

Raytracing et Optimisation BVH

Le Prince Jules
Lycée Chateaubriand, Rennes
juleslprince@icloud.com

Résumé—L'imagerie de synthèse consiste à générer des images par ordinateurs. Dans notre TIPE nous nous sommes concentrés sur la réalisation d'images 3d photoréalistes par le biais du raytracing. Cette technique consiste à utiliser le principe de retour inverse de la lumière pour calculer efficacement les rebonds de rayons dans une scène. Il est alors nécessaire d'avoir une structure de donnée efficace permettant un calcul rapide d'intersections. La solution à ce problème est l'introduction d'une structure de Hiérarchie de Volumes Englobants (ou BVH pour Bounding Volume Hierarchy) dont l'idée est de calculer les intersections avec des volumes englobants grossiers afin de réduire le nombre d'intersections à tester avant de faire des calculs plus précis.

Mots clés—Ray Tracing, Bounding Volume Hierarchy, Imagerie de synthèse

Les types utilisés seront explicitement décrits par une syntaxe fortement inspirée de OCaml. Cependant ce ne sont pas exactement ceux utilisés lors de notre développement qui repose sur taichi [1], une librairie python permettant d'utiliser facilement les ressources GPU (nécessaire pour préserver un temps de calcul acceptable) au prix de structures très rigides qui ont obligées des adaptations techniques peu pertinentes à exhiber. Il a donc été choisi d'adopter dans cet article une structure compréhensible, restant tout de même le plus fidèle possible à notre projet.

I. DÉFINITIONS PRÉLIMINAIRES

Le raytracing est principalement centré sur des calculs d'intersections entre rayons et primitives (sphère, triangles...). Il convient alors dans un premier temps de définir ces primitives et de concevoir des algorithmes d'intersection efficaces.

A. Vecteurs

On définit un vecteur de \mathbb{R}^3 par

```
type Vec3 = float*float*float
```

et une couleur codée en RGB par

```
type Color = float*float*float
```

B. Rayon

Un rayon est défini par le type suivant :

```
type Rayon = {
    origin: Vec3,
    direction: Vec3 }
```

c'est à dire simplement une droite de \mathbb{R}^3 :

$$r(t) = \text{origin} + t \cdot \text{direction} \quad (1)$$

C. Sphère

a) *Type*: Une sphère est définie par le type :

```
type Sphere = {
    center: Vec3,
    radius: float }
```

b) *Intersection avec un rayon*: Prenons ray de type Rayon et sph de type Sphere . on note $\begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix}$ le vecteur de \mathbb{R}^3 ray.direction , $\begin{pmatrix} o_x \\ o_y \\ o_z \end{pmatrix}$ ray.origin et $\begin{pmatrix} c_x \\ c_y \\ c_z \end{pmatrix}$ sph.center On peut facilement calculer l'intersection entre notre rayon et notre sphère et résolvant l'équation en t de degré 2 suivante :

$$(o_x + d_x \cdot t - c_x)^2 + (o_y + d_y \cdot t - c_y)^2 + (o_z + d_z \cdot t - c_z)^2 = \text{sph.radius} \quad (2)$$

D. Triangle

a) *Type*: un triangle est défini par trois points de \mathbb{R}^3 :

```
type Triangle = {
    p1: Vec3,
    p2: Vec3,
    p3: Vec3 }
```

b) *Algorithme d'intersection*: On utilise ici l'algorithme décrit par Möller et Trumbore dans leur article [2] se basant sur un calcul de déterminant et ayant comme principaux avantages d'être rapide (grâce au pré-calcul d'opérations redondantes) et d'utiliser peu de mémoire.

E. Objet

Nos primitives peuvent avoir différents matériaux donc on défini un type matériau au préalable :

```
type Material = {
    | Lambertian of Color
    | Light of (Color, float)
    | Metal of Color }
```

Chaque matériau est principalement caractérisée par deux valeurs: émission et atténuation.

