un moyen simple, pour un mouvement linéaire uniforme du moins : utiliser une Area_light en forme de ligne.

Y a-t'il une solution pour avoir des lumières en mouvement, surtout si ce mouvement est complexe, et avoir un flou de mouvement ? Sachant qu'une lumière est invisible, elle ne peut pas être floutée. Par contre, les ombres qu'elle projette peuvent-elles être perçue comme floutées ? Est-ce qu'un semis de lumières fixes le long du déplacement ne ferait pas aussi bien ? En fait, il n'y a guère que le cas du « projected_throught » qui puissent éventuellement avoir un effet très particulier. Est-ce qu'un flou de mouvement d'un projecteur de poursuite est vraiment visible ?

En dehors du patineur tenant une lampe torche allumée photographié en pose B, le déplacement des lumières ne m'évoque aucun besoin réel. Pour ce patineur, la torche pourrait être simplement un objet avec un niveau ambient élevé, et non une réelle source lumineuse. Resterait cependant à éventuellement traiter avec un ensemble de lumières coniques le trajet du faisceau, si il y a un réel intéret visible à simuler ce déplacement.

Maintenant, peut-être que l'objet devant être flouté pourrait également effectuer automatiquement la duplication des lumières ?

L'aspect « flou » me fait penser que l'on évolue à des vitesses suffisament lentes et avec une persistence rétinienne ou photographique tel que l'on puisse sortir les lumières

[1] François Dispot, 14 janvier 2005

des mouvements des objets. En particulier, je ne m'attendrai pas à obtenir un résultat correct dans le cas d'une simulation de l'expérience de calcul de la vitesse de la lumière qui utilise une roue dentée en rotation et un miroir aligné avec une source lumineuse[2].

6.3 uvmap

http ://membres.lycos.fr/stalex/uv-mapping.htm

6.3.1 En provenance du site Web

6.3.1.1 Description

Toutes les textures de POV sont définies en 3 dimensions. Même le mapping d'une image plane est fait de cette manière. Cependant, il est parfois plus désirable d'avoir une texture définie selon la surface de l'objet. Cela est spécialement vrai pour les objets "bicubic_patch" et les maillages d'objets, issus de modeleur, et qui peuvent être allongés (étendus) et compressés. Quand l'objet est allongé et compressé, il serait préférable que la texture soit "collée comme de la glue" à la surface des objets et qu'elle suive les déformations de cet objet.

Un nouveau mot-clé a été ajouté aux modificateurs d'objet. Si le modificateur uv_mapping est spécifié, alors la texture de l'objet lui sera appliquée en utilisant ses coordonnées de surface (u et v) au lieu de ses coordonnées spatiales (x, y et z). Cela est fait par une "coupure en tranche" de la texture 3D régulière de l'objet depuis le plan X Y et par l'enveloppement de cette coupure en tranche autours de la surface de l'objet, tout en suivant les coordonnées de surface de l'objet.

- Le mot-clef uv_mapping doit être spécifié avant la texture de l'objet.
- Avec l'uv-mapping, il n'est pas important que la texture soit ajoutée à l'objet avant ou après les transformations de l'objet !

6.3.2 Avis personnel

L'UVmapping, c'est un mélange d'image map et de textures procédurales. Est-ce que ça vaut vraiment la peine de s'embêter avec un truc pareil ?

En plus, ça n'est intéressante que pour des objets comme le bicubic_patch, le lathe, la surface de révolution (sor) et le mesh. Le patch original donnent une liste d'autres candidats, mais en dehors des figures 2D et du tore, c'est relativement inintéressant, puisque une simple projection d'une image donne le même résultat. (D'ailleurs, pour le tore, revoir le warp correspondant, ça fait la même chose si le motif de base est une image).

Le mapping du cube proposé est totalement arbitraire, et celui du lathe/sor est un mapping sphérique, alors qu'il est probable qu'un mapping cylindrique pourrait parfois être plus utile (si ce n'est le problème de gérer alors les extrémités).

Bref, je ne suis pas forcément convaincu par ce patch. Il est passé dans la version 3.6, avec un changement pour le cube, on pourra donc le reprendre, mais en dehors du bicubic_patch et du mesh (et peut-être du parametric, mais c'est pas clair pour l'instant), il ne s'agit que de Warps qui ne disent pas leur noms, et sont codés en dur de manière particulière.

Peut-être faudrait-il rendre la méthode d'UV mapping obligatoire pour les objets ?

J'ai vraiment tendance à penser que l'UV mapping devrait être vu comme un simple pattern (qui utilise une fonction de l'objet pour obtenir des coordonnées, mais en dehors de ça, ça n'a rien de si spécial). Au mieux, ça me conforte dans l'idée que les fonctions d'évaluation de pattern devrait avoir accès à plus d'informations contextuelles, comme :

- le rayon incident (origine, direction et distance)
- la normal à l'intersection
- le point d'intersection
- l'objet à l'intersection (au sens C++, donc disposant de fonctions appellables par le pattern)

Par contre, vu le surcoût de calculer systématiquement des coordonnées UV, je pense qu'il faut vraiment laisser ce genre d'initiative à l'évaluation du pattern.

6.4 persistance

http ://membres.lycos.fr/stalex/persistence.htm

6.4.1 En provenance du site Web

6.4.1.1 La persistance de variables

Vous pouvez faire que les variables persistantes soient utilisées en ajoutant la ligne suivante au fichier INI utilisé pour faire votre scène :

Persistent_Animation=yes

[2] Au lieu de 100% ou 0% du flux lumineux, on obtiendrait un 50% de moyenne, quelque soit les réglages de distances et de vitesses, ainsi que la période de prise de vue.

Par cette option, toutes les variables seront persistantes tout au long d'une animation. Cela signifie qu'elles ne sont pas détruites à la fin du rendu d'une image. Elles sont existantes jusqu'à ce que le rendu soit terminé ou interrompu. Continuer un rendu préalablement arrêté (avec l'option + C dans le fichier INI) ne restaurera pas les variables. Rappelez-vous, vous pouvez dévalider les variables avec #undef si vous voulez les supprimer. Vous pouvez également redéfinir une variable vers un type différent avec MegaPov.

[3]

6.4.1.2 La persistance de scènes

L'option Persistent_Animation permet également la persistance des objets dans une scène.

Habituellement, quand une image vient de finir d'être rendue, POV détruit tous les objets dans la scène. MegaPov vous permet d'ajouter des labels aux objets de vos scènes. Quand l'option Persistent_Animation est employée, tous les objets rendus pour l'image ayant des labels resteront en mémoire à la fin du rendu et ne seront pas détruits.

Donner un label au objets Actuellement, il y a deux manière de donner un label à des objets. Par la suite, une de ces deux manières sera abandonnée au profit de la seconde. Au passage, veuillez signaler à Nathan Kopp celle qui vous plairez le plus...

Méthode 1 Vous pouvez donner un label à un objet en utilisant le mot clef label comme modificateur d'objet. Utilisez-le juste comme si vous vouliez utiliser une translation ou une rotation. Par exemple :

```
union{
sphere{ 0,1 texture{MaTex} label MaSphere}
box{ -1,1 texture{MaTex2} label MonCube}
label Ma_CSG! }
```

Notez qu'aucun guillemet n'est placé autour du label. N'importe quel nom peut être utilisé comme label (même les mots clefs réservés comme sphere ou x). Des objets peuvent partager des labels.

Méthode 2 Utilisez #create juste comme vous utiliseriez #declare. Cela créera un objet en lui donnant un label. Voici un exemple :

```
#create Ma_CSG! = union{
#create MaSphere = sphere {0,1 texture {Ma-
    Tex}}
#create MonCube = box {-1,1 texture {Ma-
    Tex2}} }
```

Il est important de savoir que des objets ayant un label sont complètement différents au niveau syntaxique que des objets #declare ou #local. Vous ne pouvez pas utiliser d'objet label là où POV attend un objet déclaré, et inversement. Aussi, un objet label deviendra seulement effectif quand cet objet fera partie de la scène. S'il a simplement été déclaré et placé sous un label, vous ne pourrez pas utiliser les modificateurs modify{} et delete{} décrits ci-dessous, jusqu'à ce qu'il soit placé dans la scène.

Modifier un objet comportant un label Une fois que vous avez donné un label à un objet, il persistera tout au long de l'animation. Une fois une image rendue, l'objet ne sera pas supprimé avant que l'image suivante soit parsée.

Il se peut que vous vouliez modifier un objet persistant. Pour cela utilisez le mot clef modify. Voici un exemple :

```
modify{ Ma_CSG! translate 10*x }
```

Si de multiples objets partagent ce label, le modificateur d'objet sera appliqué à chacun d'eux.

[4]

Vous pouvez également supprimer un objet ayant un label :

```
delete {Ma_CSG!}
```

Finalement, vous pouvez modifier ou supprimer un objet ayant un label et se trouvant à l'intérieur d'un objet CSG ayant aussi un label. Pour cela, utilisez un point pour séparer les noms (comme la programmation d'objets orientés). Voici comment :

```
modify { Ma_CSG!.MonCube
texture{ MaTex3) // application d'une nouvelle
    texture
}
```

Dans le futur, nous aurons des modificateurs qui nous permettront d'ajouter un nouvel enfant à un objet CSG et de remplacer un objet par un autre.

[3] Il n'y a actuellement pas de ligne de commande qui corresponde à cette option INI.

[4] le code POV pour les modificateurs d'objet sera re-parsé pour chaque objet auquel il s'applique. Pour cette raison, si vous utilisez les nombres aléatoires dans le block modify{}, chaque objet modifié le sera différemment. Pensez à cela, comme une boucle #while qui serait automatiquement créée et se répéterait pour chaque objet ayant le même nom.

6.4.2 Avis personnel

La persistance des variables n'est utile que dans le cas de calcul itératif, afin de gagner du temps : si pour calculer les images d'une animation, il faut juste une seconde depuis les informations de l'image précédente, ça devient pénible lorsque à chaque image on repart de l'image 0 (puisqu'alors le temps de lecture de la scène croit de manière réguliere plus on s'éloigne de l'image initiale).

Je ne vois pas d'autres intérêts à la persistance des variables, et si on autorise l'écriture et la lecture de fichiers, il est tout à fait possible de se passer d'une implémentation interne de la persistance : la scène sauve les valeurs qui l'intéressent dans un fichier et les rechargent par la suite (et en l'absence de fichier, effectue les itérations pour obtenir les valeurs à utiliser).

En dehors de fournir une interface simplifiée pour la sauvegarde de variables, tableaux et autres, il n'y a pas vraiment d'intérêt à avoir une persistance des variables, j'y verrai même un inconvéninent.

L'idée de label est intéressante, surtout si l'on désire fournir un accès en lecture d'un objet, car il se pose toujours la question de nommé l'objet auquel on désire accéder. Or les objets affectés à une variable ne font pas partie de la scène, et les objets faisant partie de la scène n'ont pas de nom de variable actuellement.

