Real Time Global Illumination

Audric Schiltknecht

Université de Montréal

27 Novembre 2009

Outline

Plan

Objectifs

- Illumination globale

Objectifs

- Illumination globale
- Gestion de multiples effets (caustiques, rebonds multiples, reflexions *glossy*, etc.)

Objectifs

- Illumination globale
- Gestion de multiples effets (caustiques, rebonds multiples, reflexions *glossy*, etc.)
- Rendu sur GPU

Objectifs

- Illumination globale
- Gestion de multiples effets (caustiques, rebonds multiples, reflexions *glossy*, etc.)
- Rendu sur GPU
- Optimisé pour du temps réel (interactions de l'utilisateur : changement de la géométrie, lumière(s), matériaux, etc)

État de l'art

- Algorithmes déjà existants, mais calculs sur CPU $\Longrightarrow$ temps de rendu long

État de l'art

- Algorithmes déjà existants, mais calculs sur CPU $\Longrightarrow$ temps de rendu long
- Algorithmes sur GPU existants, mais seulement pour certains effets :
  - [SA07] : global shadow effects
  - [Kel97], [DS05] : éclairage indirect avec un seul rebond

État de l'art

- Algorithmes déjà existants, mais calculs sur CPU $\Longrightarrow$ temps de rendu long
- Algorithmes sur GPU existants, mais seulement pour certains effets :
  - [SA07] : global shadow effects
  - [Kel97], [DS05] : éclairage indirect avec un seul rebond
- [ZHWG08] : caustiques interactives et reflexion spéculaire. Le plus « complet »

Plan

Préliminaires

Définition

Regroupement de points par partitionnement de l'espace

Les kd-Tree sont des BSPs particuliers.

Préliminaires

Définition

Regroupement de points par partitionnement de l'espace

Les kd-Tree sont des BSPs particuliers.

- Arbres binaires

Préliminaires

Définition

Regroupement de points par partitionnement de l'espace

Les kd-Tree sont des BSPs particuliers.

- Arbres binaires
- Noeuds sont des k-points

Préliminaires

Définition

Regroupement de points par partitionnement de l'espace

Les kd-Tree sont des BSPs particuliers.

- Arbres binaires
- Noeuds sont des k-points
- Alignés avec les axes

Préliminaires

Définition

Regroupement de points par partitionnement de l'espace

Les kd-Tree sont des BSPs particuliers.

- Arbres binaires
- Noeuds sont des k-points
- Alignés avec les axes
- À chaque branchement, génération d'un hyperplan , qui divise l'espace en deux

Idée de base

- Direction de l'hyperplan est fonction de la profondeur ($direction = depth \mod k$)
- Sélection du point *median* par rapport à l'hyperplan choisi
- Appel récursif sur chaque des deux sous-espaces ainsi crées.

Exemple de construction - Étape 1

Exemple

Soit la liste suivante à ranger dans un *kd-Tree* :
$[(2,3),(5,4),(9,6),(4,7),(8,1),(7,2)]$

Exemple de construction - Étape 1

Exemple

Soit la liste suivante à ranger dans un *kd-Tree* :
$[(2,3),(5,4),(9,6),(4,7),(8,1),(7,2)]$

Axe X

Exemple de construction - Étape 1

Exemple

Soit la liste suivante à ranger dans un *kd-Tree* :
$[(2,3),(5,4),(9,6),(4,7),(8,1),(7,2)]$

Axe X

Liste triée :
$[(2,3),(4,7),(5,4),(7,2),(8,1),(9,6)]$

Exemple de construction - Étape 1

Exemple

Soit la liste suivante à ranger dans un *kd-Tree* :
$[(2,3),(5,4),(9,6),(4,7),(8,1),(7,2)]$

Axe X

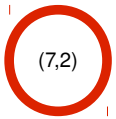
Liste triée :
$[(2,3),(4,7),(5,4),(7,2),(8,1),(9,6)]$

Point médian $(7,2)$

Exemple de construction - Étape 1

Exemple

Soit la liste suivante à ranger dans un *kd-Tree* :
$[(2,3),(5,4),(9,6),(4,7),(8,1),(7,2)]$



Axe X

Liste triée :
$[(2,3),(4,7),(5,4),(7,2),(8,1),(9,6)]$

Point médian $(7,2)$

Sous-listes
Gauche $[(2,3),(4,7),(5,4)]$
Droite $[(8,1),(9,6)]$

Exemple de constructioni - Étape 2

Exemple

Soit la liste suivante à ranger dans un *kd-Tree* :
$[(2,3),(5,4),(9,6),(4,7),(8,1),(7,2)]$

Fils gauche : $[(2,3),(4,7),(5,4)]$.

Exemple de constructioni - Étape 2

Exemple

Soit la liste suivante à ranger dans un *kd-Tree* :
$[(2,3),(5,4),(9,6),(4,7),(8,1),(7,2)]$

Fils gauche : $[(2,3),(4,7),(5,4)]$.

Axe Y

Exemple de constructioni - Étape 2

Exemple

Soit la liste suivante à ranger dans un *kd-Tree* :
$[(2,3),(5,4),(9,6),(4,7),(8,1),(7,2)]$

Fils gauche : $[(2,3),(4,7),(5,4)]$.

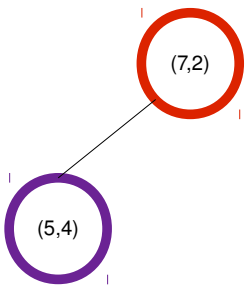
Axe Y

Liste triée $[(2,3),(5,4),(4,7)]$

Exemple de constructioni - Étape 2

Exemple

Soit la liste suivante à ranger dans un *kd-Tree* :
$[(2,3),(5,4),(9,6),(4,7),(8,1),(7,2)]$



Fils gauche : $[(2,3),(4,7),(5,4)]$.

Axe Y

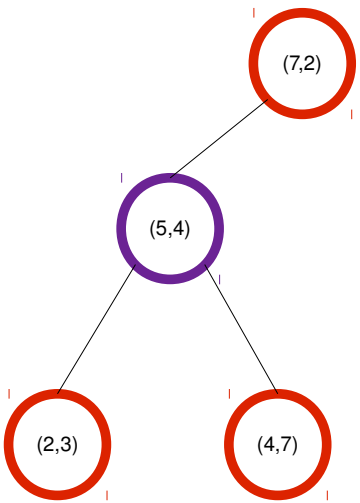
Liste triée $[(2,3),(5,4),(4,7)]$

Point médian $(5,4)$

Exemple de constructioni - Étape 2

Exemple

Soit la liste suivante à ranger dans un *kd-Tree* :
$[(2,3),(5,4),(9,6),(4,7),(8,1),(7,2)]$



Fils gauche : $[(2,3),(4,7),(5,4)]$.

Axe Y

Liste triée $[(2,3),(5,4),(4,7)]$

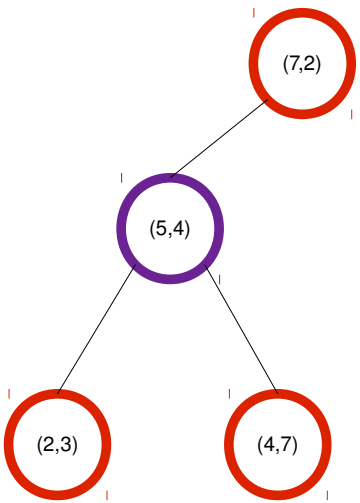
Point médian $(5,4)$

Sous-listes
Gauche $[(2,3)]$
Droite $[(4,7)]$

Exemple de construction - Étape 3

Exemple

Soit la liste suivante à ranger dans un *kd-Tree* :
$[(2,3),(5,4),(9,6),(4,7),(8,1),(7,2)]$

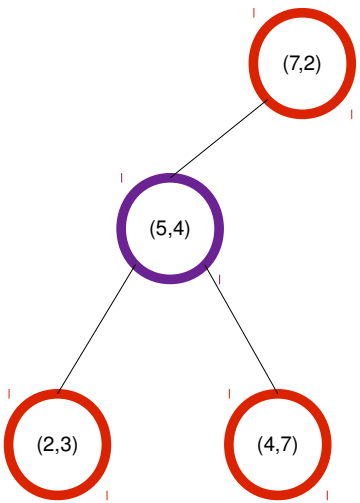


Fils droit : $[(8,1),(9,6)]$.