On peut alors introduire un nouveau type Object qui est simplement un couple primitive, matériau :

```
type Object =
| Triangle * Material
| Sphere * Material
```

II. DESCRIPTION D'UNE SCÈNE ET ALGORITHME DE RAY-TRACING

A. Scène

```
type Scene = Object array
```

sphere_array et triangle_array étant un tableau respectivement des sphères et des triangles présents dans la scène.

```
type Camera = {
  look_from: Vec3,
  lookat, Vec3,
  image_width: int,
  aspect_ratio: float }
```

`look_from` étant la position de la caméra, `look_at` la position dans laquelle la caméra regarde, `image_width` le nombre de pixel par rangés et `aspect_ratio` le ratio permettant de calculer alors `image_height` :

$$\frac{\text{image_width}}{\text{image_height}} = \text{aspect_ratio} \quad (3)$$

prennons alors l'exemple d'une scène décrite par :

```
sph1 = (Sphere(Vec3(3., 0., 0.)),
Lambertian(Color(0., 0., 1.))
sph1 = (Sphere(Vec3(-3., 0., 0.)), Light(Color(1.,
1., 1.))
tri_arr = []
sph_arr = [sph1, sph2]
cam = Camera(look_from=Vec3(0., -11., 4.),
look_at=Vec3(), image_width=1080, aspect_ratio=1)
sc = Scene(sph_arr, tri_arr)
```

Défini la scène dont on peut voir une représentation 3d Fig. 1

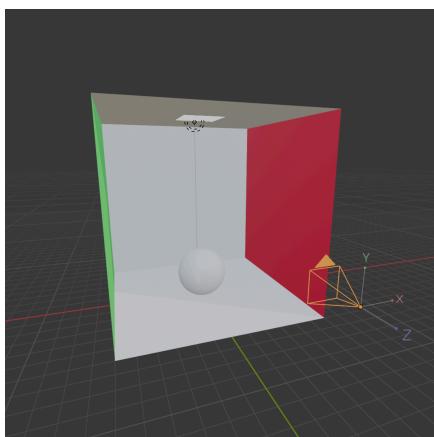


Fig. 1: Représentation 3d de la scène décrite

B. Ray Tracing

Donné un rayon ray de type Ray on peut alors déterminer une fonction d'intersection rayon - scène en $O(n)$ où n est le nombre d'objets dans la scène:

Algorithme I : intersection_scene

Entrée: ray un Ray, scene une Scene

Sortie: intersection le Vec3 premier point d'intersection entre la scène et le rayon (vallant None si pas d'intersection).

```
1  intersection ← None
2  distance ← +∞
3  Pour chaque objet obj de la scène :
   | inter_tmp, distance_tmp ← applique l'algorithme
   | d'intersection adapté en fonction de la géométrie
   | de l'objet.
5  Si distance_tmp < distance alors :
6    | distance ← distance_tmp
7    | intersection ← inter_tmp
8  renvoyer intersection
```

On va alors calculer l'image pixel par pixel en lançant un rayon depuis chaque pixel de la caméra et en le faisant rebondir dans la scène jusqu'à ce qu'il atteigne une source de lumière ou que le nombre de rebonds excède un entier arbitraire MAX_REBONDS. Ces rebonds sont en réalité une marche aléatoire; chaque rayon de rebond d'un rayon incident intersectant un objet de la scène n'est pas strictement déterminé. En effet sa direction de rebond est aléatoirement calculée selon une distribution dépendant du matériau intersecté. Ce processus aléatoire modélise les imperfections, la granularité et autres propriétés de chaque matériau. Par exemple, la distribution de probabilité de rebond d'un matériau Lambertien , modélisant une réflexion diffuse, suit une loi proportionnelle au cosinus de l'angle entre la direction de rebond et la normale à la surface, conformément à la loi du cosinus de Lambert. [3]

