GNU RT: vers une mise en œuvre GPL d'OpenRT

Benjamin Segovia[1,2], Jean-Claude Iehl[2] et Bernard Péroche [2]

[1]ENTPE: Ecole Nationale des Travaux Publics de l'Etat, Vaulx-en-Velin
[2]LIRIS: CNRS, INSA de Lyon, Université Lyon 1, Université Lyon 2, Ecole Centrale de Lyon

Abstract

This article presents our project of a GPL implementation of the OpenRT API. OpenRT is a programming interface dedicated to ray tracing and interactive / real time rendering of complex geometric scenes and also to various physical phenomenas such light reflexion, light refraction and global illumination effects. In this paper, we thus detail the state of the art required to understand the implementation of such an API and the first results we obtained. Our goal is first to provide users with a simple interface handling with difficult algorithms and non-trivial numerical schemes and then to offer programmers a lot of code freely reusable.

Cet article présente notre projet de mise en œuvre GPL de l'interface de programmation OpenRT. OpenRT est une API dédiée à l'affichage et au rendu de scènes par la méthode algorithmique du lancer de rayon. Elle a pour but de gérer des objets géométriques complexes et une grande classe de phénomènes physiques comme la réfraction, la réflection de la lumière, les phénomènes d'illumination globale ... etc. Nous détaillerons donc ici l'état de l'art nécessaire à la compréhension des principaux concepts manipulés et bien sûr les premiers résultats obtenus. Le but d'un tel projet est à la fois d'offrir aux utilisateurs une interface simple manipulant et exécutant des algorithmes ou des méthodes numériques complexes et aux programmeurs une importante quantité de sources librement réutilisables.

1. Introduction

L'algorithme de lancer de rayon ou *ray tracing* largement utilisé dans l'industrie et les rendus d'image hors ligne est bien connu pour sa capacité à fournir des images de qualité et physiquement réalistes. Néanmoins, la complexité et le temps de calcul des opérations élémentaires mises en jeu l'ont longtemps empêché d'être utilisé pour la génération d'images en temps réel. Récemment, des recherches ont cependant montré que des mises en œuvre efficaces sur les machines actuelles permettaient d'accélérer considérablement les calculs effectués au point d'obtenir des algorithmes d'affichage en lancer de rayon interactifs voire temps réel pour des scènes statiques ou dynamiques. En revanche, même si un grand nombre de chercheurs ont amélioré les algorithmes et les méthodes numériques utilisés, aucun n'a proposé une interface de programmation ouverte. Une équipe a cependant récemment proposé une bibliothèque de fonctions "OpenRT" [DWBS03] qui tend à standardiser les opérations nécessaires au lancer de rayon mais sans en fournir le code. Dans cet article, nous nous proposons d'une part de présenter les travaux que nous avons effectués afin de mettre en œuvre une version GPL d'OpenRT et d'autre part de réexposer une partie des algorithmes utilisés qui sont indispensables à la conception d'une telle bibliothèque de fonctions. Notre but est d'exposer les difficultés qui peuvent être rencontrées, les outils requis pour la réalisation d'un tel projet et les premiers résultats que nous avons obtenus.

2. Principe du lancer de rayon

Philippe Slussalek et al. [SSM*05] donnent une définition générale du lancer de rayon qui couvre l'ensemble des applications utilisées. Si l'on se donne un rayon, c'est à dire une origine et une direction,

et un ensemble de primitives (par exemple des triangles), le lancer de rayon consiste à trouver le sous-ensemble de ces primitives qui est intersecté par ce rayon. Au contraire, l'affichage par rasterisation utilisé par exemple par OpenGL consiste à trouver l'ensemble des rayons intersectés par une primitive. Considérant cette opération élémentaire, une application de rendu par lancer de rayon peut facilement être découpée en cinq opérations élémentaires : la génération des rayons, la segmentation des données, l'intersection rayon/primitive, les calculs d'éclairement et la gestion du tampon d'image. Concevoir une bibliothèque de calcul en lancer de rayon consiste donc à fournir les outils réalisant ces cinq opérations. C'est ce que nous essayons de fournir avec GNU RT en suivant les choix de conception proposés par OpenRT.

3. GNU RT

GNU RT comme OpenRT a pour but de cacher la complexité liée à la mise en œuvre d'une application en lancer de rayon. Dans cette section, nous présentons donc un exemple simple utilisant l'interface de programmation OpenRT que nous avons mis en œuvre.

La figure 1 présente un exemple de code en C créant la scène complète. Comme on peut le voir, la syntaxe et la philosophie sont très proches des concepts proposés par OpenGL. Un shader est d'abord chargé et créé, ses paramètres sont ensuite récupérés. Un objet est enfin créé d'une manière assez proche des listes d'affichage d'OpenGL puis instancié et positionné.