Exemple de construction - Étape 3

Exemple

Soit la liste suivante à ranger dans un *kd-Tree* :
$[(2,3),(5,4),(9,6),(4,7),(8,1),(7,2)]$



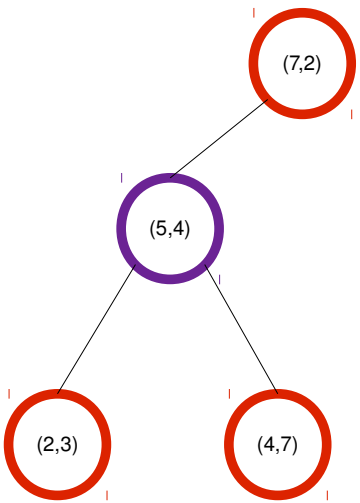
Fils droit : $[(8,1),(9,6)]$.

Axe Y

Exemple de construction - Étape 3

Exemple

Soit la liste suivante à ranger dans un *kd-Tree* :
$[(2,3),(5,4),(9,6),(4,7),(8,1),(7,2)]$



Fils droit : $[(8,1),(9,6)]$.

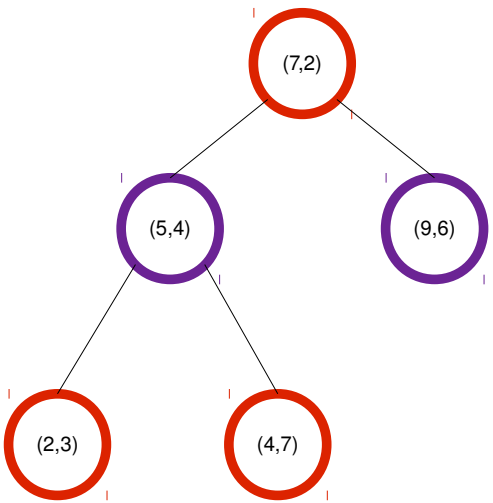
Axe Y

Liste triée $[(8,1),(9,6)]$

Exemple de construction - Étape 3

Exemple

Soit la liste suivante à ranger dans un *kd-Tree* :
$[(2,3),(5,4),(9,6),(4,7),(8,1),(7,2)]$



Fils droit : $[(8,1),(9,6)]$.

Axe Y

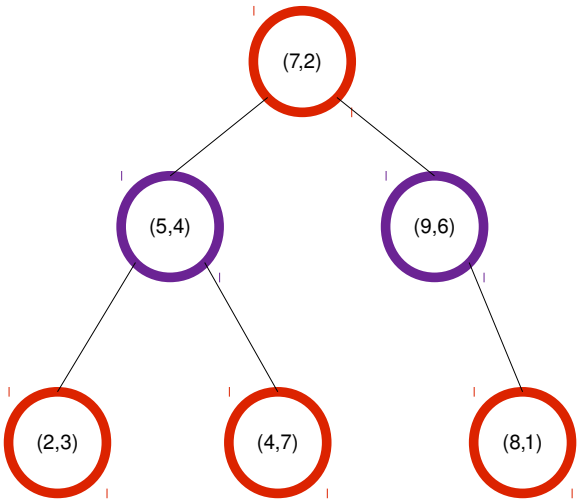
Liste triée $[(8,1),(9,6)]$

Point médian $(9,6)$

Exemple de construction - Étape 3

Exemple

Soit la liste suivante à ranger dans un *kd-Tree* :
$[(2,3),(5,4),(9,6),(4,7),(8,1),(7,2)]$



Fils droit : $[(8,1),(9,6)]$.

Axe Y

Liste triée $[(8,1),(9,6)]$

Point médian $(9,6)$

Sous-listes Gauche $[(8,1)]$

Construction de *kd-Tree* sur le GPU

Objectif

Améliorer la construction de *kd-Tree* pour l'effectuer sur le **GPU** et profiter de son fort parallélisme.

- Construction en *BFS* : adapté au GPU
- Différenciation noeuds :

Large Proche de la racine
Petit Proche des feuilles

$\rightarrow$ seuil défini par l'utilisateur (nombre de triangles dans le noeud)

Principe

On utilise la description de [ZHWG08], utilisée pour la construction de *kd-Tree* contenant des triangles, en vu de réaliser un *ray-tracer*.

1. Evaluer le coût de chaque plan de découpe
2. Choisir celui dont le coût est le plus faible
3. Repartir les triangles entre les deux sous-plans

Algorithme

Trois phases séquentiels :

1. Calcul des boîtes englobantes des triangles
2. Traitement des noeuds larges : décomposition des noeuds en *chunk* pour optimiser le calcul en parallèle
3. Traitement des noeuds petits : le *BFS* permet de paralléliser facilement le traitement de ces noeuds

Large Nodes

On applique l'algorithme suivant à une liste de *large nodes* (la racine pour l'initialisation) :

Large Nodes

On applique l'algorithme suivant à une liste de *large nodes* (la racine pour l'initialisation) :

- Groupement des triangles en *chunk* de taille fixe

Large Nodes

On applique l'algorithme suivant à une liste de *large nodes* (la racine pour l'initialisation) :

- Groupement des triangles en *chunk* de taille fixe
- Pour chaque *chunk* (en parallèle) :
  - Calcul de la boîte englobante du *chunk*

Large Nodes

On applique l'algorithme suivant à une liste de *large nodes* (la racine pour l'initialisation) :

- Groupement des triangles en *chunk* de taille fixe
- Pour chaque *chunk* (en parallèle) :
  - Calcul de la boîte englobante du *chunk*
- Calcul de la boite englobante du noeud

Large Nodes

On applique l'algorithme suivant à une liste de *large nodes* (la racine pour l'initialisation) :

- Groupement des triangles en *chunk* de taille fixe
- Pour chaque *chunk* (en parallèle) :
  - Calcul de la boîte englobante du *chunk*
- Calcul de la boite englobante du noeud
- Pour chacun des noeuds en entrée (en parallèle) :

Large Nodes

On applique l'algorithme suivant à une liste de *large nodes* (la racine pour l'initialisation) :

- Groupement des triangles en *chunk* de taille fixe
- Pour chaque *chunk* (en parallèle) :
  - Calcul de la boîte englobante du *chunk*
- Calcul de la boite englobante du noeud
- Pour chacun des noeuds en entrée (en parallèle) :
  - Suppression de l'espace vide (par rapport à un seuil défini)

Large Nodes

On applique l'algorithme suivant à une liste de *large nodes* (la racine pour l'initialisation) :

- Groupement des triangles en *chunk* de taille fixe
- Pour chaque *chunk* (en parallèle) :
  - Calcul de la boîte englobante du *chunk*
- Calcul de la boite englobante du noeud
- Pour chacun des noeuds en entrée (en parallèle) :
  - Suppression de l'espace vide (par rapport à un seuil défini)
  - *Split* suivant le plan médian (orienté suivant le plus grand axe) et ajout à la liste des suivants

Large Nodes

On applique l'algorithme suivant à une liste de *large nodes* (la racine pour l'initialisation) :

- Groupement des triangles en *chunk* de taille fixe
- Pour chaque *chunk* (en parallèle) :
  - Calcul de la boîte englobante du *chunk*
- Calcul de la boite englobante du noeud
- Pour chacun des noeuds en entrée (en parallèle) :
  - Suppression de l'espace vide (par rapport à un seuil défini)
  - *Split* suivant le plan médian (orienté suivant le plus grand axe) et ajout à la liste des suivants
- Calcul du nombre de triangle dans les noeuds crées (tri *large*/*small*)

Large Nodes

On applique l'algorithme suivant à une liste de *large nodes* (la racine pour l'initialisation) :

- Groupement des triangles en *chunk* de taille fixe
- Pour chaque *chunk* (en parallèle) :
  - Calcul de la boîte englobante du *chunk*
- Calcul de la boite englobante du noeud
- Pour chacun des noeuds en entrée (en parallèle) :
  - Suppression de l'espace vide (par rapport à un seuil défini)
  - *Split* suivant le plan médian (orienté suivant le plus grand axe) et ajout à la liste des suivants
- Calcul du nombre de triangle dans les noeuds crées (tri *large*/*small*)

Itération sur suivants tant que tous les noeuds ne sont pas considérés comme « petits ».