On devrait certainement reprendre la méthode 1 de déclaration de label, la méthode 2 ayant, je trouve, tendance à introduire une lourdeur syntaxique, surtout dans le cas de labels imbriqués. Il faut envisager qu'un label puisse se retrouver affecté aussi bien à une CSG qu'à une couleur ou un autre sous-élément de la scène.

Faut-il pour autant envisager la persistance des objets ? Je ne le crois pas, surtout si il est possible de demander à un objet portant un label de s'auto-décrire de manière normalisée dans un fichier. Il devient alors simple de procéder au rechargement d'un objet.

Concernant la modification d'un objet, j'aurai tendance à favoriser une approche simpliste basé sur la déclaration d'une variable comportant l'objet initial (avec un label) et à avoir le code de la scène qui effectue une re-création de l'objet en utilisant l'accès en lecture à la variable comme modèle.

6.5 radiance

http ://membres.lycos.fr/stalex/radiosite.htm

6.5.1 En provenance du site Web

6.5.1.1 Introduction

S'il y a bien une partie de POV-Ray qui engendre de nombreuses difficultés, c'est la radiosité. Mais lorsqu'elle est utilisée correctement, vous pouvez normalement produire de très belles images. Malheureusement, il y a quelques bogues et limitations à l'utilisation de la radiosité dans la version officielle de POV-Ray 3.1.

Premièrement, remarquons, comme le signale le manuel POV-Ray, que POV-Ray n'utilise pas de radiosité "réelle". Une vraie radiosité nécessiterait de convertir la scène entière en polygones, et de calculer une solution de lumière par approximation avec une solution d'un nombre très important d'équations linéaires.

La solution de POV-Ray pour la radiosité est parfois connue sous le nom de ray-tracing monte-carlo. Quand un rayon lancé touche un objet et que POV veut savoir combien de lumières indirectes sont en train de toucher sa surface, il envoie un groupe de rayons aléatoires dans la scène et "rassemble" la lumière des autres objets.

Manifestement, il serait vraiment très difficile de faire cela pour chaque intersection. Une supposition qui peut généralement être faite au sujet de la lumière diffuse indirecte est qu'elle ne change pas très rapidement. Alors une bonne approximation serait de lancer beaucoup de rayons depuis un faible pourcentage (entre 1 et 5 %) d'intersections et d'ensuite les interpoler entre ces valeurs.

6.5.1.2 Changements

La radiosité est orthogonale aux éléments ambiants d'une surface. Dans le tutoriel sur la répartition de la radiosité, la première règle est :

> Choisissez votre lumière ambiante avec attention.

L'explication sur la répartition démontre comment, dans POV 3.1, la brillance de la radiosité est très affectée par les réglages de la lumière ambiante.

Alors qu'est-ce qu'orthogonal signifie ? Cela signifie juste que ces deux caractéristiques sont indépendantes et s'affectent l'une l'autre dans des cas bien précis. Ce n'est pas le cas dans la version 3.1 de POV-Ray.

La brillance de la radiosité dans UVPov (ou MegaPov) est basée sur deux choses :

1. la somme de lumière 'rassemblée', et
2. les propriétés de la diffusion dans la finition de la surface.

Un objet peut avoir à la fois de la radiosité et des termes d'ambiance. Cependant, il est suggéré, si vous utilisez la radiosité, de fixer la lumière ambiante (ambient_light) à 0 dans les déclarations globales (global_settings), ou d'utiliser ambient 0 dans la finition de chaque objet. Ce modèle de lumière est bien plus réaliste, et POV n'essayera pas d'ajuster la brillance totale de la radiosité pour qu'elle s'assortisse avec la lumière ambiante spécifiée par l'utilisateur. Pour utiliser une radiosité exagérée, augmentez la brillance (brightness) dans la section de la radiosité dans les déclarations globales, au lieu d'augmenter la lumière ambiante.

nearest_count est un minimum, pas un maximum
Premièrement, dans le but d'obtenir une bonne interpolation, vous devriez obtenir un groupe de point et les interpoler. POV-Ray vous permet de spécifier un nearest_count qui, selon la documentation, est "le nombre maximum d'anciennes valeurs ambiantes mélangées ensemble". Malheureusement, cela signifie que vous devrez parfois mélanger une seule valeur. Faire la moyenne d'une seule valeur peut mener à des mauvais approximations.

Par conséquent, dans UVPov, utilisez nearest_count pour spécifier le nombre minimum d'anciennes valeurs mélangées ensemble. Le nombre total mélangé variera en dépendant de error_bound (marge d'erreur). Toutes les valeurs précédentes se trouvant spécifiée dans cette marge d'erreur seront utilisées dans la moyenne.

La distance maximum est automatiquement calculée. L'explication de la distance maximum dans la documentation ne fut jamais évidente à comprendre. En regardant dans le code source, Nathan Kopp remarqua que si vous ne spécifiez pas de distance maximum, POV l'estimera en se basant sur la distance comprise entre la caméra et la première intersection. Il a étendu ce concept et composa une distance maximum (mot clef distance_maximum) pour chaque intersection, basée sur la distance depuis cette dernière jusqu'à la caméra. Cela fonctionnait d'ailleurs très bien dans tous les tests effectués par la suite. Si vous spécifiez une distance maximum manuellement, elle sera ignorée.

La radiosité fonctionne avec les pièces de toutes dimensions La seconde règle du tutorial sur la radiosité est :

> Ne faites pas vos pièces trop grandes.

La raison pour laquelle toutes les pièces de toutes dimensions fonctionneront avec MegaPov est que cela est dû à ce calcul automatique de la distance maximum, mais aussi à la réparation du gros bogue relatif à la marge d'erreur (error_bound) et au premier passage du calcul de la radiosité. Ensemble, ces travaux parent la plupart des effets de taches dont souffrait la radiosité dans le passé.

La limite de récursivité de la radiosité n'est pas limitée à 2. Vous pouvez spécifier toutes les limites de récursivités que vous souhaitez (après 5 ou 6, cela devient inutile, quoique.... d'accord, sauf pour les scènes originales (et incroyable)).

La radiosité est calculée selon les rayons réfléchis et les rayons réfractés/transmis. Si vous avez un objet transparent, vous obtiendrez la radiosité sur sa face éloignée. Si vous avez un miroir, l'image réfléchie aura de la radiosité.

Changements divers et fixation des bogues. Vous pouvez spécifier un adc_bailout pour les rayons utilisants la radiosité. Pour des résultats meilleurs, utilisez adc_bailout = 0.01/ ambiance de l'objet le plus brillant.

Fixation d'un bogue relatif à la marge d'erreur (error_bound) et l'échantillonnage durant le premier passage de la radiosité.

L'estimation de la radiosité est affectée par la surface normale.

Astuces Vous avez dû remarquer l'utilisation de :

ini option "Preview_Start_Size=8"

ini option "Previous_End_Size=4"

dans les précédents réglages de hautes qualités. Bien que ce ne soit pas totalement nécessaire, mettre par défaut la taille de départ et la taille de fin à 8 peut causer des effets de taches. Ceci est dû aux prises d'extra échantillonnages durant le rendu final de la radiosité. Ces nouveaux échantillonnages peuvent causer des discontinuités dans la radiosité de certaines scènes. Pour réduire ces artefacts, utilisez un previous_end_size de 4 (ou même 2 si vous êtes réellement patient). Cela fera que la majeure partie des échantillonnages sera prise durant les premiers rendus (de prévisualisation), et réduira les artefacts créés durant le rendu final.

Si votre scène utilise des objets ambiants (spécialement les petits objets) comme sources de lumière, vous devrez probablement utiliser un compte supérieur (count) (environ 100 est déjà très bien, mais pour le rendu final, utilisez 150). Pour la plupart des scènes, la marge d'erreur de 1.0 (error_bound) est très bien. L'augmenter peut causer trop d'erreurs, mais la baisser rend la compilation vraiment très lente.

Les précédents réglages de qualité moyenne sont normalement bons pour les scènes comprenant un taux moyen

d'illumination. Cela permet un bon rapport "rapidité / qualité d'image ".

Si les choses paraissent tachées ou comportent des petites traces, augmentez le count et le nearest_count.

6.5.2 Avis personnel

Je me demande si il y a encore une différence entre cette radiance et celle de POV-Ray 3.6.

En fait, le seul autre aspect qui me gêne avec la radiance, c'est sa parallélisation, vu qu'elle utilise beaucoup de rayons (dans la phase de prévisualisation) afin de peupler sa structure de donnée (donc de la remplir, ce qui est contraire au prédicat que lors du rendu, les données sont en lectures seules). Il faudrait vraiment pouvoir paralléliser ce morceau.

6.6 photons

http ://membres.lycos.fr/stalex/photons.htm

6.6.1 En provenance du site Web

6.6.1.1 Limitations actuelles

1. Les photons ne fonctionnent pas correctement avec les lumières cylindriques (les faisceaux de lumière ne sont pas retenus à l'intérieur du cylindre).
2. Les photons ignorent actuellement les modificateurs "parallel" des lumières propres au Superpatch.
3. Les photons ignorent actuellement les modificateurs "circular" et "orient" des aera_lights propres au Multipatch.

6.6.2 Avis personnel

En dehors d'en apprendre un peu plus sur les limitations des photons, il semble bien que POV-Ray 3.6 intègre déjà cette évolution.

Il faudrait certainement corrigé les limitations actuelles, ça ne devrait pas être si difficile que ça à faire.

Parallel me semble réellement faire double-emploi avec les lumières cylindriques de rayon immense. Ou alors c'est les lumières cylindriques qui ont besoin d'être revue : pour moi, ça devrait être comme un faisceau laser... Le rayon doit-il être oblique ? je ne crois pas.

6.7 autres

http ://membres.lycos.fr/stalex/misc.htm

Il n'y a rien que je ne désire retenir, même les spécifications de taille de l'image me semble une très mauvaise idée.

6.8 pattern

http ://membres.lycos.fr/stalex/patterns.htm

6.8.1 En provenance du site Web

6.8.1.1 Les motifs fractals

Les motifs fractals furent créés par Nieminen Juha.

Le patch fractal a actuellement les nouveaux types de motif suivants :

- mandel3, mandel4 : Comme mandel, mais la formule est z(n+1) = z(n)^p+c où 'p' est respectivement 3 et 4.
- julia, julia3, julia4 : type de julia, où 'p' est respectivement 2,3 et 4.
- magnet1m, magnet2m : fractales dérivées des transformations des renormalisations magnétiques (voir l'aide de fractint pour plus de détails).
- magnet1j, magnet2j : la version julia de ces mêmes fractales.

La syntaxe pour les motifs de type mandel est la même que celle pour les motifs réguliers :

TYPE_DE_MOTIF ITERATIONS

La syntaxe pour les motifs de type julia est :

TYPE_DE_MOTIF COORDONNEES ITERATIONS

COORDONNEES est un vecteur 2 dimensions (en relation avec un nombre complexe) à partir duquel la forme est calculée. Par exemple : julia<.353,.288>30

De plus, ce patch a deux nouveaux mots-clef qui fonctionnent avec tous les types de fractales (mandel inclu) :

fractal_interior_type TYPE, FACTEUR
fractal_exterior_type TYPE, FACTEUR

Cela fixe la couleur de l'intérieur et de l'extérieur des fractales. Actuellement la valeur du TYPE peut-être intégré entre 0 et 6.