Comme on peut le constater, OpenRT emploie des concepts simples et classiques (en particulier ceux d'OpenGL) pour créer les objets, gérer des shaders ...etc. La gestion interne est néanmoins complètement différente puisque contrairement à OpenGL, l'API doit gérer le graphe de scène complet et doit maintenir une liste de tous les objets créés, de leurs instances, de tous les shaders ...etc. Nous allons donc maintenant détailler la plupart de algorithmes mis en œuvre pour la réalisation d'une telle bibliothèque de fonctions.

```
/* Creation d'un shader */
shaderClassId =
    rtGenNewShaderClass("sample", "shader.so");
/* Le shader est instancié une fois */
Id = rtGenNewShader();
/* On récupère 3 paramètres de shader */
scolorId = rtParameterHandle("shader_color");
tcolorId = rtParameterHandle("triangle_color");
vcolorId = rtParameterHandle("vertex_color");
rtParameter3f(scolorId, 1.f, 0.f, 0.f);

/* Création d'un objet */
objId = rtGenObjects(1);
rtNewObject(objId, RT_COMPILE);
for(i = 0; i < tri_nb; i++) {
    /* Paramètre par triangle */
    rtParameter3fv(tcolorId, ... );
    for(j = 0; j < 3; j++) {
        rtBegin(RT_TRIANGLES);
            /* Paramètre par triangle */
            rtParameter3fv(vcolorId, ... );
            rtColor3fv( ... );
            rtVertex3fv( ... );
        rtEnd();
    }
}

/* Création d'une instance */
/* Matrice de modéle associée à l'instance */
rtMatrixMode(RT_MODELVIEW);
rtLoadIdentity();
rtScalef(10.f, 10.f, 10.f);
instId = rtInstantiateObject(objId);
```

Figure 1: *Création typique d'une scène avec OpenRT / GNU RT*

4. Diviser pour régner

Trouver l'intersection la plus proche le long d'un rayon ("rayon de visibilité") ou établir s'il y a intersection entre deux points ("rayon d'ombre") a longtemps été une opération très coûteuse. Plusieurs travaux ont permis d'accélérer grandement ces opérations.

Utiliser des kD-trees

V. Havran a montré durant sa thèse [Hav00] que la structure de kD-tree (introduite par Bentley [Ben75]) était la plus appropriée pour accélérer les calculs de traversée. Le kD-tree qui consiste à subdiviser récursivement l'espace par un plan (voir Figure 2) donne des résultats supérieurs à ceux obtenus avec les octrees, les hiérarchies de volumes ou les grilles régulières. Il gère tous les types de scènes (comme la théière dans un stade), peut être très compact et sa traversée est peu coûteuse (une soustraction, une multiplication et deux comparaisons). Pour ces raisons, il est couramment choisi pour accélérer les calculs d'intersection dans la plupart des outils par lancer de rayon. C'est celui que nous avons choisi, en particulier pour gérer les scènes *statiques*. Néanmoins, plusieurs précautions doivent être prises pour rendre l'algorithme le plus efficace possible.

Utiliser une heuristique de coût

La partition spatiale est choisie dans l'objectif de limiter le nombre d'opérations effectuées pour connaître l'objet visible le long d'un rayon. Le choix du plan de coupe est en particulier primordial. Ainsi, couper l'espace en deux parties égales est particulièrement in-

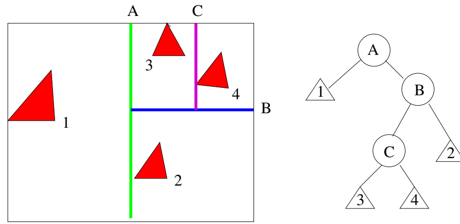

Figure 2: ***Un exemple de kD-tree*** *: Un kD-tree coupe récursivement l'espace en deux par des plans ou des hyperplans*

adapté. Un modèle de coût associée à une heuristique simple dite "Surface Area Heuristic" ou SAH [MB90] est néanmoins tout à fait adapté à ce problème. Ce modèle repose sur deux hypothèses :

- Les rayons sont uniformément distribués dans l'espace.
- Les coûts d'intersection et de traversée de nœud sont connus.