Small Nodes

Les petits noeuds sont ceux obtenus à la sortie de l'algorithme précédant.
L'algorithme se décompose en deux phases :

Small Nodes

Les petits noeuds sont ceux obtenus à la sortie de l'algorithme précédant.
L'algorithme se décompose en deux phases :

1. *Préprocessing* : Trouve tous les plans de coupure potentiels, et création des listes de triangles se trouvant dans chacun des sous-plans ainsi crées

Small Nodes

Les petits noeuds sont ceux obtenus à la sortie de l'algorithme précédant.
L'algorithme se décompose en deux phases :

1. *Préprocessing* : Trouve tous les plans de coupure potentiels, et création des listes de triangles se trouvant dans chacun des sous-plans ainsi crées
2. Division effective

Coût du split

Utilisation d'une fonction d'estimation du coût de split.

Surface Area Heuristic

$$\mathrm{SAH}(x) = C_{ts} + \frac{C_L(x)A_L(x)}{A} + \frac{C_R(x)A_R(x)}{A}$$

avec :

$C_{ts} =$ coût de la traversée du noeud (constante)
$C_i(x) =$ coût du fils i de x (gauche ou droite)
$A_i(x) =$ surface du fils i de x
$A =$ surface du noeud

On prendra souvent $C_i(x) =$ nombre de triangles dans le fils i de x.

Split des *small nodes*

Cet algorithme prend en entrée une liste de noeud à diviser.

Split des *small nodes*

Cet algorithme prend en entrée une liste de noeud à diviser.

- Pour chacun des noeuds n en entrée (en parallèle)

Split des *small nodes*

Cet algorithme prend en entrée une liste de noeud à diviser.

- Pour chacun des noeuds n en entrée (en parallèle)
  - $S_0 \leftarrow$ nombres de noeuds dans n (coût du noeud).

Split des *small nodes*

Cet algorithme prend en entrée une liste de noeud à diviser.

- Pour chacun des noeuds n en entrée (en parallèle)
  - $S_0 \leftarrow$ nombres de noeuds dans n (coût du noeud).
  - Pour chacun des plans j candidats pour la division (en parallèle) :

Split des *small nodes*

Cet algorithme prend en entrée une liste de noeud à diviser.

- Pour chacun des noeuds n en entrée (en parallèle)
  - $S_0 \leftarrow$ nombres de noeuds dans n (coût du noeud).
  - Pour chacun des plans j candidats pour la division (en parallèle) :
    - Calculer $\mathrm{SAH}(j)$

Split des *small nodes*

Cet algorithme prend en entrée une liste de noeud à diviser.

- Pour chacun des noeuds n en entrée (en parallèle)
  - $S_0 \leftarrow$ nombres de noeuds dans n (coût du noeud).
  - Pour chacun des plans j candidats pour la division (en parallèle) :
    - Calculer $\mathrm{SAH}(j)$
- $p \leftarrow$ meilleur candidat (celui de coût le plus faible)

Split des *small nodes*

Cet algorithme prend en entrée une liste de noeud à diviser.

- Pour chacun des noeuds n en entrée (en parallèle)
  - $S_0 \leftarrow$ nombres de noeuds dans n (coût du noeud).
  - Pour chacun des plans j candidats pour la division (en parallèle) :
    - Calculer $\mathrm{SAH}(j)$
- $p \leftarrow$ meilleur candidat (celui de coût le plus faible)
- Si $p > S_0$:
  - Marquage de n comme feuille

Split des *small nodes*

Cet algorithme prend en entrée une liste de noeud à diviser.

- Pour chacun des noeuds n en entrée (en parallèle)
  - $S_0 \leftarrow$ nombres de noeuds dans n (coût du noeud).
  - Pour chacun des plans j candidats pour la division (en parallèle) :
    - Calculer $\mathrm{SAH}(j)$
- $p \leftarrow$ meilleur candidat (celui de coût le plus faible)
- Si $p > S_0$:
  - Marquage de n comme feuille
- Sinon :
  - Split suivant p
  - Ajout des fils à la liste des suivants

Split des *small nodes*

Cet algorithme prend en entrée une liste de noeud à diviser.

- Pour chacun des noeuds n en entrée (en parallèle)
  - $S_0 \leftarrow$ nombres de noeuds dans n (coût du noeud).
  - Pour chacun des plans j candidats pour la division (en parallèle) :
    - Calculer $\mathrm{SAH}(j)$
- $p \leftarrow$ meilleur candidat (celui de coût le plus faible)
- Si $p > S_0$:
  - Marquage de n comme feuille
- Sinon :
  - Split suivant p
  - Ajout des fils à la liste des suivants

On itère cet algorithme tant que tous les suivants n'ont pas été marqués comme feuille.

Adaptation pour le *Photon Mapping*

Il est facile d'adapter l'algorithme de construction de *kd-Tree* vu précédement pour pouvoir l'utiliser dans un *Photon Mapping*. Les modifications à apporter sont les suivantes :

- Fonction de coût du *split*
- Modification des valeurs du seuil pour l'espace vide, et du ratio *large*/*small*
- Plus nécessaire de calculer les boîtes englobantes avant de lancer l'algorithme

Fonction de coût du *split*

Vortex Volume Heuristic

$$\mathrm{VVH}(x) = C_{ts} + \frac{C_L(x)V(d_L(x)\pm R)}{V(d\pm R)} + \frac{C_R(x)V(d_R(x)\pm R)}{V(d\pm R)}$$

avec :

$C_{ts} =$ coût de la traversée du noeud (constante)
$C_i(x) =$ coût du fils i de x (gauche ou droite)
$V(d_i(x) \pm R) =$ volume du fils i de x étendu de R
$V(d \pm R) =$ volume du noeud
$R =$ rayon de recherche *KNN* estimé

On prendra souvent $C_i(x) =$ nombre de points dans le fils i de x.

Idée

- Recherche des k plus proches voisins (k n'est pas le même que dans *kd-Tree* !)

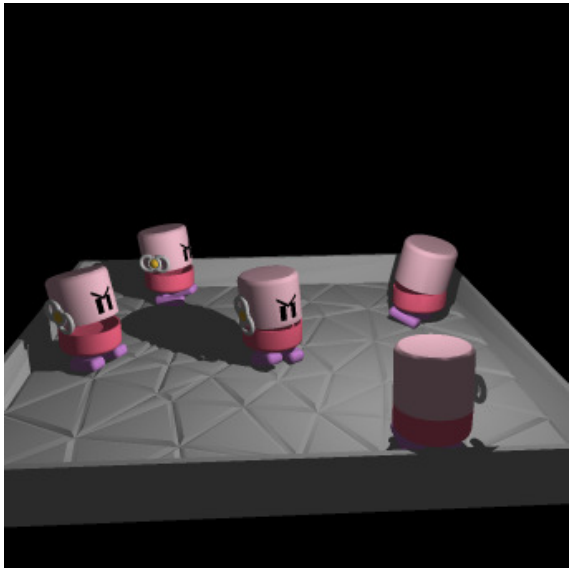
Idée

- Recherche des k plus proches voisins (k n'est pas le même que dans *kd-Tree* !)
- Parcourt de l'arbre jusqu'au point de recherche

Idée

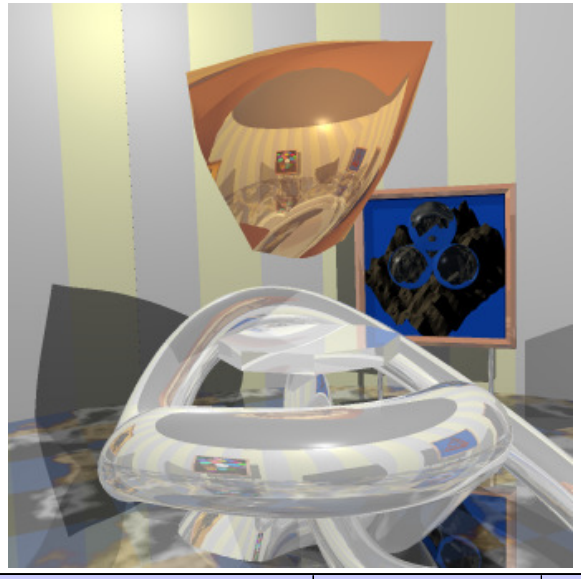
- Recherche des k plus proches voisins (k n'est pas le même que dans *kd-Tree* !)
- Parcourt de l'arbre jusqu'au point de recherche
- Remontée dans l'arbre et collecte des voisins jusqu'au nombre désiré

Comparaison des performances - GPU Ray Tracer



Scene	Nbre triangles	CPU	GPU	Speedup
Toys	11k	0.085s	0.012s	7

Comparaison des performances - GPU Ray Tracer



Scene	Nbre triangles	CPU	GPU	Speedup
Toys	11k	0.085s	0.012s	7
Museum	27k	0.108s	0.017s	6.3