Algorithme II : raytracing

Entrée: ray un Ray, scene une Scene

Sortie: Color la couleur calculée du pixel.

```
1  attenuation ← Color(1., 1., 1.)
2  accumulation ← Color(0., 0., 0.)
3  Pour rebonds=0 à MAX_REBONDS faire :
4    | intersection ← intersection_scene(ray, scene)
5    | attenuation *= intersect.mat.attenuation
6    | accumulation
     | += intersect.mat.emmission*attenuation
```

Algorithm II : raytracing

```

7   | ray ← rayon de rebond aléatoirement calculé selon
    | la distribution définie par la matériau d'intersection
    | ainsi que le rayon incident
8   | renvoyer accumulation

```

On remarque que l'accumulation (valeur renvoyée) ne prend que des valeurs non nulles si la marche du rayon intersecte un objet dont le matériau a une émission non nulle. Assez intuitivement cela signifie que la couleur renvoyée par la fonction `raytracing` est non noire seulement si une source de lumière est rencontrée lors des rebonds du rayon primaire. On obtient donc (après avoir appliqué cette fonction à chaque pixel auquel on associe un rayon) une image très bruitée (excepté pour la source de lumière):

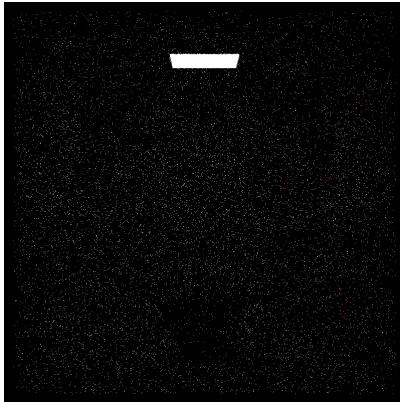
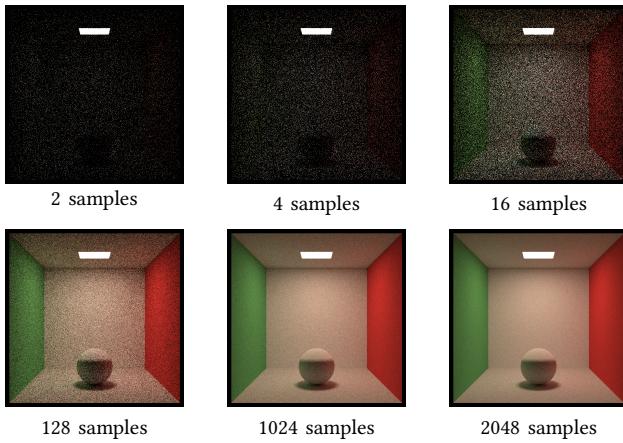


Fig. 2: Image obtenue après un appel de `raytracing` pour chaque pixel

Une telle image est donc appelée sample et on remarque empiriquement que la superposition de multiples samples converge vers une image nette.



C. Compléxité de notre algorithme naïf

Notons L et L les dimensions de l'image produite, n le nombre d'objets de notre scène, MAX_REBONDS le nombre maximum arbitraire de rebonds autorisés lors du raytracing et S le nombre de samples nécessaires pour avoir une image nette. Le nombre de calculs d'intersection de notre algorithme

actuel est alors un $O(1 \times L \times \text{MAX_REBONDS} \times S \times n)$. Pour ordre de grandeur notre Cornell Box contenant une sphère est une scène contenant 7 objets et on en veut une rendu de dimensions 1080×1080 , on choisit aussi comme paramètres $\text{MAX_REBONDS} = 5$ et $S = 2048$ ce qui est plutôt raisonnable. Le dernier rendu de notre grille d'images ci dessus à donc nécessité 83 607 552 000 tests d'intersection. Ce vaste nombre d'opérations reste réalisable car nous utilisons taichi [1], un module python qui permet la parallélisation sur le GPU. Cependant notre scène est assez simple et un nombre trop grand d'objets rendra le nombre de calculs d'intersections trop important pour être réalisé en un temps raisonnable. En effet si on considère non plus $n = 7$ mais $n = 60\,000$ (qui est, on le verra plus tard, un cas courant) on obtient un nombre d'intersections à tester de l'ordre de 10^{14} ce qui n'est pas réalisable en un temps raisonnable. On a donc besoin d'introduire une structure d'accélération.