Si un rayon a intersecté un nœud donné, la probabilité d'intersection P de ce rayon avec un fils du nœud est alors proportionnelle au rapport des aires de leur surface : $SA(\text{fils})$ et $SA(\text{nœud})$:

$$P(\text{fils}|\text{nœud}) = \frac{SA(\text{fils})}{SA(\text{nœud})}$$

Le coût $C(\text{nœud})$ d'un nœud est donc :

$$\begin{aligned} C(\text{nœud}) &= C_{parcours}(\text{nœud}) \\ &+ P(\text{fils}_{gauche}|\text{nœud}) * C(\text{fils}_{gauche}) \\ &+ P(\text{fils}_{droit}|\text{nœud}) * C(\text{fils}_{droit}) \end{aligned}$$

On peut donc calculer le coût total d'un ensemble d'arbres afin de ne garder que le meilleur. Cependant, comme la recherche de la solution optimale en temps raisonnable ne semble pas possible, on utilise une heuristique gloutonne choisissant la coupe qui minimise localement le coût $C(n\oe ud)$. Elle consiste simplement à considérer que les fils du nœud courant sont des feuilles et ainsi à approcher le coût de l'arbre à chaque nœud sans évaluer le coût des coupes suivantes. D'où :

$$\begin{aligned} C(n\oe ud) &\approx C_{parcours}(\text{nœud}) \\ &+ P(\text{fils}_{gauche}|\text{nœud}) * C_{feuille}(\text{fils}_{gauche}) \\ &+ P(\text{fils}_{droit}|\text{nœud}) * C_{feuille}(\text{fils}_{droit}) \end{aligned}$$

L'heuristique SAH fournit en même temps un critère d'arrêt élégant. Si le coût de la coupe la moins coûteuse est plus grand que le coût engendré par l'absence de coupe, on transforme le nœud en feuille. Plusieurs algorithmes de construction existent avec des complexités variant de $O(N^2)$ à $O(N\ log\ N)$. La plupart des techniques efficaces repose sur la technique du "Perfect Split". Elle consiste à calculer les six plans de la boîte englobante de chaque primitive restreint à la boîte englobante du nœud. Selon la mise en œuvre, la complexité en fonction du nombre de triangles varie entre $O(N^2)$ et $O(N\ log\ N)$. Pour GNU RT, nous avons mis en œuvre une technique en $O(N\ log\ N)$. Tous les détails concernant les algorithmes de construction peuvent être trouvés dans [WH06].

Parcourir un kD-tree

Une fois la scène partitionnée par un kD-tree, le calcul d'inter d'un rayon avec la scène se réduit à un parcours en profondeur de l'arbre. Le rayon est associé à un intervalle d'abscisses $[0, \infty]$ restreint à la boîte englobante du kD-tree. Cet intervalle sera mis à jour à chaque visite d'un nœud et sera noté $[t_{min}, t_{max}]$. Pour chaque nœud visité, il suffit de calculer l'abscisse d de l'intersection du rayon et du plan de coupe du nœud. En fonction de la position de cette intersection par rapport à l'intervalle $[t_{min}, t_{max}]$, il est possible de déterminer quel est le prochain nœud à traverser. (voir Figure 3)

Si le rayon se trouve entièrement à gauche ou à droite du plan de coupe ($t_{max} < d$ ou $d < t_{min}$), il suffit d'explorer le seul fils correspondant. Sinon, il est nécessaire de parcourir le fils proche (associé à l'intervalle $[t_{min}, d]$) puis s'il y a n'y a pas eu intersection, le fils loin (associé à l'intervalle $[d, t_{max}]$). L'efficacité du kD-tree repose également sur le fait que l'on peut parcourir les nœuds et les feuilles dans l'ordre du parcours le long du rayon. Ainsi, dès que l'intersection la plus proche a été trouvée dans une feuille, on peut arrêter le parcours. Plusieurs variantes de parcours sont proposées dans [Hav00].

Intersection rayon / triangle

Une fois l'intersection d'un rayon et d'une feuille réussie, il reste à déterminer l'intersection rayon / triangle. Les algorithmes les plus utilisés sont celui de Badouel [Bad90] et celui de Möller-Trumbore [MT97]. Le premier est très rapide mais nécessite un précalcul pour tous les triangles. Le second est plus lent mais travaille directement sur les données. Comme nous considérons pour l'instant seulement les scènes statiques, nous avons préféré utiliser une version modifiée de l'algorithme de Badouel. Dans GNU RT, les scènes statiques sont donc segmentées à l'aide d'un kD-tree et les triangles sont prétraités pour être intersectés à l'aide de l'algorithme de Badouel.