Comparaison des performances - GPU Ray Tracer

Scene	Nbre triangles	CPU	GPU	Speedup
Toys	11k	0.085s	0.012s	7
Museum	27k	0.108s	0.017s	6.3
Robots	71k	0.487s	0.039s	12.5

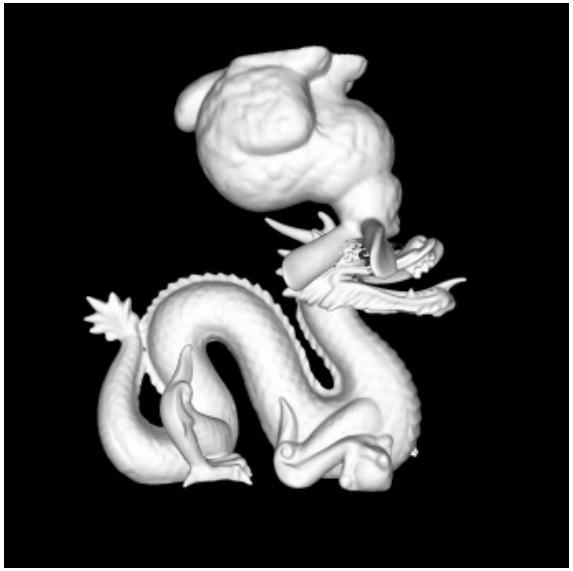
Comparaison des performances - GPU Ray Tracer

Scene	Nbre triangles	CPU	GPU	Speedup
Toys	11k	0.085s	0.012s	7
Museum	27k	0.108s	0.017s	6.3
Robots	71k	0.487s	0.039s	12.5
Kitchen	111k	0.559s	0.053s	10.5

Comparaison des performances - GPU Ray Tracer

Scene	Nbre triangles	CPU	GPU	Speedup
Toys	11k	0.085s	0.012s	7
Museum	27k	0.108s	0.017s	6.3
Robots	71k	0.487s	0.039s	12.5
Kitchen	111k	0.559s	0.053s	10.5
Fairy Forest	172k	1.226s	0.077s	16

Comparaison des performances - GPU Ray Tracer



Scene	Nbre triangles	CPU	GPU	Speedup
Toys	11k	0.085s	0.012s	7
Museum	27k	0.108s	0.017s	6.3
Robots	71k	0.487s	0.039s	12.5
Kitchen	111k	0.559s	0.053s	10.5
Fairy Forest	172k	1.226s	0.077s	16
Dragon	252k	1.354s	0.093s	14.6

Comparaison des performances - GPU Photon Mapping

Scene	Nbre triangles	FPS	CPU	GPU	Speedup
Anneau	3k	12.2	0.081s	0.009s	9

Comparaison des performances - GPU Photon Mapping

Scene	Nbre triangles	FPS	CPU	GPU	Speedup
Anneau	3k	12.2	0.081s	0.009s	9
Champagne	19k	7.5	0.237s	0.017s	13.9

Plan

Équation du rendu

Équation de l'illumination [Kaj86]

$$L_o(x) = \rho(x) \int_{H^2} L_i(x, \vec{\omega_i})(\vec{\omega_i} \cdot \vec{n}) \mathrm{d}\vec{\omega_i} = \rho(x) E(x)$$

Avec :

$L_o(x) =$ radiance sortante au point x
$\rho(x) =$ BRDF au point x
$E(x) =$ irradiance au point x
$\vec{\omega_i} =$ direction incidente
$\vec{n} =$ normale
$L_i(x, \vec{\omega_i}) =$ radiance incidente au point x, de direction $\vec{\omega_i}$

En utilisant la technique de *Photon Mapping* ([Jen01]), on peut calculer le terme $E(x)$ en deux passes.

Photon mapping

- Lancer des photons depuis les sources lumineuses et les accumuler dans la *photon map*
- La radiance d'un point dans la scène (*hit point*) est fonction du nombre de photons dans son entourage

Inconvénients du *Photon Mapping*

1. Grand nombre de *Final Gather Rays* pour un rendu correct
2. Recherche des plus proches voisins longue

Sampling

Proposé dans [WZPB09] : réalisation d'un sampling :

Sampling

Proposé dans [WZPB09] : réalisation d'un sampling :

- des hits points (*Irradiance caching*, [WRC88])

Sampling

Proposé dans [WZPB09] : réalisation d'un sampling :

- des hits points (*Irradiance caching*, [WRC88])
- de la structure contenant les photons (*Illumination cut*, [WFA+05])

Pipeline

Pipeline

1. Construction d'un *kd-Tree* de la scène ([ZHWG08])

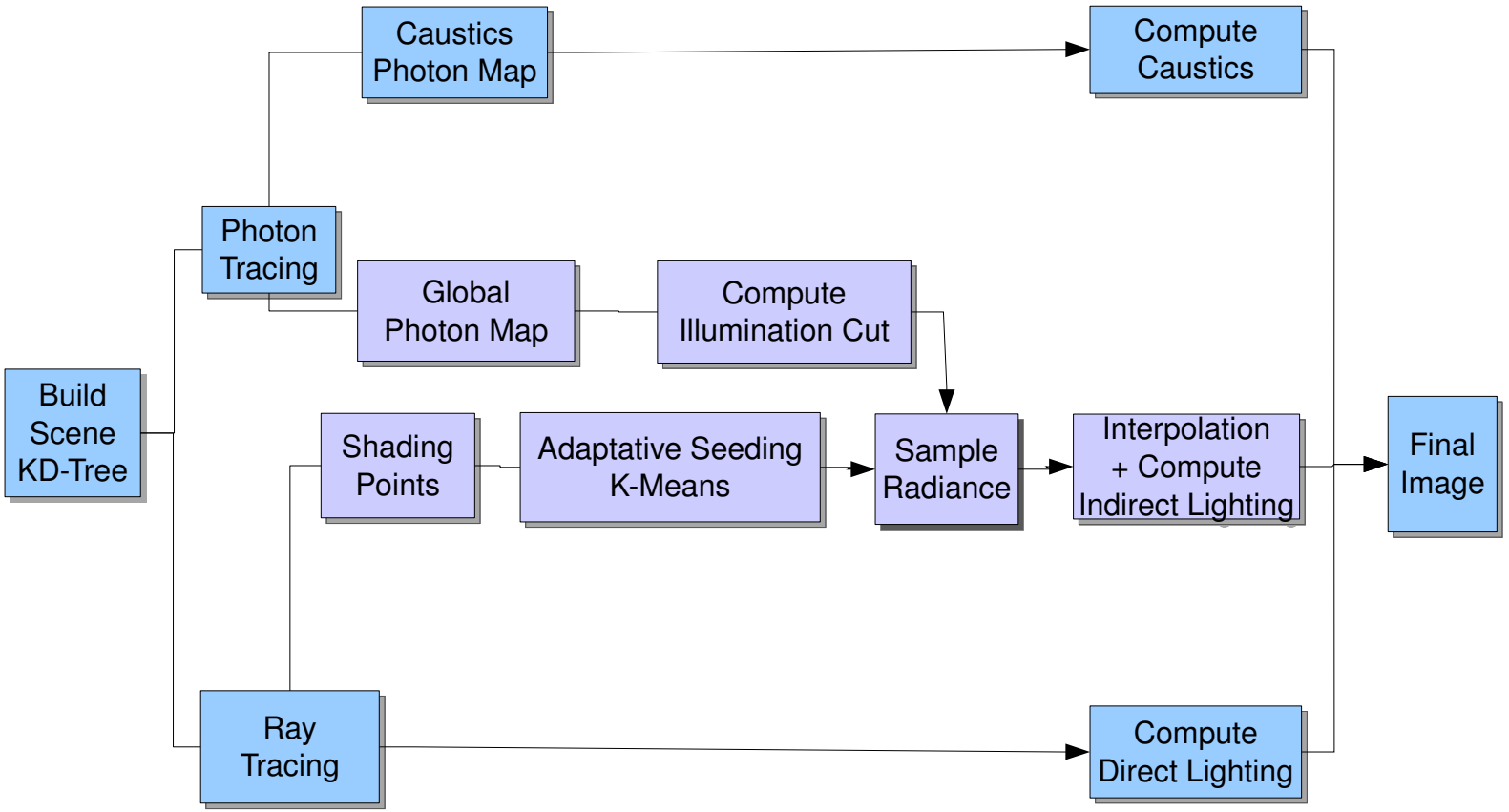
Pipeline

1. Construction d'un *kd-Tree* de la scène ([ZHWG08])
2. En parallèle :
   - Sélection d'un échantillonage de *hit points*
   - Sélection d'un échantillonage de la *photon map*.