Toutes les valeurs retournées (excepté pour le type 2 de l'extérieur) sont multipliées par le FACTEUR avant d'être retournées.

Ces types sont les suivants :

0 Retourne juste 1 (multiplié par le facteur).

1 Pour l'extérieur : le nombre d'itérations jusqu'à ce qu'elles soient suffisantes divisé par ITERATIONS. Pour l'intérieur : la valeur absolue du plus petit point de l'orbite par le point point calculé.

2 Partie réelles du dernier point dans l'orbite.

3 Partie imaginaire du dernier point dans l'orbite.

4 Partie réelle au carré du dernier point dans l'orbite.

5 Partie imaginaire au carré du dernier point dans l'orbite.

6 Valeur absolue du dernier point dans l'orbite.

Quand cela n'est pas spécifié, la valeur par défaut pour fractal_exterior_type est 1, pour fractal_interior_type c'est 0 et pour FACTOR, c'est 1.

6.8.1.2 Reset_Children

Tout d'abord merci à Daren Scot Wilson. Son warp_map_only a inspiré cette caractéristique. Malheureusement son patch a des bugs que cette version essaie de corriger.

En utilisant des spectres de texture, ou n'importe quelles textures qui ont des sous-textures, vous pouvez rencontrer des problèmes lorsque vous voulez modifier ces spectres de texture (comme avec de la turbulence), mais que vous ne voulez pas modifier les sous-textures.

La solution est un nouvel enveloppement, appelé reset_children. Quand vous désirez ajouter cet enveloppement à une texture, les sous-textures sont remises à normal. Tous les enveloppements et tranformations suivants seront appliqués à la texture, mais tout ce qui est avant la déclaration reset_children est supprimé pour les sous-texture. Voici une scène faites avec la déclaration reset_children :

Pour les intéressés, la mise en place du reset_children s'est faite en deux temps. Premièrement, Nathan Kopp à ajouté un nouveau type de warp (d'enveloppement) appelé reset_children. Ensuite, il a ajouté un paramètre à Warp_EPoint pour que l'on puisse savoir s'il était en train d'envelopper les textures parents ou les sous-textures. Si Warp_EPoint est en train d'appliquer à l'objet une sous-texture et que le ray-traceur touche un warp reset_children, il se stoppera. Puisque les warps sont exécutés en ordre inverse, cela signifie que seuls les warp après le warp reset_children sont utilisés, ce qui apparaît à l'utilisateur comme la remise à zéro des sous-textures.

6.8.2 avis personnel

Les motifs sont déjà repris dans POV-Ray 3.6 (et généralisé, puisque la puissance va jusqu'à 33), cependant je trouve que :

1. Les types devraient être homogénéïsés et nommés, au lieu d'être numéroté
2. le type 1 devrait être séparé en deux types distincts

Concernant reset_children, c'est à la limite intéressant, je me demande pourquoi ce n'a pas été repris par POV-Ray 3.6, en dehors de l'utilisation probablement anecdotique qui doit en être fait.

6.9 pvmpov

http ://www.geocities.com/CapeCanaveral/Lab/6386/pvmpov

Une version faite avec POV-Ray 3.0.

6.9.1 Problèmes connus

- la radiance ne marche pas correctement
- l'option UO pour les animations (utiliser les lignes impaires dans les images impaires) ne marche pas
- il y a une fuite mémoire. Les esclaves consomment de plus en plus de mémoire au fur et à mesure de l'exécution.
- parfois, des blocs sont mal placés.

6.9.2 Avis personnel

PVM permet la communication entre différents processeurs au sein d'une machine virtuelle, mais sa notion d'atome est le processus. En particulier, il ne duplique pas automatiquement les données d'une machine à l'autre, c'est à l'applicatif de prévoir d'empacter et d'envoyer ses données partagées aux autres processeurs, et sur place de reconstruire les données.

Oserais-je rappeler que le gros problème de POV-Ray/crayon, c'est justement la taille des données à transférer (une fois la scène analysée). On échangerait au mieux le temps de lecture et d'analyse de la scène par le temps de transfert en binaire, mais en l'absence de photon et de radiance, on ne fait guère d'autres choses que des rendus par blocs, avec aggrégation de l'image en automatique.

6.10 pvmegapov

http ://www.wozzeck.net/images/pmp/

Une version faite avec POV-Ray 3.1 et megapov, encore avec PVM pour la parallélisation.

En présence de photons ou de radiance, les choses sont encore plus compliquées (l'émission des photons est faite sur chaque machine, et non parallélisé... ce n'est pas vraiment un gain de temps).

Je crains définitivement que la parallélisation sur des machines distantes ne soient trop lourd pour un gain de plus en plus faible au fur et à mesure que les choses se compléxifient. Il est certainement raisonnable de ne pas prévoir de parallélisme sur des machines distantes dans la conception de Crayon.

6.11 POV-Ray35

Les changements par rapport à POV-Ray 3.1 sont les suivants :

- la fonction de bruit change pour supprimer des plateaux que son ancienne version avait par endroit
- les photons font leur apparition
- on a une simulation de l'effet de dispersion
- la radiance change, mais reste expérimentale
- les sources de lumières gagne parallel, circular et orient
- la notion de groupe de lumière apparaît
- les formes définies par des isosurfaces font leur apparition, ainsi que l'objet parametric
- l'objet sphere_sweep apparaît
- une nouvelle syntaxe pour l'objet mesh est disponible (mesh2), l'ancienne syntaxe continuant d'exister.
- un bricolage essaye de rendre les mesh utilisables avec la CSG, à condition que le mesh soit bien fermé
- UV-mapping fait son entrée, à la mode mégapov.
- des patterns et des warps additionels
- cutaway_textures et interior_textures sont aussi de la partie
- l'interpolation de texture pour les triangles fait son apparition
- conserve_energy et fresnel apparaissent, ainsi qu'un bloc reflection dans finish, tout comme le contrôle de la reflection en fonction de l'angle incident
- des corrections sur les comportements de filter et transmits
- des méthodes supplémentaires pour les media qui deviennent plus rapide et plus doux
- les functions sont utilisables aussi en dehors des isosurfaces
- le support en lecture des formats JPEG et TIFF
- projected_through fait son apparition
- l'atténuation est plus réaliste
- de nouvelles variables concernant l'horloge apparaissent
- de nouvelles variables concernant la taille de l'image apparaissent
- l'inversion d'une transformation est disponible
- la caméra sphérique apparaît
- une nouvelle fonction inside est disponible
- les splines (sans representation 3D) apparaissent
- il est possible de déclarer une variable en tant que nombre flottant depuis un fichier INI
- support simplifier de l'encodage Unicode pour l'objet text
- no_image et no_reflection rejoignent no_shadow
- les reflections métalliques sont disponibles
- les warp peuvent servir pour des mappings
- une collection de petits changements décrits dans la doc de POV-Ray 3.5 en section 2.6.14, qui peuvent empêcher une ancienne scène de fonctionner

C'est vraiment megapov de l'époque qui est repris, avec des restrictions cependant [5].

6.12 POV-Ray31

La documentation est de 1999, et donne généreusement la liste des changements :

- Media remplace Halo et Atmosphere
- apparition de #macro
- apparition des tableaux de variables
- apparition des opérations sur fichiers (#fopen, #fclose, #read et #write)
- ajout de #undef
- les directives #... qui utilise un nombre à virgule doivent désormais se terminer par un point-virgule
- Lathe et Prism peuvent utiliser une courbe de Bezier
- apparition de clock_delta

6.13 POV-Ray30

La documentation est en HTML de 1997, et on nous promet (déjà) une version 4.0 en C++ mais pas avant 1998.

6.14 POV-Ray22

L'ancêtre.

[5] la gestion des polices de caractères pour l'objet text par exemple est différente

6.15 povman

http ://www.aetec.ee/fv/vkhomep.nsf/pages/povman2

En dehors de l'idée d'avoir à compiler des shaders qui me semble n'être l'expression que d'une carence plus importante pour la spécification des patterns, il y a d'excellentes idées de patterns qui devrait être réutilisable.

En particulier, en utilisant la liste des exemples, on devrait pouvoir vérifier que l'ensemble des informations qu'on envisage de fournir à un pattern est désormais suffisant, au niveau de l'interface de la fonction.

6.16 mpipov

Un portage de POV-Ray (depuis la 2.2 jusqu'à 3.1) utilisant MPI (parallélisation).

On a encore le même souci qu'avec PMVPOV (voir 6.9) : le besoin d'avoir la scène complète sur chaque machine.

6.17 megapov05

6.17.1 interior_texture

Je verrais ça comme un motif particulier, renvoyant 0 ou 1 selon l'angle entre la normale de la surface et le rayon. C'est vraiment pas la peine d'alourdir la structure de donnée des objets pour ça.

6.17.2 cutaway_texture

Je verrais ça comme une particularité de la différence CSG, donc encore un autre type de CSG, plutôt qu'une option venant polluer les objets.

6.17.3 Polarical pattern

Son usage n'est pas courant, mais ce n'est pas une raison de ne pas l'avoir.

6.17.4 Proximity pattern

On devrait pouvoir l'optimiser un peu à cause du code des photons[6]. Enfin, quand je dis l'optimiser, c'est aussi le remettre à jour, bien que l'idée semble excellente (même si elle peut être un peu longue à cause des rayons à tracer en plus).

[6]Le code des photons nécessite d'avoir une sphère englobante d'un objet, ce qui est parfois mieux que la boite englobante dont dispose chaque objet de POV-Ray. En particulier, les rotations ne font pas croître une sphère englobante !

6.17.5 Trace

C'est super mignon, c'est même dans POV-Ray 3.6, mais il y a un truc qui m'agace : l'obligation de fournir un objet. Si on veut utiliser toute la scène courante (pour calculer l'emplacement lors d'un empilement itératif aléatoire), il faut la mettre dans une union... et quand on rajoute un objet, créer une nouvelle union et ainsi de suite... très mauvais pour la consommation mémoire.

6.17.6 post-processing

6.17.6.1 focal blur

C'est un effet de caméra, pas un truc à faire après un rendu. On a pas besoin de connaître tout les pixels pour pouvoir le faire.

6.17.6.2 depth

La encore, c'est à la limite un effet de caméra, ou bien un pattern appliqué à un objet pile devant la caméra qui utilise la longueur du rayon réfracté pour obtenir une couleur. Comme les bornes doivent être fournis par le fichier, il n'est même pas nécessaire de connaître tout les pixels pour pouvoir le faire.