5. Exploiter la cohérence des données

Exploiter la cohérence des données efficacement, c'est à dire regrouper les données proches (en mémoire, spatialement ... etc) permet d'obtenir un

	Nombre de triangles	Temps de calcul
Office	23K	$0,065$ s
Theater	120K	$3,0$ s
Conference	195K	$3,9$ s
Buddha	1100K	$18,5$ s

Table 1: ***Temps de construction pour construire un kD-tree avec GNU RT***. *La complexité du calcul en fonction du nombre N de triangles est $O(NlogN)$.*

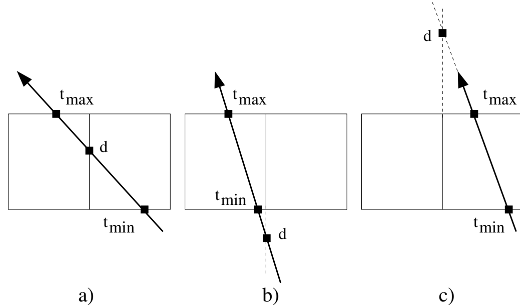

Figure 3: *3 cas possibles lors du parcours d'un kD-tree. a) $t_{min} \leq d \leq t_{max}$, on parcours le fils proche puis le fils loin. b) $d \leq t_{min}$, on ne parcours que le fils loin. c) $t_{max} \leq d$, on ne parcours que le fils proche.*

gain considérable de temps de calculs dans une application en lancer de rayon.

5.1. Utiliser les instructions vectorielles

Les architectures modernes proposent aujourd'hui des instructions vectorielles ou *SIMD* (Single Input Multiple Data) permettant d'effectuer une opération (Single Input) sur plusieurs données (Multiple Data. Par exemple, les instructions SSE disponibles sur les processeurs x86, permettent d'effectuer des opérations arithmétiques ou logiques sur 2 doubles ou 4 flottants à la fois. Utiliser les instructions SSE revient donc à effectuer des opérations sur des mots de 128 bits qui contiennent 2 doubles ou 4 flottants. Lors de l'apparition des instructions vectorielles, la seule manière d'utiliser du code SIMD était d'écrire les routines en assembleur. Cela rendait donc la programmation difficile puisque la manipulation des registres devait se faire manuellement. Sur les compilateurs récents (ICC ou GCC) existent à présent des routines dites "intrinsic" qui génèrent exactement l'instruction assembleur correspondante. L'intérêt des "intrinsics" est que la manipulation des registres est laissée au compilateur et seule celle des variables revient au programmeur

```
/* Vecteur 3d: version "non SIMD" */
struct vec3 {
    float x, y, z;
};
/* Produit scalaire "non SIMD" */
static inline float vec3dot(
    const struct vec3 *v0,
    const struct vec3 *v1) {
    return v0->x * v1->x +
        v0->y * v1->y +
        v0->z * v1->z;
}

/* Vecteur 3d: version "SIMD" */
struct sse_vec3 {
    __m128 x, y, z;
};
/* Produit scalaire "SIMD" */
static inline __m128 sse_vec3dot(
    const struct sse_vec3 *v0,
    const struct sse_vec3 *v1) {
    return _mm_add_ps(
        _mm_add_ps(
            _mm_mul_ps(v0->x, v1->x),
            _mm_mul_ps(v0->y, v1->y)),
        _mm_mul_ps(v0->z, v1->z));
}
```

Figure 4: ***Exemple de code "SIMD" et "non SIMD"*** *: Paralléliser le code revient simplement à dupliquer chaque champ d'une structure et à remplacer les opérations arithmétiques et logiques standards par les "intrinsic" fournies par les compilateurs. Les branchements sont gérés par des masques de calculs.*

Principe

L'intérêt des instructions SIMD est qu'elles permettent de paralléliser le code à la condition de réorganiser les données. La figure 4 donne un exemple de structure (vecteur à 3 dimensions) adaptée au calcul SIMD et un exemple de fonction exécutant quatre produits scalaires simultanément. Il suffit de donc de dupliquer tous les champs atomiques d'une structure à l'aide des types prédéfinis (comme __m128 par exemple).

Application au lancer de rayon

I. Wald a été le premier à utiliser les instructions SIMD pour le lancer de rayon [WBWS01]. Sa méthode consistait à regrouper quatre rayons caméra au sein d'une structure modifiée qui consiste simplement à utiliser un "sse_vector" pour l'origine et un autre pour la direction (cf figure 4). Une fois créée, le parcours du kD-tree, l'intersection rayon / triangle, la génération des rayons d'ombre et réfléchis se font à l'aide des instructions SIMD. On

	Nombre de triangles	Rayons/s (SSE)	Rayon/s (non-SSE)
Office	23K	5000K	2300K
Theater	120K	2800K	1500K
Conference	195K	4400K	2200K
Buddha	1100K	3800K	1900K

Table 2: ***Nombre de rayons par seconde.*** *L'accélération apportée par les instructions SIMD varie de 1.8 à 2.2 ce qui reste cohérent avec les résultats que l'on peut trouver dans la littérature*

doit noter que les branchements posent problème. En effet, si lors d'un test, les quatre rayons ont des comportements différents et prennent des branches différentes (comme lors d'un test d'intersection), il est nécessaire de masquer les calculs effectués par les rayons non concernés par la branche en cours. Pour plus de détails, la thèse d'I. Wald est une très bonne référence [Wal04]. A propos des instructions vectorielles et leur utilisation en lancer de rayon, la thèse de C. Benthin est incontournable [Ben06]. Tout le code d'intersection, d'éclairement, de parcours de structures sera également disponible librement dans le code de GNU RT.