III. ACCÉLÉRATION BVH

A. Mesh

Afin de simplifier les raisonnements on définit une nouvelle primitive : le mesh. Un mesh est simplement un tableau de triangles composants un maillage. L'agencement de multiples triangles permet l'élaboration de formes complexes comme celle présentée Fig. 3. On étend également le type `Object` pour qu'un élément de type `Mesh * Material` soit considéré comme de type `Object`.

type Mesh = Triangle array



Fig. 3: mesh de La Valse de Camille Claudel composé de 60 000 triangles

On va alors chercher à accélérer le calcul d'intersection entre un mesh et un rayon car jusqu'ici, notre algorithme `intersection_scene` n'a pas mieux que de tester l'intersection avec chaque triangle un par un, ce qui est comme expliqué plus haut, non raisonnable.

B. Hierarchie de Volumes Englobants (Bounding Volume Hierarchy)

L'idée de la BVH est de construire une hiérarchie récursive de boîtes englobantes plusieurs triangles de manière que l'on puisse éliminer la possibilité d'intersection entre le rayon et ces triangles en testant uniquement la boîte englobante et non tous les triangles qu'elle contient. Cette structure recursive est représentée par un arbre dont les noeuds sont des sous boîtes englobantes

a) *Construction*: Pour construire l'arbre BVH récursivement, donné un mesh on définit

```
type BVH_Box = {
    boite: Vec3*Vec3;
    nb_tri: int;
    pos: int }
```

(où boite est la donnée des coordonnées des deux sommets opposés de la boîte, de nb_tri le nombre de triangles dans la boîte et de pos la position dans notre mesh du premier triangle dans cette boîte. On défini alors un noeud par :)

```
type BVH_Node =
| Vide
| Nœud of BVH_Box*BVH_Node*BVH_Node
```

Une première boîte englobant notre mesh entier est alors initialisée avant d'appeler récursivement la fonction de division de boîte diviser. On veille aussi à maintenir l'invariant suivant : « Tous les triangles qui appartiennent à une même boîte sont consécutifs dans notre mesh » de manière à pouvoir savoir quels triangles sont dans une boîte avec l'unique donnée du nombre de triangles dans la boîte et de la position du premier triangle dans le mesh.

Algorithme III : diviser

Entrée: noeudParent de type BVH_Node une feuille, p la profondeur de construction atteinte et le mesh dont on construit le BVH

Sortie: un BVH_node racine de l'arbre BVH construit

```
1   BVH_Node(borneParente, _, _) ← noeudParent ▷ les
    deux enfants sont vides
2   Si p ≤ PROFONDEUR_MAX alors :
3       boiteA ← boiteVide()
4       boiteB ← boiteVide()
5       Pour i = noeud.pos à noeud.pos + noeud.nb_tri :
6           Si mesh[i] est du côté A :
7               agrandir la boiteA pour contenir le triangle
                mesh[i]
8               boiteA.nb_tri += 1
9               swap = boiteA.pos + boiteA.nb_tri + 1
10              mesh[i], mesh[swap] ← mesh[swap], mesh[i]
11              ▷ pour maintenir l'invariant décrit précédemment
12              boiteB.pos += 1
13
14          Sinon :
15              agrandir la boiteB pour contenir le triangle
                mesh[i]
16              boiteB.nb_tri += 1
17              noeudA ← BVH_Node(borneA, Vide, Vide)
18              noeudB ← BVH_Node(borneB, Vide, Vide)
```