J'ai tendance à aimer l'idée du pattern plutôt que la caméra, afin de pouvoir utiliser le pattern ailleurs. Des variantes de ce pattern pourrait être avec :

- la continuation du rayon incident
- le rayon réfracté (presque comme la continuation du rayon incident, mais un changement d'IOR peut intervenir)
- le rayon réfléchi
- la normale à l'intersection (quelque soit son orientation, opposée ou pas au rayon incident)
- la normale à l'intersection ramenée dans le domaine de la réfraction (donc en continuation approximative du rayon incident)
- la normale à l'intersection ramenée dans le domaine de la réflection (donc en opposition approximative avec le rayon incident)

6.17.6.3 Soft glow

Je vois vraiment pas pourquoi ça devrait être fait dans un moteur de rendu 3D.

6.17.6.4 patterned blur

La camera de blur n'a qu'à avoir une variante supportant un pigment (qui sera un patterned_pigment, bien évidement)

6.17.6.5 stars

C'est un remplacement de sky_sphere, pour éviter l'anticrénelage, je suppose. Il vaudrait mieux rajouter ce substitut à sky_sphere et background, au lieu de chercher à rattraper les choses par la suite.

6.17.6.6 posterize

Je vois vraiment pas pourquoi ça devrait être fait dans un moteur de rendu 3D.

6.17.6.7 normal

ça ressemble beaucoup à 6.17.6.2. Je verrais encore bien un pattern qui utilise un rayon secondaire (faut-il les mêmes variations ? j'ai franchement rien contre, mais je crains de me faire plaisir et de complexifier la compréhension des débutants).

Deux grosses questions, cependant :

1. Qu'est-ce qu'on utilise comme valeur en cas d'absence d'intersection ?
2. La valeur d'une normale étant normée, le troisième canal de couleurs est quasiment redondant, à un signe près qui peut disparaitre facilement si l'on ramène la normal dans un demi-plan. Ne faudrait-il pas proposer un autre pattern qui dans les deux premiers canaux mettrait la normale et depth dans le dernier canal ? Faut-il autoriser l'utilisateur à intervertir les canaux ? faut-il aussi l'autoriser à spécifier le demi-plan ainsi que l'affectation des canaux pour la projection ?

6.17.6.8 add, multiply, exponent, clip_color, invert

Je vois vraiment pas pourquoi ça devrait être fait dans un moteur de rendu 3D.

6.17.6.9 find_edge

J'ai des doutes sur la couleur opposée à un gris moyen. En dehors de ça, je ne vois vraiment pas pourquoi ça devrait être fait dans un moteur de rendu 3D (surtout si on peut sortir depth et normal, le traitement sur la couleur est vraiment du traitement de logiciel 2D).

6.17.6.10 Convolution matrix, colour matrix

Je ne vois vraiment pas pourquoi ça devrait être fait dans un moteur de rendu 3D.

6.18 megapov06

6.18.1 post_min, post_max

Je ne vois vraiment pas pourquoi ça devrait être fait dans un moteur de rendu 3D. C'est vraiment du post-processing 2D.

6.18.2 glows

A regarder de plus près au niveau du code, ne serait-ce que pour avoir d'autres idées d'effets que fog et rainbow.

6.19 megapov07

6.19.1 curves, limit_brightness, divide, substract

Je ne vois vraiment pas pourquoi ça devrait être fait dans un moteur de rendu 3D. C'est vraiment du post-processing 2D.

6.19.2 raw

Encore un post-processing, mais celui-là, je ne vois vraiment pas à quoi il sert, en dehors du format Raw, il est impossible de stocker des pixels plus blanc que blanc ou même avec des canaux négatifs.

6.19.3 star

Désormais, au lieu de remplacer background et sky_sphere, l'effet s'ajoute. J'aurais donc tendance à considérer star comme similaire à fog et rainbow.

6.20 megapov10

http ://megapov.inetart.net/index.html

Il y a beaucoup de choses déjà vu ailleurs, y compris megapov07 et Clothray.

6.20.1 f_triangle

C'est une fonction certainement utile pour les isosurfaces, et pour une fois, le triangle a une épaisseur (c'est donc plus un prisme droit triangulaire qu'un simple objet 2D).

6.20.2 Alignement de texte

C'est une extension intéressante qui permet de modifier le positionnement initial d'un objet text.

6.20.3 Résolveur polynomiale accessible depuis la scène

Il serait peut être plus intelligent de rendre l'ensemble des racines dans un tableau, plutôt que de recalculer à chaque fois les racines.

6.20.4 Controle des messages de gradient des isosurfaces

A mon avis, ça ne sert à rien, donc c'est rigoureusement indispensable !

6.20.5 Simulation d'exposition d'un film

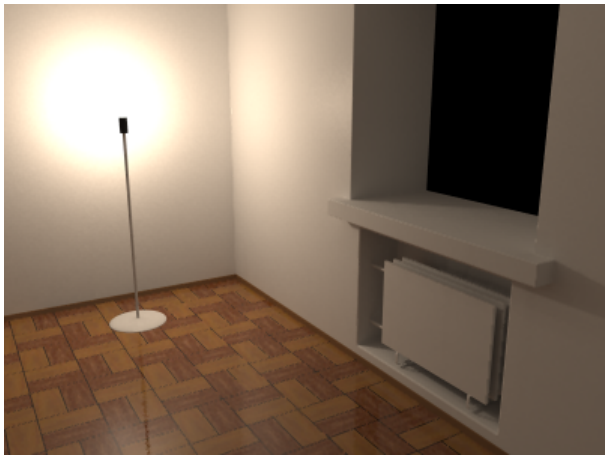

FIG. 6.4 – Sans utilisation de simulation

C'est à la limite du postprocessing, si ce n'est que j'espère que le calcul est fait dans l'espace HSV et non directement dans RGB, pour que seule la luminosité change. La formule est $c \leftarrow g \cdot (1 - e^{-c \cdot f})$, avec g le gain et f la caractéristique de la prise de vue[7].

[7] f est un peu comme le produit de la vitesse du film, l'ouverture du diaphragme et le temps d'exposition. Bien entendu, f permet de compenser le niveau des lumières de la scène.

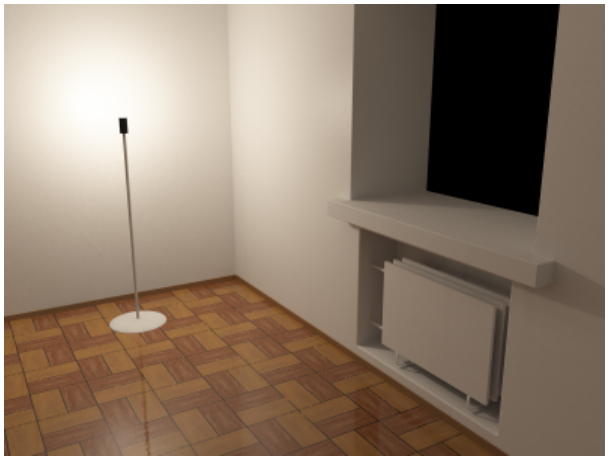

FIG. 6.5 – Avec utilisation d'un f à 1.6

A noter qu'il est possible de changer ainsi la compression[8] d'une image, ce qui peut être fort utile.

6.20.6 Frame_Step

C'est une option d'environnement intéressante, qui permet non seulement de partager le travail, mais également d'inverser l'ordre de rendu des trames.

6.20.7 Fichiers temporaires de radiance

Les noms des fichiers temporaires de radiance comportent désormais le numéro de la trame. C'est une très bonne chose, surtout avec des machines partageant les mêmes répertoires.

6.20.8 Simulation physique

C'est une superbe extension qui permet de faire des simulations. Je pense qu'il n'est pas urgent de le reprendre, même si c'est très intéressant.

Ce qui me gêne le plus actuellement, c'est l'universalité obligatoire des paramètres de l'environnement : il n'est pas possible de spécifier des coefficients de frictions différents par exemple.

L'accès aux données résultant de la simulation est également contraignant : il faut passer par la lecture d'un fichier, alors qu'il serait peut-être plus simple de fournir des tableaux.

Le bon coté, c'est qu'il est possible de contraindre des points à évoluer selon une spline, en fonction du temps. Le mauvais coté, c'est que la spline ne sert pas de guide

[8] Au sens d'une comparaison entre l'oeil humain et une photographie, au niveau de l'adaptation à la luminosité.

aux mouvements du point, mais bel et bien de définition de position en fonction du temps[9].

6.21 megapov11

http ://megapov.inetart.net/index.html

La version courante de megapov, basé sur POV-Ray 3.6.

6.21.1 Fur

C'est un nouveau type de média du genre scattering. Il ne devrait pas y avoir de problème à le reprendre, même si la syntaxe changera certainement un peu.

6.21.2 Type checking

Un concept intéressant, permettant de faire éventuellement du polymorphisme comportemental dans des scripts. Je me demande si c'est vraiment utile et raisonnable de le reprendre tel quel.

6.21.3 Measurement of dimensions in vectors

Ce patch permet de connaître la dimension d'un vecteur. C'est un concept intéressant, mais je me demande si il est encore utile de permettre du polymorphisme comportemental : je trouve que ça a tendance à ralentir l'apprentissage de l'utilisateur d'un script ou d'une macro.

6.21.4 New spline types

Il faudrait vraiment faire un graphique comparatif des différentes fonctions de splines possibles, mais je n'ai rien contre l'ajout de nouveaux types de splines.

6.21.5 Sizes of images measurement

Je ne suis pas persuadé de l'utilité de cette extension, ni même de son comportement si il y a plus d'une image accroché à l'objet passé en paramètre. De plus, la surcharge sémantique me paraît être très mal choisie.

[9] D'où encore des conflits possibles avec la valeur de clock. ça n'aide vraiment pas l'utilisation dans des animations un peu longue.

6.21.6 Reduced memory requirement for each object

Cette extension, qui remplace une Union par un pointeur dans les structures internes, est sans aucun intérêt dans un code propre C++.

6.21.7 High Dynamic Range image type from MLPov

Je demande si le PNG avec 48 bits ne ferait pas un peu mieux en terme de couverture que ce format d'image. Mais rien n'empêche de pouvoir lire les HDR.

6.21.8 High Dynamic Range image write support

Il est tout à fait possible de supporter également l'écriture des HDR. Le format en est fort simple, et n'est pas plus complexe que le mode Raw.

6.21.9 Connection-connection collisions for mechsim patch

A mon avis, il est tout à fait possible de reprendre cette extension sans rencontrer de problème particulier.

6.21.10 Normal modifier for transforms

En dehors de l'occupation mémoire, je ne comprends pas pourquoi cette modification n'est pas active par défaut. Et comme c'est en fait spécifique aux meshes, je pense que ça devrait en fait être intégré à la transformation d'un mesh plutôt que d'être visible dans la création d'une transformation.[10]

6.21.11 String function with numbered output filename

C'est une fonctionnalité intéressante qui peut être reprise sans trop de problème.

6.21.12 New user_defined camera projection type

C'est une superbe généralisation et ouverture de l'objet camera qui devrait être reprise.

[10] Les meshes sont les rares objets qui autorise l'utilisateur à forcer les valeurs des vecteurs normaux. Il y a aussi les smooth_triangles et peut-être quelques autres. Il faudrait intégrer le calcul correct de la normale à tout ces objets. Je ne suis pas sûr que laisser l'option ouverte à l'utilisateur soit une bonne chose.