Résultats
Comme indiqué par le tableau 2, l'accélération fournie par l'utilisation des instruction SIMD est de l'ordre de 2 à 2,5. Comparé à l'accélération théorique maximale de 4, le gain peut sembler faible. Néanmoins, la gestion des masques, le chargement / déchargement des données au format SIMD ralentit l'exécution des algorithmes. Malgré tout, les résultats obtenus et présentés par nos concurrents sont du même ordre de grandeur. De plus, nous n'avons pas encore essayé de regrouper les rayons dans des ensemble plus larges. Cela permettrait de réduire relativement les coûts lié au chargement / déchargement des données.

5.2. Augmenter la compacité de la structure

Comme la traversée du kD-tree est une opération élémentaire très souvent répétée pendant un rendu, la qualité de sa mise en œuvre est primordiale. Avec un peu de soin, la taille occupée par la structure peut être grandement réduite afin de diminuer la latence des accès et la place mémoire occupée. Même si un nœud ou une feuille est une boîte englobante, il n'est pas nécessaire de la stocker lors du parcours de l'arbre. Les données nécessaires au parcours se réduisent à :

- Un champ indiquant si c'est un nœud ou une feuille
- Le plan de coupe (si c'est un nœud) et le l'axe principal du plan

```
/* Mise en oeuvre compacte et portable du kD-tree
 * Le kD-tree est juste un tableau de kd_data
 */
struct kd_data {
    /* Indice vers le fils gauche
     * égal à 0 si c'est une feuille
     */
    uint32_t left;
    union {
        /* Axe du plan encodé dans value */
        float value;
        /* Indice dans le tableau de
         * données à intersecter
         */
        uint32_t elt_index;
    };
};
```

Figure 5: *Une structure compacte et portable de kD-tree. Il occupe seulement 8 octets sur toutes les architectures.*

- Les pointeurs vers les deux fils (si c'est un nœud)
- Un pointeur (ou un indice) vers le tableau d'indices de primitives (si c'est une feuille)

Sur une machine 32 bits standard, une mise en œuvre naïve respectant un alignement des données sur 8 octets requièrerait donc 16 octets. Néanmoins, on peut stocker les fils consécutivement afin de n'avoir qu'un pointeur et encoder l'axe principal du plan dans les bits de poids faible de la valeur du plan. On a alors une structure requérant seulement 8 octets. D'autres variantes sont possibles. On peut par exemple utiliser les bits de poids faibles des pointeurs. Pour les machines 64 bits, il suffit de remplacer les pointeurs par des indices dans un tableau (voir Figure 5). Le tableau 1 présente les résultats obtenus pour la construction de kD-tree. La complexité en $O(N\ log\ N)$ contraint la méthode à gérer seulement les scènes statiques. Pour les scènes dynamiques, d'autres approches doivent être considérées (voir Partie 6)

5.3. Multi Level Ray Tracing

A. Reshetov et al. ont récemment présenté une technique simple permettant d'obtenir un affichage temps réel par lancer de rayon [RSH05]. Cette technique exploite la cohérence spatiale des rayons et plus particulièrement les rayons issus de la caméra. L'idée est de subdiviser l'image en blocs et de générer une pyramide de rayons pour chaque bloc. Puis, au lieu d'obliger chaque rayon (ou un ensemble de rayons) à parcourir tout l'arbre, la pyramide est d'abord utilisée pour trouver à l'ensemble des rayons qu'elle contient un nouveau point d'entrée commun plus bas dans l'arbre. En utilisant les instructions SIMD, le parcours effectué

par la pyramide est encore accéléré et de nombreux calculs rayon par rayon sont efficacement évités.

Néanmoins, les optimisations faites requièrent que les rayons appartiennent au même huitième d'espace et qu'ils aient une origine commune ce qui contraint l'efficacité de la méthode à la gestion des rayons d'ombre ou des rayons de visibilité issus de la caméra. De plus, pour des scènes contenant beaucoup de petits objets, la méthode s'avère bien moins efficace.

6. Gérer les scènes dynamiques

Comme indiqué par le tableau 1, les temps de construction d'un kD-tree sont importants et inappropriés pour l'animation. C'est pourquoi, des recherches ont été menées récemment pour mettre au point des techniques et des structures efficaces pour gérer le dynamisme des scènes affichées.