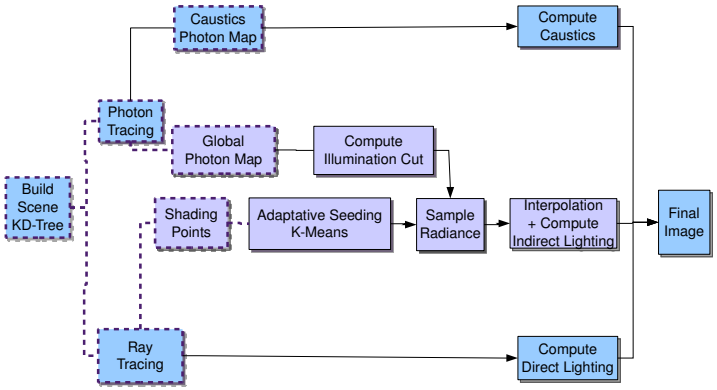
Pipeline

1. Construction d'un *kd-Tree* de la scène ([ZHWG08])
2. En parallèle :
   - Sélection d'un échantillonage de *hit points*
   - Sélection d'un échantillonage de la *photon map*.
3. Calcul de l'irradiance par interpolation

Pipeline - Graphique



kd-Tree

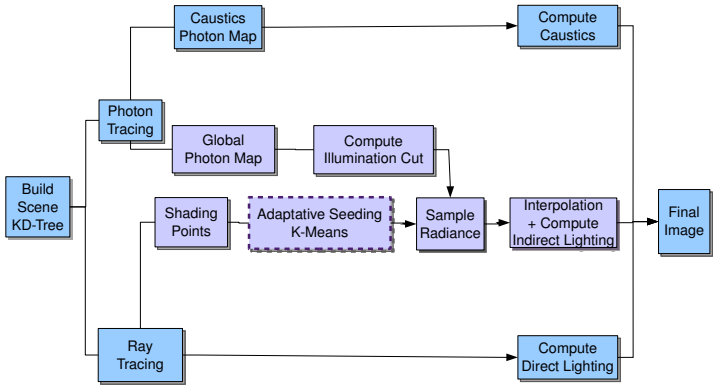


- Calcul de *kd-Tree* en utilisant les méthodes décrites dans [ZHWG08].
- Réalisation des *Photon Map* :

  Globale Dernier rebond du photon sur une surface non spéculaire

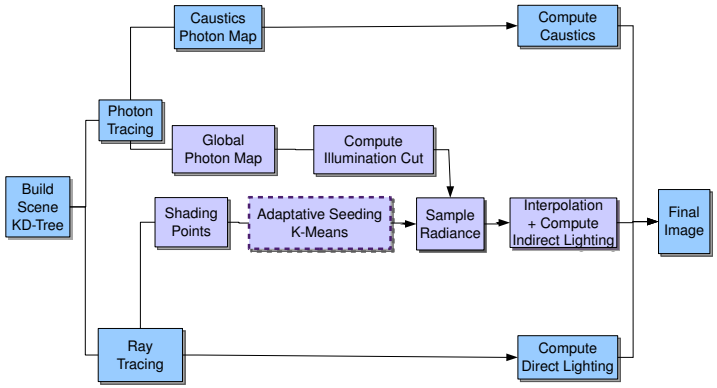
  Caustiques Dernier rebond du photon sur une surface spéculaire

Principe



[WRC88] décrit le principe de l'*irradiance caching* :

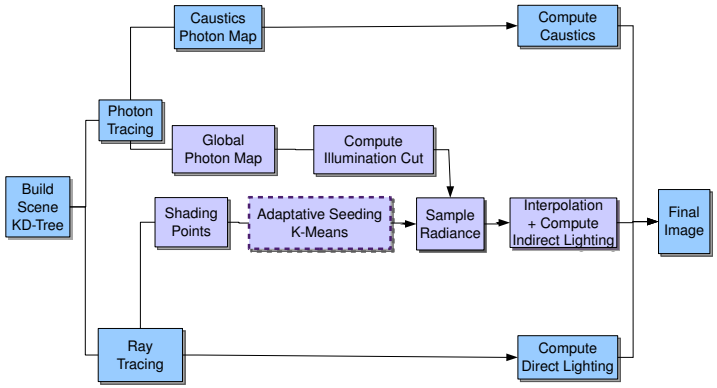
Principe



[WRC88] décrit le principe de l'*irradiance caching* :

- Illumination indirecte sur une surface varie doucement

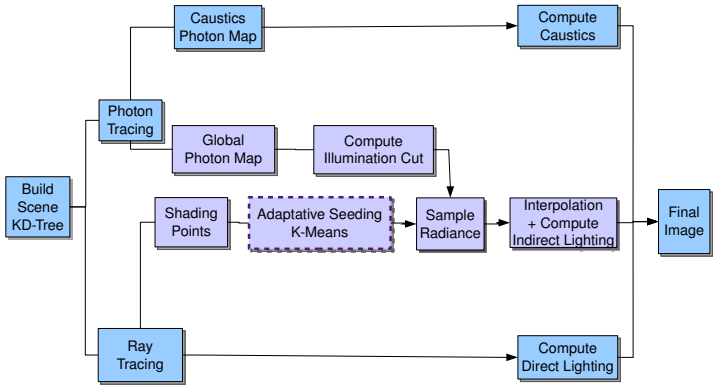
Principe



[WRC88] décrit le principe de l'*irradiance caching* :

- Illumination indirecte sur une surface varie doucement
- Création d'un cache d'irradiance

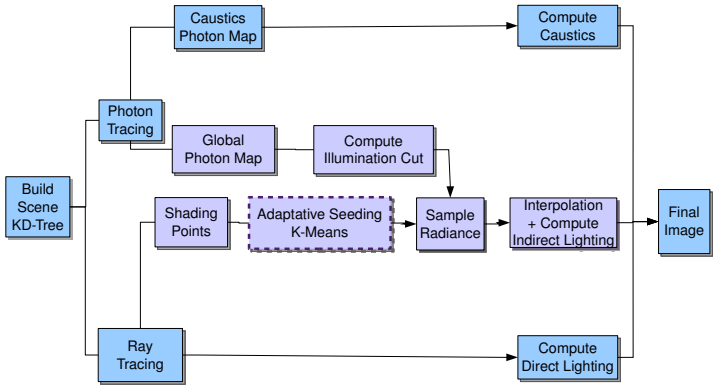
Principe



[WRC88] décrit le principe de l'*irradiance caching* :

- Illumination indirecte sur une surface varie doucement
- Création d'un cache d'irradiance
- Interpolation des valeurs du cache pour le calcul de l'irradiance d'un point

Principe



[WRC88] décrit le principe de l'*irradiance caching* :

- Illumination indirecte sur une surface varie doucement
- Création d'un cache d'irradiance
- Interpolation des valeurs du cache pour le calcul de l'irradiance d'un point

Problème

Choix du cache ?

Illumination Change term

« Prédir » les changements dans l'irradiance, en se basant sur la géométrie de la scène.

Fonction d'erreur

$$\epsilon_k(x_i) = \alpha\|x_i - x_k\| + \sqrt{2 - 2(\vec{n_i} \cdot \vec{n_k})}$$

avec :

$\alpha =$ facteur de poids

$n_i =$ normale de la surface au point x_i

Adaptive Seeding

Objectif

Distribution de k *seeds* dans la scène.

Adaptive Seeding

Objectif

Distribution de *k seeds* dans la scène.

- Découpage de la scène en *quadtree*

Adaptive Seeding

Objectif

Distribution de *k seeds* dans la scène.

- Découpage de la scène en *quadtree*
- Retrouver le *quadtree* contenant les *shading points*

Adaptive Seeding

Objectif

Distribution de *k seeds* dans la scène.

- Découpage de la scène en *quadtree*
- Retrouver le *quadtree* contenant les *shading points*
- Calcul de la normale et position moyenne de chacun des *quadtree* (en parallèle)

Adaptive Seeding

Objectif

Distribution de k *seeds* dans la scène.