Algorithme III : diviser

```
17   renvoyer BVH_Node(borneParente,
        diviser(noeudA, p+1, mesh), diviser(noeudB,
        p+1, mesh))
```

On peut alors construire l'arbre BVH en entier :

Algorithme IV : construction_BVH

Entrée: le mesh dont on veut construire le BVH et la profondeur max PROFONDEUR_MAX de notre construction récursive (considérée par la suite comme variable globale)

Sortie: un BVH_node racine de l'arbre BVH construit

```
1   boiteRacine ← boiteVide
2   Pour i=0 à len(mesh) faire :
3       agrandir boiteRacine pour qu'elle contienne
            mesh[i]
4   racine ← BVH_node(borneRacine, Vide, Vide)
5   renvoyer diviser(racine, 0, mesh)
```

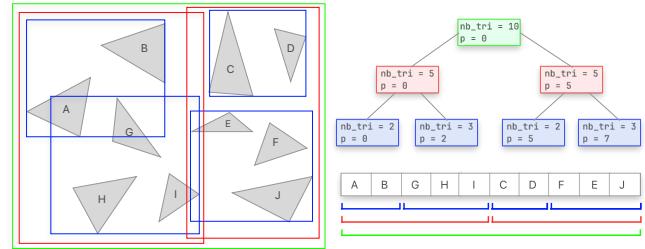


Fig. 4: Représentation d'une structure BVH (représentation des triangles dans l'espace, arbre et tableau de triangles) dans un espace 2D. Chaque couleur correspond à une profondeur.

b) *Algorithme d'intersection rayon / mesh*: Là où on testait l'intersection entre le rayon et chaque triangle on va alors pouvoir réduire considérablement le nombre de calculs d'intersection en testant d'abord nos boîtes englobantes et en convergeant récursivement vers une boîte contenant un nombre réduit de triangles.

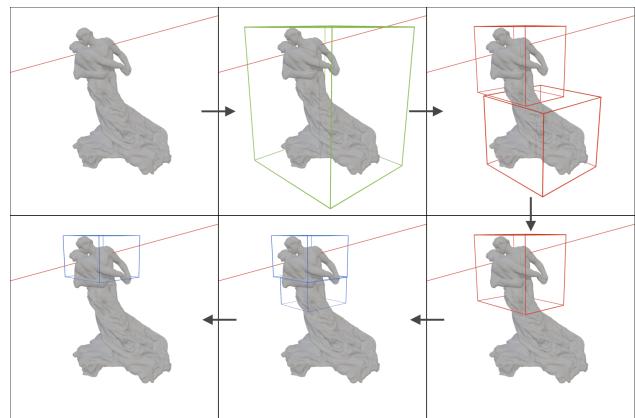


Fig. 5: représentation intuitive de l'algorithme d'intersection du BVH. (ici en 3D)

Algorithme V : intersection_BVH

Entrée: Un rayon incident ray, un mesh mesh et la racine de son arbre BVH associé noeud

Sortie: la distance et le point d'intersection entre le mesh et le rayon incident

```

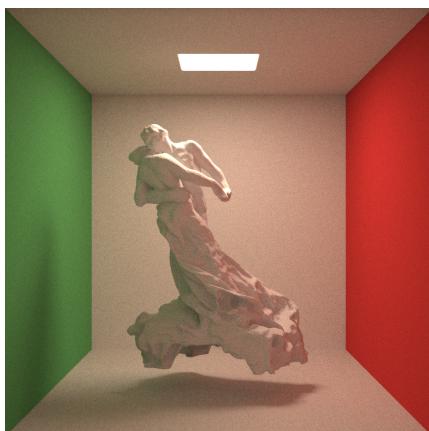
1 | inter ← None
2 | dst ← +∞
3 | Si noeud est une feuille :
4 |   Pour chaque triangle dans la boîte associée à noeud :
5 |     dst_tmp, inter_tmp ← test d'intersection rayon
|       triangle ▷ en utilisant l'algorithme de Möller Trumbore
6 |     Si dst_tmp < dst alors :
7 |       | dst, inter ← dst_tmp, inter_tmp
8 | Sinon :
9 |   BVH_Node(boîteEnglobante, noeudA, noeudB) ←
|     noeud           ▷ juste un matching pour les notations
10 |  Si ray intersecte boîteEnglobante alors :
11 |    dstA, interA ← intersection_BVH(ray, mesh,
|      nodeA)
12 |    dstB, interB ← intersection_BVH(ray, mesh,
|      nodeB)
13 |    Si dstA < dst alors :
14 |      | dst, inter ← dstA, interA
15 |    Si dstB < dst alors :
16 |      | dst, inter ← dstB, interB
17 | Renvoyer inter, dst