Mouvements rigides et pseudo-rigides
La première idée est de restreindre le problème à un ensemble de mouvements rigides. Une solution simple et intuitive a été proposée par Wald et al. [WBS03a]. Elle consiste à calculer un kD-tree pour chaque partie rigide (comme le bras d'un robot par exemple) puis de construire un kD-tree regroupant toutes les parties rigides de la scène. Ce kD-tree de "plus haut niveau" peut être reconstruit interactivement voire en temps réel pour chaque image si la scène contient un nombre restreint de parties rigides (de l'ordre de mille par exemple). Cette technique permet donc d'*instancier* un grand nombre de fois le même objet avec une faible occupation mémoire. C'est la technique que nous avons pour l'instant mise en œuvre dans GNU RT.

Plus récemment, J. Günther et al. [GFW*06] ont proposé une méthode, les "fuzzy" kD-trees, pour gérer une plus grande classe d'animations. Un ensemble de positions clés (ou "key frames") définissant une animation est d'abord segmenté afin de trouver automatiquement les parties cohérentes se déplaçant quasi-rigidement. Lors de l'animation et du rendu, un kD-tree des parties quasi-rigides est reconstruit à chaque image. Au sein de chaque partie, un précalcul fournit également une boîte englobante pour chacun de ces triangles. Cette boîte a pour propriété d'englober le triangle quelque soit l'instant considéré dans l'animation. Il suffit alors de précalculer pour chaque partie rigide, un kD-tree sur les boîtes englobantes des triangles, ce kD-tree restant valable quelque soit l'instant considéré. La même équipe [GFSS06] a également présenté une variante de cette méthode pour l'animation par squelette (ou "skinning"). Comme on ne connaît pas a priori l'animation, les boîtes englobantes par triangle sont obtenues par un échantillonnage important des paramètres d'animation.

Reconstruction "from scratch"
I. Wald et al. ont récemment proposé une méthode "brute force" pour afficher des scènes de complexité moyenne [WIK*06]. L'idée est de recalculer complètement une grille ou une multi-grille à chaque image. Grâce à des idées proches du Multi Level Ray Tracing [RSH05], c'est à dire effectuer le parcours d'une pyramide de rayons avant de commencer l'intersection rayon / structure accélératrice, ils obtiennent un affichage interactif sur des scènes complètement animées d'une complexité variant de 15 000 à 180 000 triangles.

Utilisation d'englobants
La partition spatiale n'est pas la seule partition possible. L'utilisation d'une hiérarchie de volumes englobants ou BVHs ("Bounding Volume Hierarchies" introduites par J. Clark en 1976 [Cla76]) permet de segmenter les objets de la scène plutôt que son volume. I. Wald et al ont proposé l'utilisation de boîtes alignées sur les axes (souvent appelées "AABBs") afin de gérer des scènes dynamiques [WBS06]. L'intérêt des volumes englobants est qu'ils peuvent aisément être mis à jour lors de l'animation. Il suffit simplement de recalculer la position des triangles et les volumes englobants sans modifier l'indexation de données. A l'aide des instructions SIMD et de méthodes proches du Multi Level Ray Tracing, I. Wald et al. obtiennent des temps de calcul intéractifs voire temps réel sur des scènes modérément complexes.

Le problème des BVHs est que l'espace occupé par les nœuds et les feuilles est important (32 octets pour la mise en œuvre de I. Wald et al.). De plus, le parcours est relativement coûteux à cause de l'algorithme d'intersection rayon / boîte englobante. Pour répondre à ce problème, des chercheurs ont mis au point une structure hybride entre kD-tree et BVH, le BkD-tree [WMS06] ou Bounding Interval Hierarchy (BIH) [WK06]. L'idée est d'utiliser un arbre binaire où chaque nœud contient deux plans alignés sur le même axe. Le premier indique que les données contenues par le fils gauche sont à sa gauche, le second indiquant que les données contenues par le fils droit sont à sa droite. Grâce à ces techniques, le parcours de la structure est rapide, et la taille de l'arbre est limité. Enfin, comme pour les BVHs, l'arbre peut être mis à jour en temps linéaire durant une animation.

Et pour GNU RT ?
L'intérêt d'une interface de programmation est d'exposer simplement des phénomènes ou des

algorithmes difficiles. La méthode des "fuzzy" kD-trees [GFSS06] [GFW*06] nécessite de connaître le mouvement ou de pouvoir l'évaluer facilement et ne peux gérer des animations comme des explosions ou des mouvements de particules. La méthode par grille régulière proposée par Wald [WIK*06] reste très lente sans l'utilisation de méthodes type MLRT et ne peux gérer que des scènes modérément complexes.