6.21.13 New camera_view pigment type

L'idée est plaisante, mais une étude plus approfondie des variantes me semble nécessaire. En fait, j'aurai tendance à vouloir que les postprocessing lui soient applicables tout comme sur la caméra classique, plutôt que d'avoir des options spécifiques.

Ce qui me ramène à la problèmatique pour 6.17.6.2, de savoir si on en fait un postprocessing ou un simple pattern. Peut-être faut-il les deux[11] ?

6.21.14 Angle of incidence from MLPov

C'est un pattern qui ressemble beaucoup à 6.31. L'option de fournir un point fixe peut-être intéressante.

6.21.15 Projection pattern from MLPov

C'est un pattern intéressant, même si il faut activer blur pour avoir d'autres résultats que les extrêmes 0 et 1.

6.21.16 Custom radiosity sampling directions

Cette extension permet un meilleur contrôle des échantillons de radiance, et je n'ai rien contre.

6.21.17 Randomised radiosity sampling directions

La notion de verticalité de cette extension est implicite et peut ne pas correspondre à celle de la scène. Il serait préférable en fait de fournir une rotation explicite quelconque qui serait appliquée aux directions des échantillons de radiance.

6.21.18 Reintroduced motion_blur from MegaPov 0.7 with new type feature

Le retour de motion_blur, avec des variations sur les époques d'échantillonnages, me conforte dans mon idée de changer la syntaxe de cette extension (voir la section 6.2).

[11] Avec un comportement légérement différent : dans le cas d'un pattern, on utilise la normale au point de l'objet où l'on se trouve pour avoir le rayon à tracer, alors que dans le cas d'un postprocessing, c'est la camera qui donne le rayon. Je me demande si dans le cas du pattern, il faut ou pas appliquer les perturbations de normale de l'objet. Je crains que ça ne soit en fait une option du pattern (voire même avoir un bloc de perturbations de la normal propre au pattern, car après tout, on peut fort bien vouloir séparer les deux perturbations).

6.21.19 New post processing feature

C'est une approche intéressante, surtout qu'elle permet de sauver plus d'une image. En fait, l'image originale n'est plus modifiée, mais on crée en parallèle des images modifiées, mais sans anti-crénelage[12].

Une étude plus approfondie du contenu des actions de « pprocess.inc » me semble nécessaire.

J'ai toujours un doute sur la notion UV.

6.22 macmegapov

C'est une version binaire de megapov pour Mac, basé sur megapov 1.1 et POV-Ray 3.6.1.

6.23 smpov

6.24 Superpatch

Beaucoup de choses ont été reprise depuis dans POV-Ray 3.6.

Le source semble ne plus être disponible, mais on doit pouvoir retrouver le code dans megapov 0.5.

6.24.1 bicubic_patch

Un type 2 fait son apparition avec simplement une méthode de calcul différente et un paramètre accuracy.

Un type 3 fait également son apparition, qui utilise des vecteurs pondérés 4D (x,y,z,w) pour la spécification des points.

Mise à part qu'il serait préférable de nommé les types plutôt que de les numérotés, il ne devrait pas y avoir de problème pour reprendre ça.

6.24.2 bezier_patch

http ://atrey.karlin.mff.cuni.cz/~0rfelyus/

Une nouvelle forme, qui dispose de deux variantes, selon que les vecteurs sont 3D ou pondérés 4D, avec possibilité de découpe. Il manque, je pense, une belle illustration pour susciter un intérêt pour le bezier_patch (surtout qu'avec la possibilité de découpe, ça peut être très rafraichissant par rapport à un bicubic_patch toujours rectangulaire).

[12] Je me demande si on ne pourrait tout de même pas effectuer une moyenne pour chaque valeur, même si le critère d'anti-crénelage demeure la distance dans l'espace HSV entre les différents pixels.

6.24.3 Avis personnel

Je me demande si il y a vraiment un intérêt à avoir une syntaxe d'entrée qui permettent les vecteurs pondérés 4D.

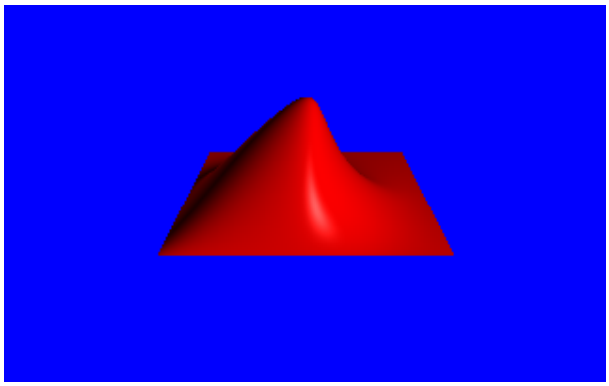

FIG. 6.6 – Poids importants

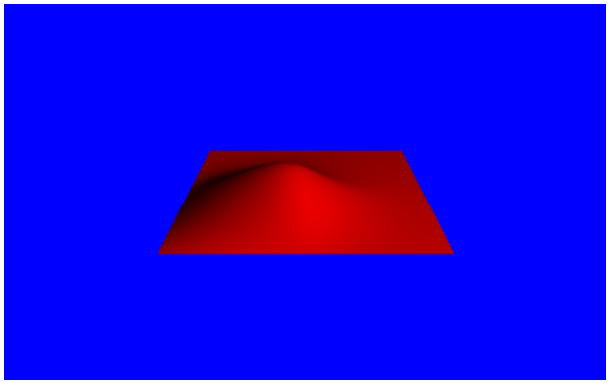

FIG. 6.7 – Poids modérés

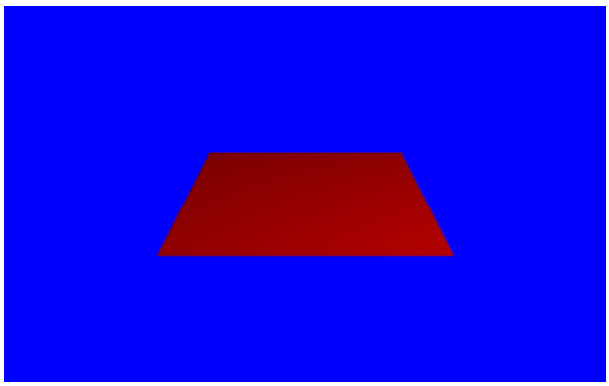

FIG. 6.8 – Poids identiques

En fait les vecteurs pondérés 4D non rien à voir avec les coordonnées 4D qu'utilise POV-Ray en interne, mais sont spécifiques à la surface de Bezier (voir 6.6, 6.7 et 6.8).

Comme on peut le voir sur 6.9 et 6.10, la découpe peut être très intéressante.

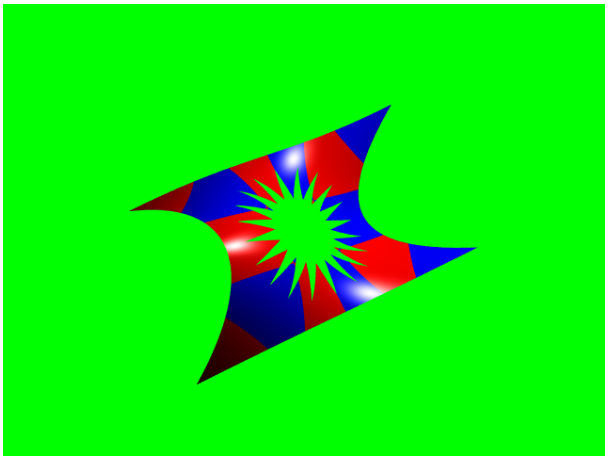

FIG. 6.9 – La découpe intérieure

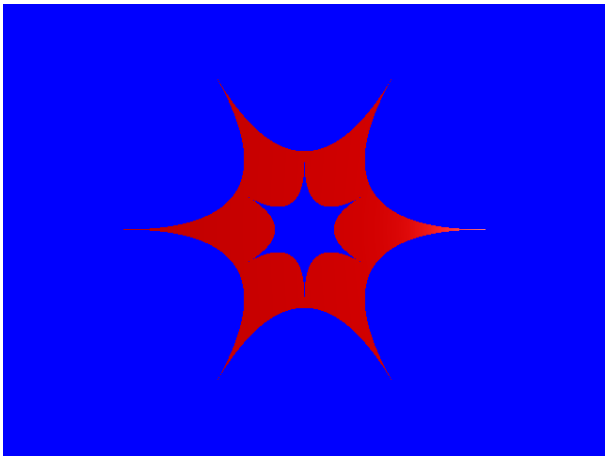

FIG. 6.10 – La découpe compléte

6.25 Square & Triangular

http ://jgrimbert.free.fr/pov/patch/pattern.html

Deux simples patterns (voir 6.11 et 6.12)qui ne devraient pas poser de problème à implémenter.

6.26 Warps

http ://jgrimbert.free.fr/pov/patch/warp.html

Le truc d'un warp est d'entrer avec les coordonnées d'un point et de resortir avec celles d'un autre. C'est similaire à une matrice de transformations, si ce n'est que la transformation effectuée peut ne pas pouvoir s'écrire avec une matrice (et ça ne s'applique que sur les patterns.

Il est tout à fait envisageable de reprendre ces warps en particulier.

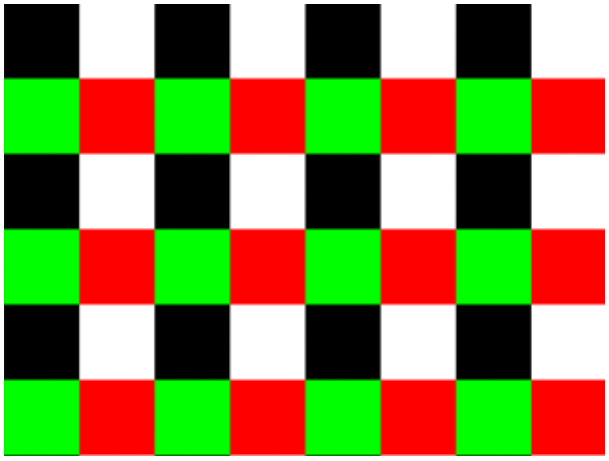

FIG. 6.11 – Square

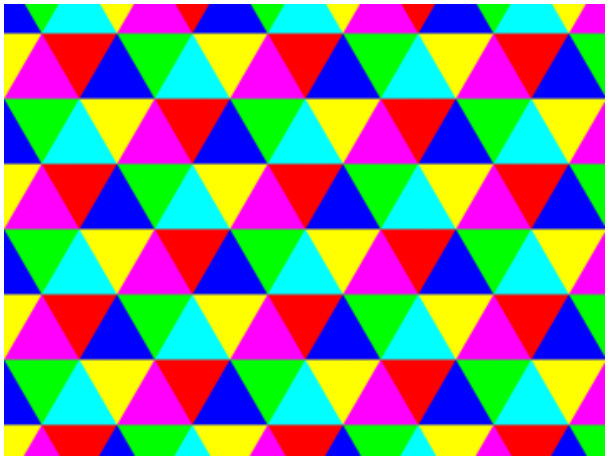

FIG. 6.12 – Triangular

6.27 Transmit

http ://jgrimbert.free.fr/pov/patch/transmit.html

La gestion du coefficient de transmission est un peu bizarre dans la version 3.1g de POV-Ray.