```

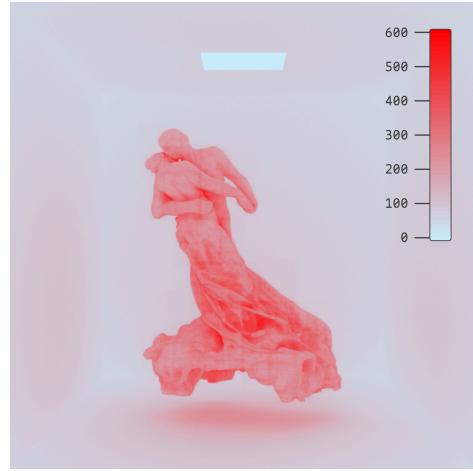
On intègre donc ce nouvel algorithme d'intersection à notre **Algorithme I** en particulier, la ligne 4 utilisera l'algorithme d'intersection accélérée BVH si l'objet est un mesh.

C. Résultats et performances

On peut dorénavant afficher, en utilisant l'algorithme de raytracing, des géométries complexes :



En ce qui concerne les performances, pour l'image affichée ci dessus, le nombre de tests d'intersections (rayon/triangle ou rayon/boîte) par pixel pour chaque sample est de 108 en moyenne. On peut aussi représenter le nombre de tests d'intersection moyen par sample pour chaque pixels :



Il est compliqué d'obtenir avec précision le facteur d'accélération car cela dépend beaucoup de la géométrie du mesh ainsi que de la scène mais la structure BVH permet d'espérer une accélération logarithmique, ce qui est en ordre de grandeur ce que l'on constate expérimentalement.

IV. CONCLUSION

Finalement, nous sommes parvenus à implémenter un algorithme permettant l'affichage d'images dont le calcul repose sur un modèle réaliste de la lumière basé sur les lois de la physique. De plus, la structure BVH a permis une accélération considérable des calculs.

Il faudrait désormais se pencher sur un modèle de matériau encore plus complet, comme la BRDF (Bidirectional Reflectance Distribution Function) permettant de rendre n'importe quel type de matériaux (anisotropes, spécularité non-parfaite...). Quant à l'amélioration de la vitesse de rendu, il faudrait se pencher sur la convergence de notre superposition de samples car notre étude n'a pas vraiment dépassée le « ça marche » très empirique. En effet la convergence des samples peut être considérablement améliorée grâce à l'introduction de biais statistiques et une meilleure compréhension des méthodes d'intégration dites de Monte Carlo.

RÉFÉRENCES

- [1] Y. Hu, T.-M. Li, L. Anderson, J. Ragan-Kelley, et F. Durand, « Taichi: a language for high-performance computation on spatially sparse data structures », *ACM Transactions on Graphics (TOG)*, vol. 38, n° 6, p. 201, 2019.
- [2] T. Möller et B. Trumbore, « Fast Minimum Storage Ray Triangle Intersection ».
- [3] P. Shirley, « True Lambertian Reflection ». [En ligne]. Disponible sur: <https://raytracing.github.io/books/RayTracingInOneWeekend.html#diffusematerials/truelambertianreflection>