Les méthodes par BVHs, BkD-trees ou BIHs sont plus intéressantes dans le cadre d'une interface de programmation. On peut ainsi construire la structure accélératrice lors de la création de l'objet et mettre à jour les boîtes englobantes ou les positions des plans sans modifier l'indexation des données durant l'animation. Enfin, les BkD-trees et les BIHs sont certainement plus intéressants que les BVHs car plus compacts et plus rapides à parcourir. Cependant, nous avons remarqué que la gestion des espaces vides n'est pas correctement faite par ces structures ce qui les rend particulièrement lentes pour certaines scènes (la théière dans un stade par exemple). De plus, aucune méthode n'a été proposée pour gérer plusieurs instances d'un même objet dynamique dans une scène. La seule méthode en l'état actuel serait de dupliquer l'arbre autant de fois qu'il y a d'instances de cet objet provoquant une consommation de ressource mémoire problématique. Nos investigations vont donc porter à la fois sur la gestion des cas pathologiques soulignés et sur un mécanisme automatique permettant d'instancier des personnages animés de manière non rigide. Enfin, il est à noter que plusieurs chercheurs ont récemment considérablement accéléré les constructions de kD-trees en remplaçant l'évaluation exhaustive des coûts de toutes les coupes par une méthode d'échantillonage adaptatif [HMS06]. Cette méthode pourrait s'avérer intéressante pour l'affichage intéractif de scènes dynamiques.

7. Applications au calcul d'éclairement

La principale utilisation d'une interface de programmation pour le lancer de rayon est la synthèse d'images réalistes. Néanmoins, comme les méthodes que nous avons décrites nécessitent de conserver la cohérence spatiale des blocs de rayons traités, il est nécessaire de trouver une classe de méthodes numériques adaptée à une telle organisation des calculs.

Instant Radiosity
En simplifiant, Instant Radiosity proposée par A. Keller [Kel97] est une méthode en deux passes qui consiste à propager des particules d'energie ou Virtual Point Lights ("VPLs") depuis les sources lumineuses et à collecter l'energie qu'elles émettent sur l'écran. Comme les sources sont ponctuelles, cette dernière passe revient simplement à lancer des rayons d'ombre entre un pixel et un VPL. L'intérêt d'Instant Radiosity est d'assurer la cohérence des rayons lors de la deuxième passe de calcul. Comme l'ont montré I. Wald et al dans [WKB*02], elle se prête donc très bien au lancer de rayon cohérent ou à des techniques type MLRT. Néanmoins, Instant Radiosity comme toute méthode de Monte-Carlo pose des problèmes de variance ou plus généralement de vitesse de convergence. Plusieurs travaux ont été proposés. I. Wald et al. proposent d'associer aux sources physiques une distribution cumulée afin de déterminer en fonction de la position de la caméra quelles sources choisir pour commencer la propagation des VPLs [WBS03b]. B. Segovia et al. ont proposé de générer les VPLs depuis la caméra également et de rééchantilloner toutes les sources virtuelles produites par une méthode de sampling / resampling (Bidirectional Instant Radiosity [SIMP06]).

Metropolis Instant Radiosity
Même si un développement important a été réalisé pour améliorer les méthodes basées sur Instant Radiosity, un pas décisif peut être franchi en terme de qualité d'échantillonage. Bidirectional Instant Radiosity souffre en effet de problèmes liés à la difficulté de mise en œuvre de la génération des sources issues de la caméra et du biais difficilement contrôlable dû à l'estimation de densité effectuée pour ces mêmes sources. Nous comptons donc proposer une méthode pour effectuer le calcul d'illumination globale dans une scène en utilisant un échantillonneur basé Monte-Carlo Markov Chain (dits "MCMC") et de méthodes proches de celles proposées par E. Veach dans [VG97]. Le but est de fournir une schéma numérique stable et simple pour réaliser rapidement et simplement l'intégration numérique quelque soit le domaine d'intégration.

8. Conclusion

Nous avons donc présenté les travaux que nous menons actuellement pour mettre en œuvre une version GPL d'OpenRT. A l'heure où est rédigé cet article, la partie géométrique de l'API (soumission de triangles, des paramètres de shaders ... etc) ainsi que les calculs d'intersection et de parcours d'arbre sont entièrement réalisés. Nous travaillons aujourd'hui sur une interface de shader plus évoluée que celle proposée par OpenRT. En effet, les prototypes proposés empêchent actuellement de paralléliser les calculs d'éclairement. Une solution intéressante serait à notre sens de créer un langage de "shading" spécifique qui rendrait l'écriture de shaders très intuitive et cacherait à l'utilisateur toutes les optimisations faites (calculs SSE, multithreading

...etc). Quoi qu'il en soit, GNU RT reste une aventure passionnante avec de nombreuses pistes de recherche à suivre. Nous espérons donc pouvoir faire profiter à la communauté de tous les travaux que nous effectuerons et du code qui sera produit.