- Découpage de la scène en *quadtree*
- Retrouver le *quadtree* contenant les *shading points*
- Calcul de la normale et position moyenne de chacun des *quadtree* (en parallèle)
- Pour chaque *shading point* x (en parallèle)
  - Calcul de $\epsilon_p(x)$, p étant le noeud contenant x dans le *quadtree*

Adaptive Seeding

Objectif

Distribution de k *seeds* dans la scène.

- Découpage de la scène en *quadtree*
- Retrouver le *quadtree* contenant les *shading points*
- Calcul de la normale et position moyenne de chacun des *quadtree* (en parallèle)
- Pour chaque *shading point* x (en parallèle)
  - Calcul de $\epsilon_p(x)$, p étant le noeud contenant x dans le *quadtree*
- Pour chaque noeud p dans le *quadtree* (en parallèle)
  - Calcul de l'erreur ϵ_p comme moyenne des $\epsilon_x, x \in p$

Adaptive Seeding

Objectif

Distribution de *k seeds* dans la scène.

- Découpage de la scène en *quadtree*
- Retrouver le *quadtree* contenant les *shading points*
- Calcul de la normale et position moyenne de chacun des *quadtree* (en parallèle)
- Pour chaque *shading point* x (en parallèle)
  - Calcul de $\epsilon_p(x)$, p étant le noeud contenant x dans le *quadtree*
- Pour chaque noeud p dans le *quadtree* (en parallèle)
  - Calcul de l'erreur ϵ_p comme moyenne des $\epsilon_x, x \in p$

$\Rightarrow$ Distribution des *k seeds* dans le *quadtree* en fonction de la valeur ϵ_q

Clustering

Répartition des points en *k clusters* : les points sont rangés dans le cluster le plus « proche », considéré au sens de la fonction d'erreur vu précédement.

Clustering

Répartition des points en *k clusters* : les points sont rangés dans le cluster le plus « proche », considéré au sens de la fonction d'erreur vu précédement.

- Définition des centres des clusters comme étant les points *seeds*

Clustering

Répartition des points en *k clusters* : les points sont rangés dans le cluster le plus « proche », considéré au sens de la fonction d'erreur vu précédement.

- Définition des centres des clusters comme étant les points *seeds*
- Tant qu'il n'y a pas convergence, ou nombre d'itérations est sous le seuil défini :

Clustering

Répartition des points en *k clusters* : les points sont rangés dans le cluster le plus « proche », considéré au sens de la fonction d'erreur vu précédement.

- Définition des centres des clusters comme étant les points *seeds*
- Tant qu'il n'y a pas convergence, ou nombre d'itérations est sous le seuil défini :
  - Construction d'un *kd-Tree* à partir de la liste des clusters

Clustering

Répartition des points en *k clusters* : les points sont rangés dans le cluster le plus « proche », considéré au sens de la fonction d'erreur vu précédement.

- Définition des centres des clusters comme étant les points *seeds*
- Tant qu'il n'y a pas convergence, ou nombre d'itérations est sous le seuil défini :
  - Construction d'un *kd-Tree* à partir de la liste des clusters
  - Pour chaque *shading point i* (en parallèle)
    - Recherche dans le *kd-Tree* du cluster le plus proche (au sens de la fonction d'erreur)
    - Ajout de *i* dans le cluster

Clustering

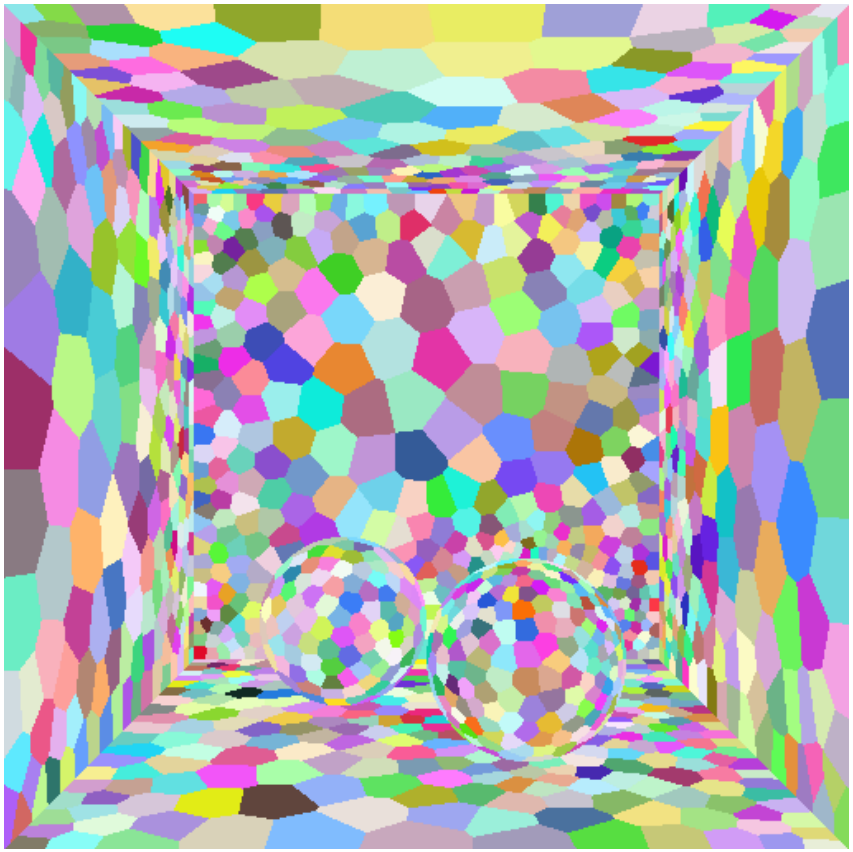
Répartition des points en *k clusters* : les points sont rangés dans le cluster le plus « proche », considéré au sens de la fonction d'erreur vu précédement.

- Définition des centres des clusters comme étant les points *seeds*
- Tant qu'il n'y a pas convergence, ou nombre d'itérations est sous le seuil défini :
  - Construction d'un *kd-Tree* à partir de la liste des clusters
  - Pour chaque *shading point i* (en parallèle)
    - Recherche dans le *kd-Tree* du cluster le plus proche (au sens de la fonction d'erreur)
    - Ajout de *i* dans le cluster
  - Calcul des nouveaux centres des clusters

Détermination des points d'échantillonage

- Pour chaque *cluster c* (en parallèle)
  - Trouver le *shading point* ayant la plus faible erreur avec le centre de *c*

Exemple



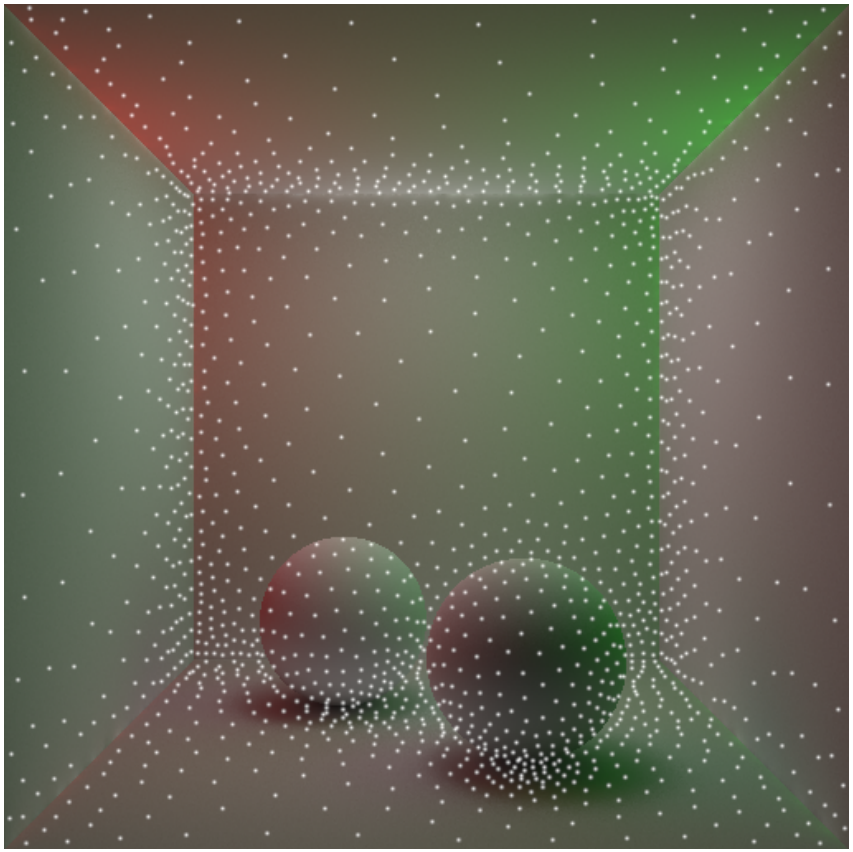
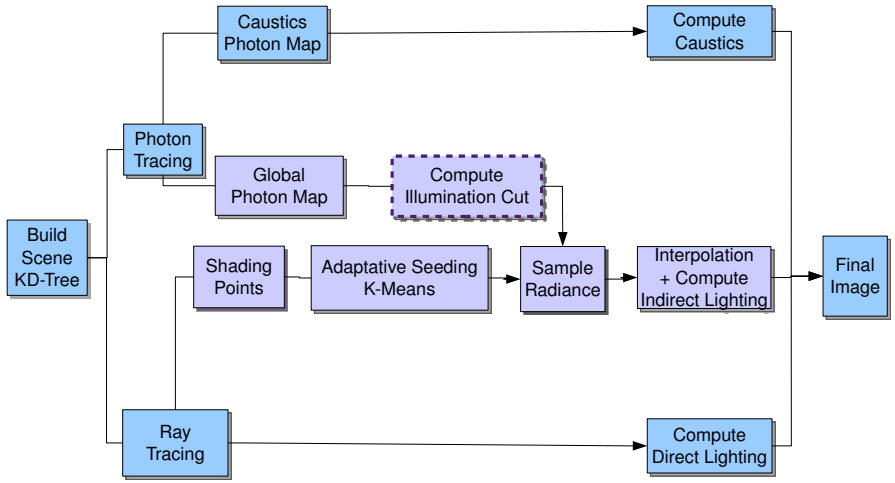