Le modèle est linéaire et symétrique pour les utilisations normales du coefficient, c'est-à-dire entre 0 et 1, mais se refuse à être négatif pour le pigment de l'objet (alors qu'il n'y a pas ce refus pour la couleur transmise).

Mais avant d'aller plus loin, une explication s'impose : le coefficient transmit denote normalement le pourcentage de fond qu'une surface laisse apparaître.

- à 0, la surface ne laisse rien passer, elle a une couleur propre bien visible (enfin, selon l'éclairage)..
- à 1, la surface n'a pas de couleur propre visible car son apparence depend entièrement des rayons qui sont réfractés.

En fait, si on appel R la couleur provenant de la réfraction et C celle de la surface (si elle était sans transmission), pour un coefficient t de transmission, la couleur résultante pour t entre 0 et 1 se définit comme ((1-t).C + t.R) .

A noter que cette formule reste valide en dehors de l'intervalle [0,1] pour t, et c'est bien ce que fait POV-Ray 3.1g, au moins pour t<0.

Par contre, et sans raison apparente, lorsque t est supérieur à 1, la formule devient brusquement (t.R) .

Ce patch a déjà été incorporé dans la version 3.6, donc il n'est pas nécessaire d'y faire davantage attention.

6.28 Interunion & Intermerge

http ://jgrimbert.free.fr/pov/patch/inter.html

Une paire de variations qui change des classiques CSG, on devrait pouvoir les implémenter sans trop de soucis.

6.29 Ovus

http ://jgrimbert.free.fr/pov/patch/ovus.html

Une simple forme (voir 6.13) qui ne devrait pas être difficile à implémenter.

6.30 Patch de tesselation

http ://jgrimbert.free.fr/pov/patch/tessel/index.html

Outre le format GTS, il serait bien de pouvoir charger d'autres formats comme PLY.

En fait, à condition que le format soit publié et documenté, on devrait pouvoir être capable de lire n'importe quel mesh.

6.31 Celluloid

http ://jgrimbert.free.fr/pov/patch/cellulo.html

Un simple motif qui nécessite d'avoir accès à la normal et au rayon, rien d'exceptionnel a priori et tout à fait faisable.

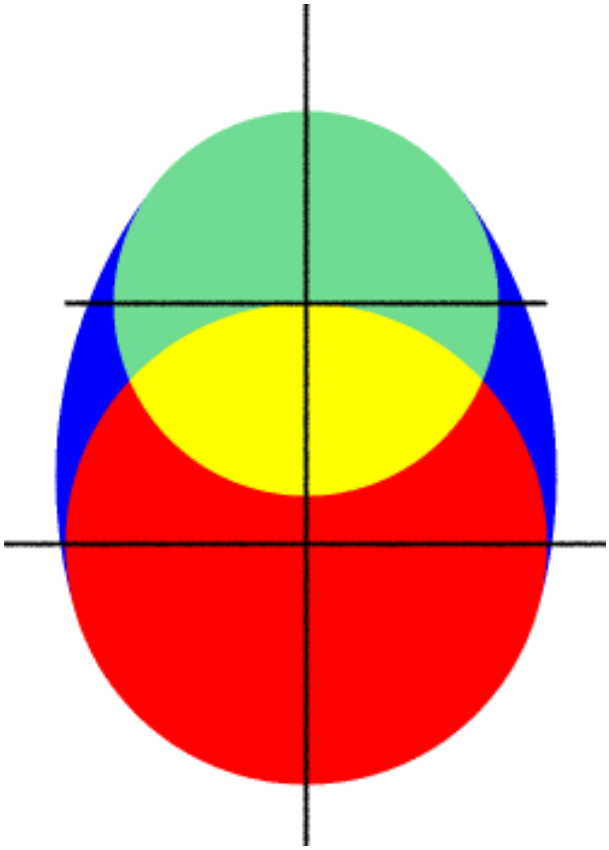

FIG. 6.13 – Ovus

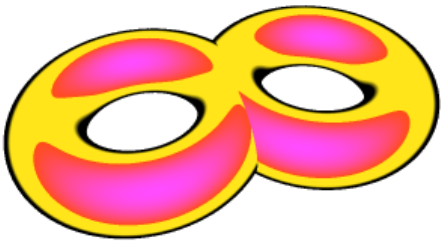

FIG. 6.14 – Illustration de celluloid

6.32 Pavage du plan

http ://jgrimbert.free.fr/pov/patch/pave/index.html

Une collection de motif qui ne devrait pas poser de problème à implémenter, même si la syntaxe est certainement à revoir un peu.

6.33 Supertorus

http ://astronomy.swin.edu.au/~pbourke/surfaces/torus/

Une forme qui est au tore ce que la superellipsoïde est à la sphère (voir 6.15). Il ne devrait pas y avoir de problème à l'implémenter[13].

[13] http ://astronomy.swin.edu.au/~pbourke/surfaces/ contient également une collection de formes qu'il pourrait être intéressant d'étudier.

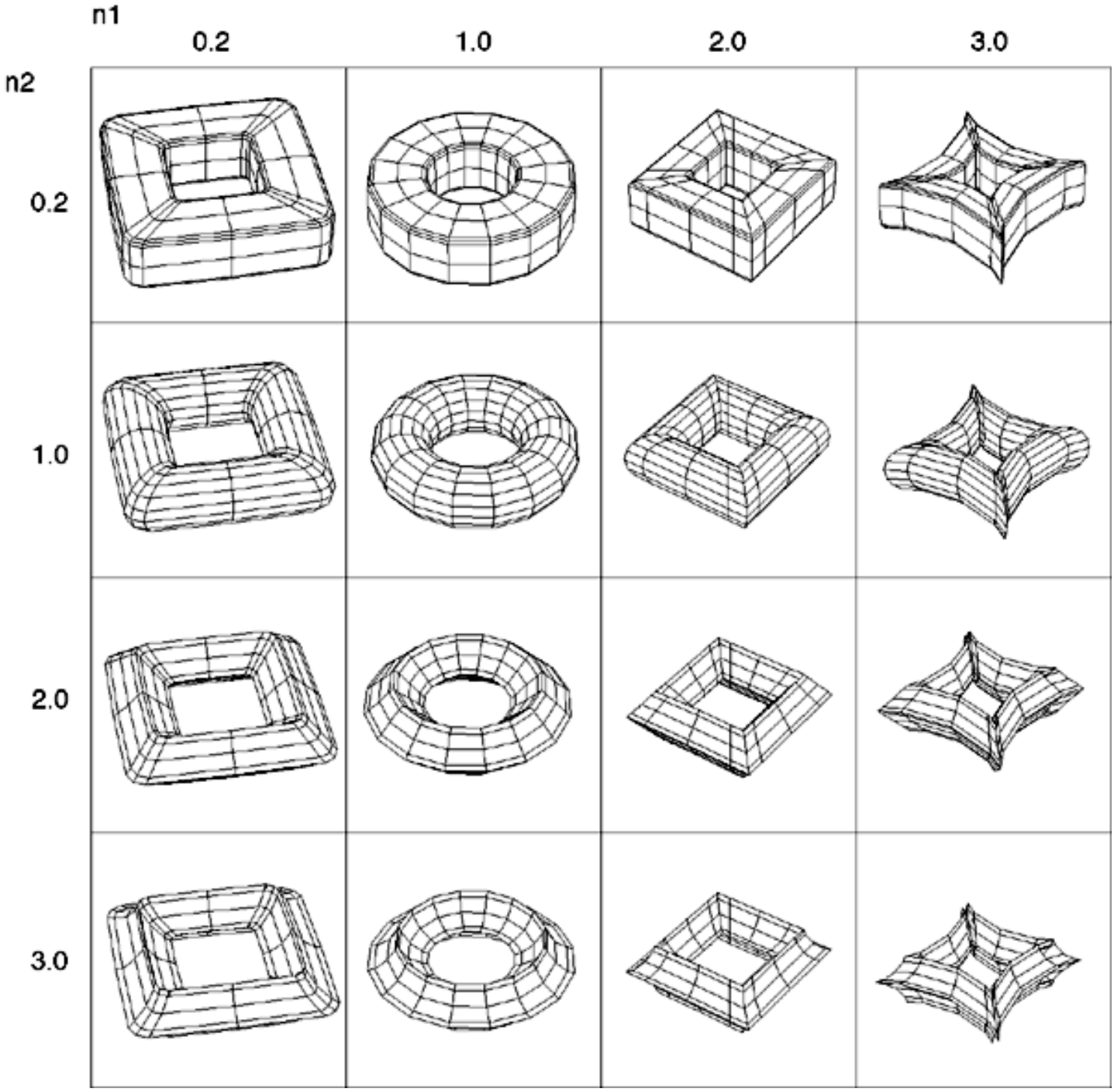

FIG. 6.15 – Supertorus

Chapitre 7

D'autres patches à écrire

7.1 Des lettres biseautés

Avoir des polices biseautées... en centrant chaque caractère et en faisant une macro qui empile entre 15 et 25 épaisseurs, on doit bien obtenir quelquechose de sympa.

Il serait cependant intéressant de ne pas avoir en mémore 15 à 25 copies de la même lettre.

7.2 Des objects arrondis

Il existe des macros qui dupliquent l'objet afin de l'arrondir, si on pouvait faire la même chose sans la consommation mémoire.

7.3 Gothique

Ajouter une forme qui permette de faire une croisée d'ogive gothique (arc brisé, demi-cercle porteur). La connexion des deux arcs de cercles est particulièrement floue (mettre une surface réglée ? ou bien généré un mesh automatiquement ? Je trouve le mesh particulierement laid, même si c'est une solution de facilité).

Les piliers d'une croisée d'ogive complète doivent être sur le même cercle. La clef de voûte de l'arc brisé doit être à la même hauteur que la clef de voûte de la croisée d'ogive (sinon, l'arc brisé devient beaucoup plus arbitraire et surtout les poussées deviennent beaucoup plus problèmatique à équilibrer).

7.4 Strip_texture/material/...

Partant d'un objet (potentiellement complexe) avec textures variées, on obtient un objet aux formes identiques, mais sans ses textures (récursivement, l'objet originel est parcouru pour en retirer les textures). On peut donc appliquer de nouvelles textures sur le nouvel objet au niveau le plus haut.

Il est certainement utile de pouvoir décliner les variations des éléments retirables.

7.5 CSG

Au début des temps, il y avait la CSG : UNION et INTERSECTION. Déjà, une petite extension pointait son nez : DIFFERENCE, implémenté comme une INTERSECTION avec l'INVERSE (l'INVERSE étant une simple possibilité des objets de base)[1].

Puis vint MERGE, lorsque les objects sont transparents, le fait de retirer les surfaces interieures est important.