References

[Bad90] BADOUEL D. : An efficient ray-polygon intersection. *Graphics gems* (1990), 390–393.

[Ben75] BENTLEY J. L. : Multidimensional binary search trees used for associative searching. *Commun. ACM* (1975), 509–517.

[Ben06] BENTHIN C. : *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Computer Graphics Group, Saarland University, 2006.

[Cla76] CLARK J. H. : Hierarchical geometric models for visible surface algorithms. *Commun. ACM* (1976), 547–554.

[DWBS03] DIETRICH A., WALD I., BENTHIN C., SLUSALLEK P. : The OpenRT Application Programming Interface – Towards A Common API for Interactive Ray Tracing. In *Proceedings of the 2003 OpenSG Symposium* (2003), pp. 23–31.

[GFSS06] GÜNTHER J., FRIEDRICH H., SEIDEL H.-P., SLUSALLEK P. : Interactive ray tracing of skinned animations. *The Visual Computer* (2006).

[GFW*06] GÜNTHER J., FRIEDRICH H., WALD I., SEIDEL H.-P., SLUSALLEK P. : Ray tracing animated scenes using motion decomposition. *Computer Graphics Forum* (2006), 517–525.

[Hav00] HAVRAN V. : *Heuristic Ray Shooting Algorithms*. Phd thesis, Czech Technical University, 2000.

[HMS06] HUNT W., MARK W. R., STOLL G. : Fast kd-tree construction with an adaptive error-bounded heuristic. In *2006 IEEE Symposium on Interactive Ray Tracing* (2006).

[Kel97] KELLER A. : Instant radiosity. In *SIGGRAPH '97 (Proceedings of the 24th annual conference on Computer graphics and interactive techniques)* (1997), pp. 49–56.

[MB90] MACDONALD D. J., BOOTH K. S. : Heuristics for ray tracing using space subdivision. *Visual Computer* (1990).

[MT97] MÖLLER T., TRUMBORE B. : Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools* (1997).

[RSH05] RESHETOV A., SOUPIKOV A., HURLEY J. : Multi-level ray tracing algorithm. *ACM Transaction on Graphics* (2005), 1176–1185.

[SIMP06] SEGOVIA B., IEHL J.-C., MITANCHEY R., PÉROCHE B. : Bidirectional instant radiosity. In *Proceedings of the 17th Eurographics Workshop on Rendering, to appear* (2006).

[SSM*05] SLUSALLEK P., SHIRLEY P., MARK B., STOLL G., WALD I. : Siggraph 2005 course 41 : Introduction to real time ray tracing, 2005.

[VG97] VEACH E., GUIBAS L. : Metropolis light transport. *SIGGRAPH '97 (Proceedings of the 24th annual conference on Computer graphics and interactive techniques)* (1997), 65–76.

[Wal04] WALD I. : *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004.

[WBS03a] WALD I., BENTHIN C., SLUSALLEK P. : Distributed Interactive Ray Tracing of Dynamic Scenes. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)* (2003).

[WBS03b] WALD I., BENTHIN C., SLUSALLEK P. : Interactive Global Illumination in Complex and Highly Occluded Environments. In *Proceedings of the 14th Eurographics Workshop on Rendering* (2003).

[WBS06] WALD I., BOULOS S., SHIRLEY P. : Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies (revised version). *ACM Transactions on Graphics* (2006).

[WBWS01] WALD I., BENTHIN C., WAGNER M., SLUSALLEK P. : Interactive rendering with coherent ray tracing. *Computer Graphics Forum (Proceedings of Eurographics 2001)* (2001).

[WH06] WALD I., HAVRAN V. : On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006).

[WIK*06] WALD I., IZE T., KENSLER A., KNOLL A., PARKER S. G. : Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics* (2006).

[WK06] WÄCHTER C., KELLER A. : Instant raytracing : The bouding interval hierarchy. In *Rendering Techniques 2006 (Proceedings of the 17th Eurographics Symposium on Rendering)* (2006).

[WKB*02] WALD I., KOLLIG T., BENTHIN C., KELLER A., SLUSALLEK P. : Interactive global illumination using fast ray tracing. In *Proceedings of the 13th Eurographics Workshop on Rendering* (2002).

[WMS06] WOOP S., MARMITT G., SLUSALLEK P. : B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *Proceedings of Graphics Hardware* (2006).