Figure: (a) montre les *clusters* (1600) ; (b) leur centre

Description



- Pour chaque point-échantillon calculé précédément, le calcul du champ d'irradiance se fait en lançant 250/500 *final gather rays*.
- Pour chacun de ces *FGR*, il faut parcourir le *kd-Tree* représentant la *Photon map*

Estimation de l'irradiance

Irradiance

$$\tilde{E}_p = \frac{\sum_{i \in p} (\vec{\omega_i} \cdot \vec{n_p}) \Phi_i}{r_p^2}$$

Avec :

$\tilde{E}_p =$ irradiance approximative du noeud p

$\vec{\omega_i} =$ direction incidente du photon

$\vec{n_p} =$ normale au centre de p

$\Phi_i =$ puissance du photon

$r_p =$ taille maximale de la boîte englobante de p

$\Rightarrow$ elle peut être calculée lors de la construction du *kd-Tree*

Illumination *Cut*

Parcourt en profondeur de l'arbre, et comparaison des valeurs de $\tilde{E}_p$ pour chaque noeud, à une référence :
$E_{min} = \text{mean}(\tilde{E}_p), p \in \{\text{niveau}(12\text{-}13 \text{ de l'arbre}) \}$

Illumination *Cut*

Parcourt en profondeur de l'arbre, et comparaison des valeurs de $\tilde{E}_p$ pour chaque noeud, à une référence :
$E_{min} = \text{mean}(\tilde{E}_p), p \in \{\text{niveau}(12\text{-}13 \text{ de l'arbre}) \}$

- Pour chaque noeud p du *kd-Tree* (en parallèle)

Illumination *Cut*

Parcourt en profondeur de l'arbre, et comparaison des valeurs de $\tilde{E}_p$ pour chaque noeud, à une référence :
$E_{min} = \text{mean}(\tilde{E}_p), p \in \{\text{niveau}(12\text{-}13 \text{ de l'arbre}) \}$

- Pour chaque noeud p du *kd-Tree* (en parallèle)
  - Si $\tilde{E}_p > E_{min}$:
    - p est ajouté au *cut*

Illumination *Cut*

Parcourt en profondeur de l'arbre, et comparaison des valeurs de $\tilde{E}_p$ pour chaque noeud, à une référence :
$E_{min} = \text{mean}(\tilde{E}_p), p \in \{\text{niveau}(12\text{-}13 \text{ de l'arbre}) \}$

- Pour chaque noeud p du *kd-Tree* (en parallèle)
  - Si $\tilde{E}_p > E_{min}$:
    - p est ajouté au *cut*
- Pour chaque noeud p du *cut* (en parallèle)

Illumination *Cut*

Parcourt en profondeur de l'arbre, et comparaison des valeurs de $\tilde{E}_p$ pour chaque noeud, à une référence :
$E_{min} = \text{mean}(\tilde{E}_p), p \in \{\text{niveau}(12\text{-}13 \text{ de l'arbre}) \}$

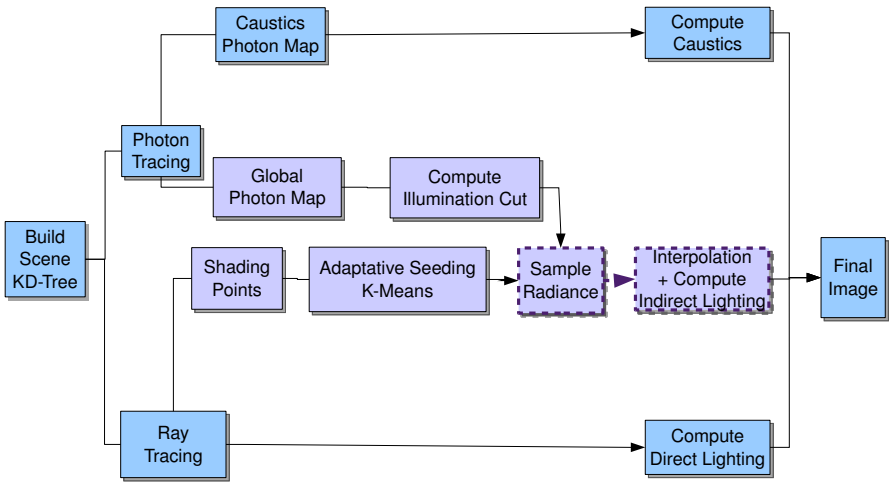
- Pour chaque noeud p du *kd-Tree* (en parallèle)
  - Si $\tilde{E}_p > E_{min}$:
    - p est ajouté au *cut*
- Pour chaque noeud p du *cut* (en parallèle)
  - Calcul de E_p, irradiance exacte au centre du noeud

Illumination *Cut*

Parcourt en profondeur de l'arbre, et comparaison des valeurs de $\tilde{E}_p$ pour chaque noeud, à une référence :
$E_{min} = \text{mean}(\tilde{E}_p), p \in \{\text{niveau}(12\text{-}13 \text{ de l'arbre}) \}$

- Pour chaque noeud p du *kd-Tree* (en parallèle)
  - Si $\tilde{E}_p > E_{min}$:
    - p est ajouté au *cut*
- Pour chaque noeud p du *cut* (en parallèle)
  - Calcul de E_p, irradiance exacte au centre du noeud
  - Si $\|\tilde{E}_p - E_p\| > \delta_E \approx 1.2\tilde{E}_p$:
    - Remplacer p par ses fils dans le *cut*

Résumé de la situation



On dispose de

- Un ensemble de points issus d'un *seeding* et *clustering* de *shading points*. Ils serviront à remplir le cache d'irradiance
- Un *illumination cut*, issu d'une coupe dans le *kd-Tree* contenant la *photon map*. Il servira à calculer l'irradiance pour les points du cache.

Sampling

Soit y un *hit point*. La radiance du *FGR* peut être évaluée à partir d'un ensemble de noeuds p_j avoisinants, par interpolation en utilisant la fonction suivante :

Fonction de poids

$$\omega_j = \mathrm{K}(\frac{\|p_j - y\|}{r(p_j)}) \max(0, \vec{n(p_j)} \cdot \vec{n_y})$$

Avec :

$\omega_j =$ poids de la radiance issue du noeud p_j

$r(p_j) =$ taille maximale de la boîte englobante de p

et :

$$\mathrm{K}(x) = \begin{cases} 1 - x^2 & \text{si } x \leq 1 \\ 0 & \text{sinon} \end{cases}$$

Interpolation

- Pour tous les points *seeds*, lancer des *FGR*, et calculer l'irradiance par interpolation sur l'*illumination cut*, en utilisant la fonction de poids précédement explicitée.