Outre le patch INTERUNION et INTERMERGE (6.28), plutôt anecdotique, une nouvelle option fit son apparition pour DIFFERENCE & INTERSECTION : CUTAWAY, qui permettait de conserver la texture de l'objet principal (parfois fort complexe) sur la portion en intersection avec l'objet secondaire[2]. Il serait certainement plus simple d'introduire enfin une différence de comportement entre DIFFERENCE et INTERSECTION : l'une impliquant CUTAWAY, et l'autre pas.

Enfin, j'ai constaté que souvent l'UNION est utilisé simplement pour grouper des objets, sans pour autant qu'il y ait des recouvrements des objets à l'intérieur de l'UNION. C'est certes pratique pour la manipulation, mais ça ralentit parfois beaucoup le rendu, car il faut tester chaque intersection au minimum contre chacune des boîtes de contenu[3]. Je me demande cependant, si il ne serait pas intéressant d'avoir une opération GRAPPE qui se contenterait d'appliquer les modifications sur l'ensemble de ces objets, en supprimant lors du rendu ce lien artificiel, ne laissant que des objets isolés en place, comme si ils avaient été placé individuellement. Dans le cadre d'une approche pure boîte de contenu, ce n'est pas intéressant, mais dans

[1] Ou le contraire, l'intersection comme une difference avec l'inverse. C'est pareil.

[2] Alors que le comportement de difference & intersection était jusque là d'utiliser la texture de l'objet secondaire sur cette zone.

[3] Ce qui permet, en faisant des regroupements plus géographique en plusieurs couches d'UNION de gagner sensiblement par rapport à un groupe unique.

le cadre d'une répartition des objets dans l'espace (et de leur boîte de contenu) dans un arbre, on échange une boite potentiellement gigantesque, donc non optimisable, contre une collection de boîte plus petites, donc potentiellement optimisable[4].

Les objets de CSG devraient donc être :

- UNION
- MERGE
- INTERSECTION[5]
- DIFFERENCE[6]
- INTERUNION
- INTERMERGE
- GRAPPE

[4]On gagne également sur les temps de calculs des matrices de transformations, puisque les calculs peuvent être propagé aux objets individuels dès la mise en place de la grappe. Mais je m'avance, voir 10 ; Il faut peut-être envisager également un CSG OBJET dans ce cadre, bien qu'il n'est pas évident de le faire apparaître dans le langage.

[5]Sans aucun CUTAWAY possible.

[6]Avec CUTAWAY systématique.

Chapitre 8

Organisation de Crayon

Il y a, c'est indispensable, un parseur, qui construit les données. Il est difficile de paralléliser ce travail.

Normalement, une fois les données construites, elles ne devraient plus être accéder qu'en lecture.

Il est alors temps de passer en mode multithread : L'un des threads remplit un mécanisme de communication où il demande la couleur des rayons issue de la caméra. Un ensemble de threads dont le seul objet est de calculer la couleur d'un rayon vient s'alimenter à ce mécanisme et fournit ses réponses via un mécanisme de communication séparé. Le premier thread récupére, corrèle et aggrège les réponses, s'occupant d'afficher l'image, de son écriture dans un fichier ainsi que de tenir l'utilisateur au courant des progrès (voir 8.1).

8.1 Les détails

8.1.1 L'anti-crénelage

il y a grosso-modo deux solutions.

1. C'est le premier thread qui prend en charge la gestion de l'anti-crénelage, donc de prendre l'initiative de redemander des rayons supplémentaires pour un pixel donné.
2. C'est le calcul de la couleur du rayon qui fait l'anti-crénelage, le premier thread recevant une réponse définitive pour le pixel.

La première solution à l'avantage de la simplicité de la communication : les seules choses à fournir sont les informations du rayon, mais elle complexifie le premier thread dans sa gestion des réponses, puisque contrairement à POV-Ray, l'ordre d'arrivée des réponses n'est plus forcément respecté.

La seconde solution à l'avantage de simplifier la tache du premier thread, mais augmente certainement le volume d'informations à fournir.[1]

> Je n'ai peut-être pas bien compris mais je pense à deux choses. D'abord au progrès considérable (et tant attendu) que constituerait un AA automatique au moins des pigments, sinon des textures. Le genre mipmapping, qui permettrait enfin de faire du papier quadrillé sans pousser l'AA à fond en récupérant des pointillés. Autre chose sur l'AA en multithread : ce n'est pas pour rien qe pvmpov travaille avec des blocs rectangulaires, et si possible carrés. Ou alors tu envisages sérieusement de faire les threads au niveau pixel, avec une communication inter-process ou de la mémoire partagée, et le "les choses sont encore plus compliquées" du 6.10 devient un enfantillage à côté. [2]

Le coté mipmapping m'avait complétement échappé, c'est un aspect à creuser, surtout si ça peut nous éviter de faire un anti-crenelage au niveau de la caméra. Il me semble qu'alors la section d'un rayon n'est plus un simple point, mais une surface.

Une recherche sur le site de Povray au sujet des mipmap indique qu'une discussion a déjà été tenue en août 2003, concluant à une complexité énorme dans le cas de Povray[3].

A part ça, effectivement, la première solution envisage effectivement de faire travailler les threads au niveau souspixel, avec une communication inter-thread, et la scène en mémoire partagée. Le calcul de blocs rectangulaires génére

[1] Je ne suis pas sûr que la deuxième solution soit même viable dans le cadre des réponses arrivants dans le désordre. Il me semble y avoir un problème de poules et d'oeufs dans l'amorçage de la pompe. Il faut peut-être avoir une structure de réordonnancement dans le cas de la première solution.

Il est bien sûr possible d'envisager un rendu par morceau, l'anti-crénelage étant pris en compte à l'intérieur du bloc par le thread de rendu, mais reste le problème de la jointure. A moins de faire se chevaucher les blocs, mais on gaspille alors de la ressource à recalculer les chevauchements.

[2] François Dispot, 14 janvier 2005

[3] Ce n'est pas forcément le cas des moteurs n'utilisant que des meshes, mais la projection d'une surface sur un objet de Povray n'a rien d'évidente, sans même se poser la question d'une projection à cheval sur plusieurs objets. Il est par contre envisageable d'avoir un pattern qui en fonction de la distance parcourue par le rayon effectue un moyennage entre différentes entrées de sa table de patterns. Le moyennage est indispensable pour éviter l'apparition d'un effet de bandes.

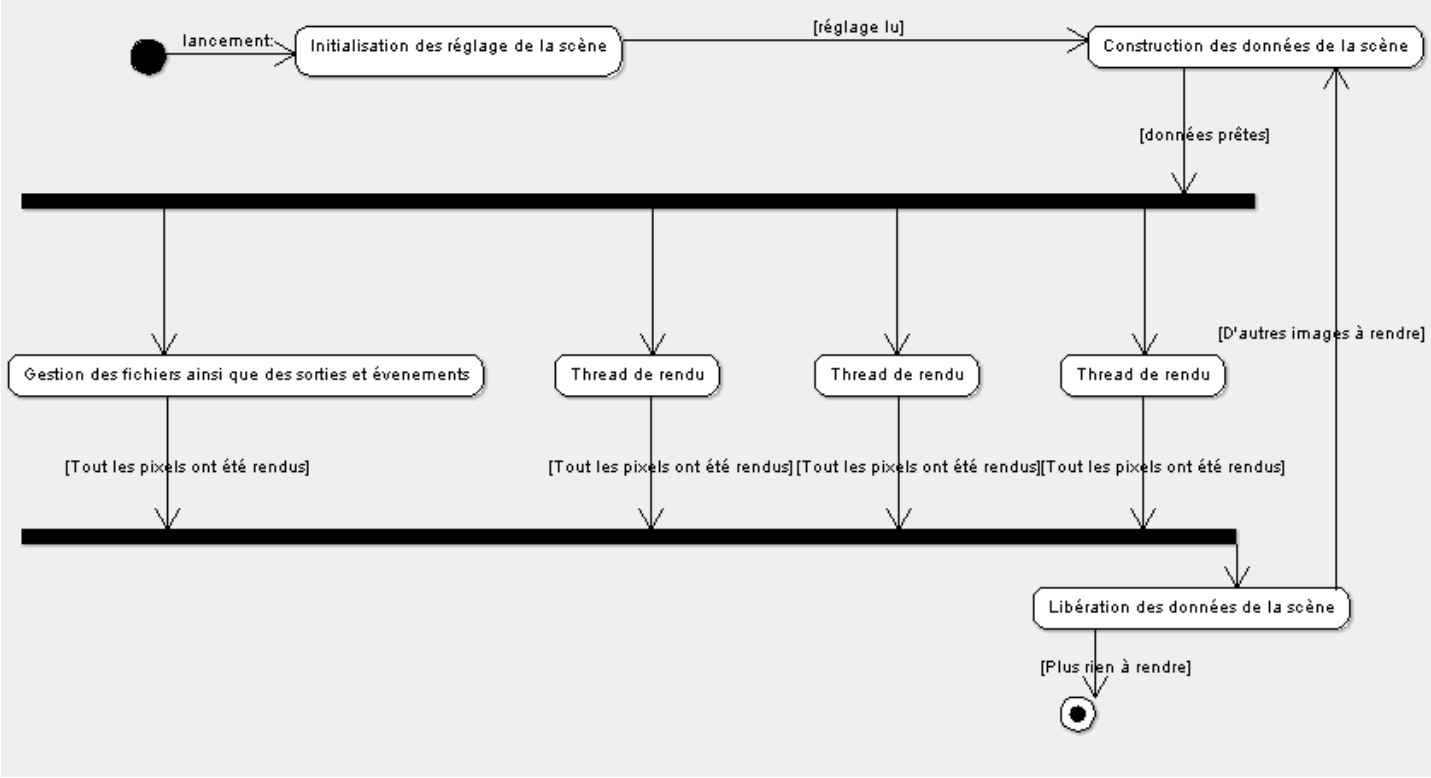

FIG. 8.1 – évolutions des threads

quand même un gachis au niveau des jointures des blocs et ça me dérange de perdre ne serait-ce que 5% de manière systématique.

8.1.2 Les photons

L'algorithme utilisé par POV-Ray consiste à traiter chaque source de lumière séquentiellement. On devrait pouvoir gagner un peu de temps en parallélisant l'émission des photons à raison d'une lumière par thread. Par contre, étant donné le volume de donnée, je pense qu'il est préférable de ne pas avoir de thread spécialisé dans la collecte des données, mais de plus simplement protéger la structure par un sémaphore.[4]

Il est probable qu'il faille générer une liste temporaire des lumières à traiter, et coder le tout de manière à utiliser au mieux le nombre de thread disponible.

Il est aussi possible qu'on désire régler différement le nombre de threads pour les photons du nombre de threads pour le rendu de l'image (voir 8.2).

Il n'est par contre certainement pas nécessaire de continuer comme POV-Ray à essayer de compresser les données de couleurs des photons sur quatre octets, au risque d'avoir des problèmes de bornes si les intensités sont trop différentes. Cela relève de l'optimisation de structure qu'avait POV-Ray, il est certainement plus simple de correctement spécifier la classe des photons afin de pouvoir se livrer à ce genre de peaufinage.

8.1.3 La radiance

Bon, déjà la portabilité est limité pour l'implémentation de la radiance avec POV-Ray, car la structure de donnée utilise la représentation normalisé des nombres flottants pour l'indexation spatiale.

En dehors de ça, la radiance de POV-Ray utilise une structure en écriture lors du rendu, outre qu'elle utilise un effet de bord de la prévisualisation (faire tracer des rayons) afin d'obtenir la population initiale de sa structure.

Il faudrait définitivement supprimer ce besoin de prévisualisation, en séparant le remplissage de la structure de la prévisualisation[5].

Le fait d'avoir une structure en écriture impose de garder un sémaphore pour protéger l'écriture dans cette structure. On pourrait également se poser la question de la cohérence de l'accès à cette structure par des threads en lec-

[4] L'idéal serait bien entendu que la structure de donnée n'est même pas besoin d'un sémaphore, mais je crains que l'aggrégation de photons ne soit problèmatique en la matière.

[5] Ce n'est pas tant le besoin d'avoir à prévisualiser que le fait que des changements de réglage dans la prévisualisation peuvent avoir des conséquences importantes sur les données de radiance.

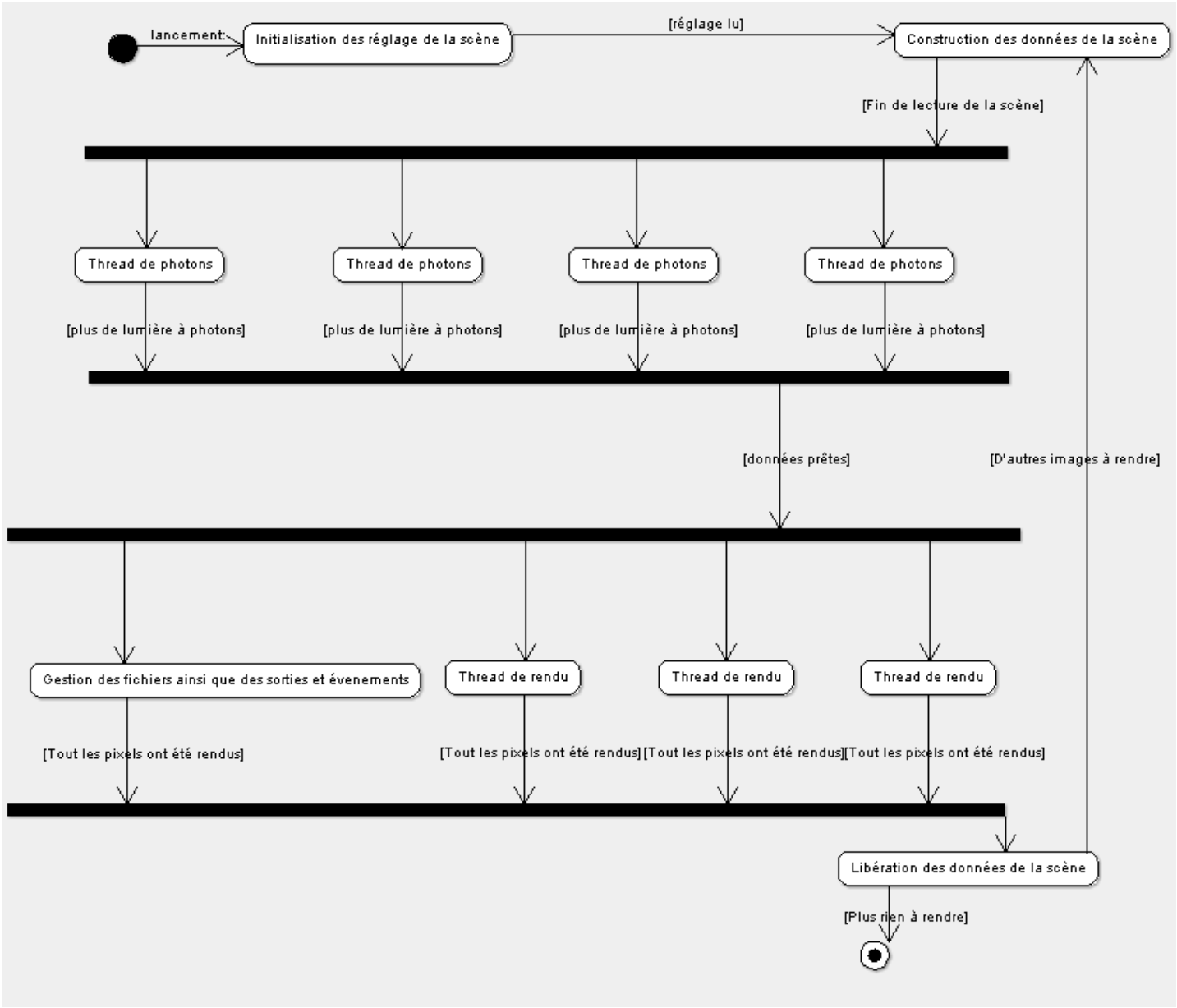

FIG. 8.2 – évolutions des threads avec les photons

ture pendant qu'un thread est en train d'y accéder en écriture. Je crains qu'on ne puisse empêcher le goulet d'étranglement et que les performances s'en ressentent un petit peu (voire beaucoup).

Une autre question en suspens concerne la cohabitation des photons et de la radiance : est-ce que les photons doivent être calculé avant ou après la radiance ?

Il est a peu près certains que les photons ne dépendent pas de la radiance, par contre je ne suis pas du tout convaincu de l'inverse.

En fait, il suffit de considérer un jet de photons sur un miroir pour s'apercevoir que la radiance doit absolument avoir lieu après les photons.

Troisième partie

Développement

Chapitre 9

Le parseur

Le parseur me semble être hautement sensible à son contexte, en dehors de l'évaluation des expressions et du traitement des directives.

Je me demande dans quelles limites il ne serait pas plus simple et plus souple d'avoir un parseur dont la liste des objects reconnus serait dynamiquement mise à jour, un peu à la manière dont les contextes sont empilés pour les déclarations de variables locales.

Dans le parseur de base, on mettrait les tokens de plus haut niveau, associés chacun à une fonction à appeler. Au début de cette fonction, on enrichirait le parseur d'un bloc supplémentaire qu'on remplirait avec les tokens et fonctions possibles à ce niveau.[1]

Cette approche n'a qu'un seul but : éviter d'avoir à tenir à jour une table unique table de tokens. Peut-être est-il plus simple pour obtenir le même effet d'avoir une fonction d'initialisation pour chaque classe qui enregistrent ses tokens automatiquement et en récupére les valeurs. Il est cependant dommage de perdre ainsi la capacité d'utiliser le switch(). En fait, il n'est pas nécessaire pour l'analyse des tokens d'avoir plus d'un contexte de tokens actif à la fois (en dehors des tokens pour les variables, mais là l'empilage est la règle, et à priori l'évaluation d'un token de variable (ou d'une expression) est normalement prévisible puisqu'on peut mettre en place une syntaxe idoïne). Il est par contre nécessaire que certaines classes (comme la classe virtuelle des patterns) regroupent les racines de leurs classes héritantes et qu'elles exportent une fonction permettant d'obtenir une instance de la classe en question (ou d'une classes héritantes, bien entendu).

Le parseur analyse donc le flux en entrée en fonction du contexte lexical courant, le parcours du texte ayant pour effet que les fonctions spécifiques d'analyses de chaque classe empilent, remplissent et dépilent des contextes lexicaux au fur et à mesure pendant que le parseur cache le traitement des directives ainsi que la gestion des contextes de variables.

On a donc pour le parseur deux piles :

1. Une pile de contextes de variables, où chaque contexte est actif mais potentiellement masqué par un contexte plus récent sur la pile
2. Une pile de contextes léxicaux, où seul le sommet de la pile est actif.

La première pile associe à un token une valeur, qui peut être un simple nombre ou une référence à un objet. La seconde pile associe à un token une fonction (de prototype encore à définir, mais c'est le même pour toutes les fonctions) appeler automatiquement par le parseur pour poursuivre son avancée dans l'analyse du fichier de la scène. En séparant les deux piles, et en obligeant à avoir une syntaxe prédictive (à un moment donné, on attend soit un mot-clef soit une expression), on supprime également l'inconvénient d'avoir des mots-clefs résérvés.

L'accès aux champs d'un objet nécessite que l'évaluateur d'expression soit capable d'avoir accès à une fonction lié à une référence qui peut être n'importe quel objet possible. Outre une syntaxe particulière pour prévenir le parseur, il faut donc que chaque classe du modèle qui supporte l'accès à ces champs hérite d'une classe fournissant une fonction différente de traitement[2].

[1] D'ailleurs, est-ce un bloc supplémentaire ou un bloc alternatif ? Il ne faut pas perdre de vue qu'il y a plusieurs endroits dans le modèle de données où l'on peut avoir une récursion potentiellement infinie : toutes utilisations de combinaisons, ainsi que tout ce qui concernent les patterned_*. Un parseur localisé semble intéressant, mais j'ai peur de la complexité pour les récursions.

[2] Le traitement normal consistant à construire des objets, ce traitement-ci consistant à fournir une valeur (qui peut très bien être aussi une référence à un autre objet).

Chapitre 10

Empilement des transformations

Parce qu'on peut vouloir réutiliser un objet dans plus d'une combinaison[1], une optimisation à laquelle il faudra certainement renoncer, c'est l'agrégation des transformations dans les objets composés au niveau des feuilles. Ce qui veut dire qu'au lieu d'avoir une seule multiplication d'un rayon par une matrice pour tester une feuille, il faudra itérer les multiplications au fur et à mesure de la descente de l'arbre. Il faudra donc porter une attention particulière pour éviter d'augmenter sans nécessité les profondeurs des combinaisons.

A cause des Warps, l'évaluation des textures nécessite que l'on effectue une succession de transformations, certaines utilisants de simples matrices et d'autres des fonctions spécifiques d'un warp. Il ne faudra quand même pas négliger de simplifier deux matrices successives.

Je me demande comment traiter proprement un warp du style « reset_children ». Il faudra certainement passer aux warps un nombre d'information plus important que le seul point d'intersection (notamment lors de leur application aux normales).

[1] Afin de faire quelques économies mémoires sur la duplication des combinaisons lourdes, d'une manière similaire à la gestion des meshs par Povray.

Chapitre 11

Classes annexes

Il est nécessaire d'avoir quelques objets supplémentaires, en dynamique, par rapport aux modèles de données :

- Une intersection, basé sur une origine et un vecteur servant de base (et donnant la direction), elle fournit un point en fonction d'une longueur. D'ailleurs elle ne fournit pas que les coordonnées d'un point, mais également d'autres informations.
- Une pile d'intersection, qui ordonne automatiquement les intersections en fonction des longueurs croissantes (les origines et vecteurs des intersections peuvent être différentes).