Interpolation

- Pour tous les points *seeds*, lancer des *FGR*, et calculer l'irradiance par interpolation sur l'*illumination cut*, en utilisant la fonction de poids précédement explicitée.
- Pour calculer l'irradiance d'un point x quelconque dans la scène, utiliser la formule d'interpolation de l'irradiance à partir du cache ([WRC88]) :

Interpolation

- Pour tous les points *seeds*, lancer des *FGR*, et calculer l'irradiance par interpolation sur l'*illumination cut*, en utilisant la fonction de poids précédement explicitée.
- Pour calculer l'irradiance d'un point x quelconque dans la scène, utiliser la formule d'interpolation de l'irradiance à partir du cache ([WRC88]) :

$$L(x) = \frac{\sum_{j \in S} \omega_j(x) L(j)}{\sum j \in S \omega_j(x)}$$

Interpolation

- Pour tous les points *seeds*, lancer des *FGR*, et calculer l'irradiance par interpolation sur l'*illumination cut*, en utilisant la fonction de poids précédement explicitée.
- Pour calculer l'irradiance d'un point x quelconque dans la scène, utiliser la formule d'interpolation de l'irradiance à partir du cache ([WRC88]) :

$$L(x) = \frac{\sum_{j \in S} \omega_j(x) L(j)}{\sum j \in S \omega_j(x)}$$

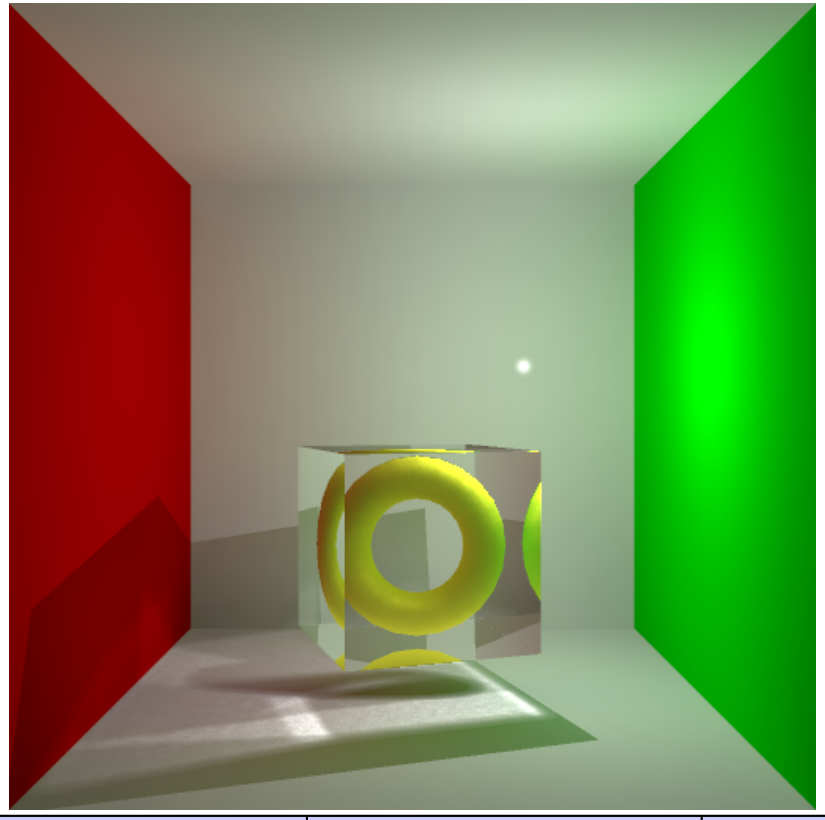
et :

$$\omega_j(x) = \frac{1}{\frac{\|x-j\|}{R_j} + \sqrt{2 - 2(\vec{n_x} \cdot \vec{n_j})}}$$

où S est un ensemble de points « caches » d'irradiance, R_j est la moyenne harmonique.

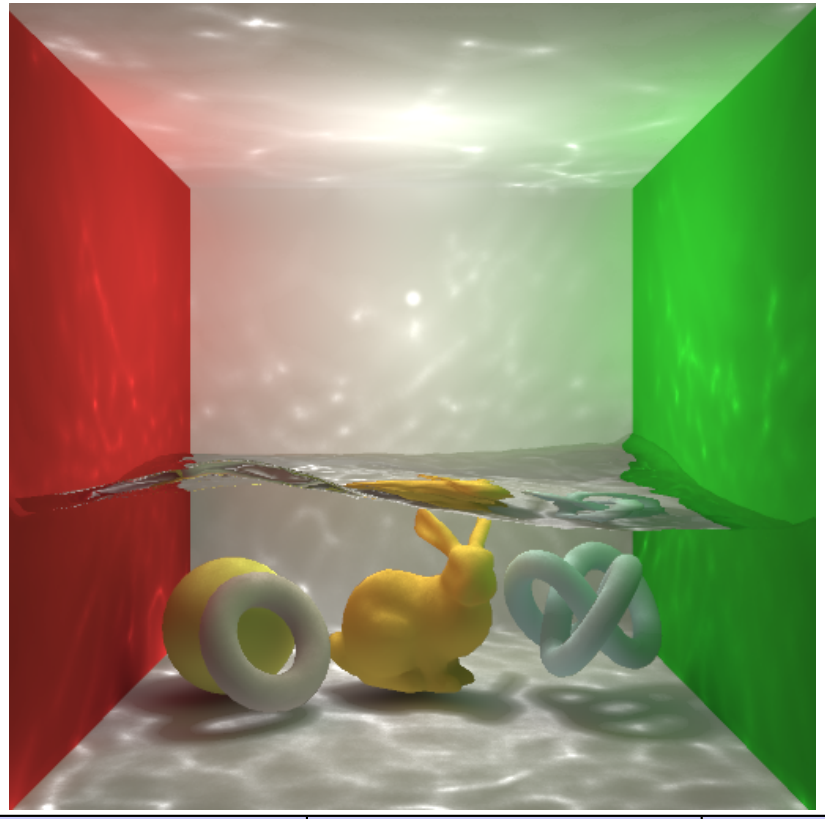
Plan

Résultats



Scène	Nbre triangles	G/C photons	Seeds	FPS
Box torus	0.6k	239k/100k	1.6k	4.2

Résultats



Scène	Nbre triangles	G/C photons	Seeds	FPS
Box torus	0.6k	239k/100k	1.6k	4.2
Box water	17k	268k/277k	1.8k	2.7

Résultats

Scène	Nbre triangles	G/C photons	Seeds	FPS
Box torus	0.6k	239k/100k	1.6k	4.2
Box water	17k	268k/277k	1.8k	2.7
Kitchen	21k	470k/109k	5k	1.5

Conclusion

+ Vitesse de rendu correcte
+ Rendu correct dans l'ensemble
+ Bonne utilisation du GPU
- Échantillonage : il est possible de manquer des détails (petites géométries)
- Recalcul des points d'échantillonage à chaque frame : peut introduire des incohérences temporelles
- Nombreux paramètres « ad-hoc » (mais pas de $\alpha = 0.7$!)

Questions ?

References I

Carsten Dachsbacher and Marc Stamminger.
Reflective shadow maps.
In *I3D '05 : Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 203–231, New York, NY, USA, 2005. ACM.

Wan. Jensen, Henrik.
Realistic Image Synthesis Using Photon Mapping.
2001.

J.T. Kajiya.
The rendering equation.
In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, page 150. ACM, 1986.

References II

Alexander. Keller.
Instant radiosity.
Citeseer, 1997.

Perumaal. Shanmugam and Okan. Arikan.
Hardware accelerated ambient occlusion techniques on GPUs.
In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, page 80. ACM, 2007.

B. Walter, S. Fernandez, A. Arbree, K. Bala, M. Donikian, and D.P. Greenberg.
Lightcuts : a scalable approach to illumination.
In *ACM SIGGRAPH 2005 Papers*, page 1107. ACM, 2005.

References III

G.J. Ward, F.M. Rubinstein, and R.D. Clear.
A ray tracing solution for diffuse interreflection.
ACM SIGGRAPH Computer Graphics, 22(4) :85–92, 1988.

Rui Wang, Kun Zhou, Minghao Pan, and Hujun Bao.
An efficient gpu-based approach for interactive global illumination.
In *SIGGRAPH '09 : ACM SIGGRAPH 2009 papers*, pages 1–8, New York, NY, USA, 2009. ACM.

Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo.
Real-time kd-tree construction on graphics hardware.
In *SIGGRAPH Asia '08 : ACM SIGGRAPH Asia 2008 papers*, pages 1–11, New York, NY, USA, 2008. ACM.