Université de Limoges
Faculté des Sciences & Techniques
Faculté des Sciences et Techniques de Limoges

# ISIR - TP4 - Triangle, maillage et BVH

Maxime MARIA

2022-2023

Jusqu'alors, nous n'avions qu'une sphère et un plan comme primitives géométriques. Généralement en informatique graphique, les objets complexes sont représentés par un ensemble de triangles organisés dans un maillage. Dans ce TP, nous allons donc faire en sorte de pouvoir gérer ces maillages triangulaires.

## Pour commencer

Le projet inclus la bibliothèque Assimp [1] qui permet d'importer des objets 3D dans les formats les plus connus. Le chargement du maillage se fait via la méthode `Scene::loadFileTriangleMesh` (attention, ligne 104-105, l'ajout du matériau est commenté).

## 1 Premiers maillages

Vous trouverez deux nouvelles classes permettant de gérer les maillages triangulaires : `MeshTriangle` et `TriangleMeshGeometry`.

La classe `MeshTriangle` représente un maillage triangulaire. Elle est constituée de quatre attributs :

- `_vertices` : l'ensemble des sommets constituant le maillage ;
- `_normals` : les normales à chaque sommet du maillage ;
- `_uvs` : les coordonnées de textures de chaque sommet (inutilisées pour l'instant) ;
- `_triangles`, un ensemble de `TriangleMeshGeometry`.

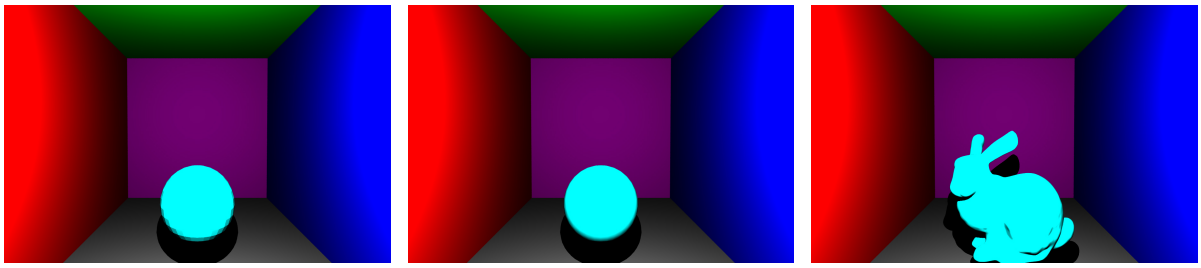
La classe `TriangleMeshGeometry` représente un triangle appartenant à un maillage. Elle est constituée de cinq attributs :

- `_refMesh` est un pointeur vers le maillage auquel le triangle appartient ;
- `_v0`, `_v1` et `_v2` correspondent aux indices des sommets du triangle dans le maillage `_refMesh` ;
- `_faceNormal` est la normale au triangle, calculée dans le constructeur.

Cette architecture un peu particulière nous évite simplement la duplication des sommets dans notre maillage.

1. Implémentez la fonction `TriangleMeshGeometry::intersect` en utilisant la méthode proposée en 1997 par Möller et Trumbore [MT97].
2. Testez votre programme en utilisant la scène donnée en Annexe A (téléchargez les obj depuis community et mettez-les dans le dossier `data`), en plaçant la caméra en $(0, 2, -6)$. Vous devriez obtenir l'image 1(a).
3. Comme nous considérons les normales des triangles, nous distinguons clairement les facettes de la sphère. Pour éviter cela, utilisez les coordonnées barycentriques du point d'intersection $(u, v)$ (calculées dans `TriangleMeshGeometry::intersect`) pour interpoler les normales des sommets et ainsi lisser le rendu : $n = (1 - u - v) * n_0 + u * n_1 + v * n_2$. Vous devriez obtenir l'image 1(b).
4. Reculez la caméra de 1, et chargez l'objet `Bunny.obj` à la place de `uvsphere.obj`. Testez votre programme vous devriez obtenir l'image 1(c). **Attention !** Le calcul est long !

1. https://www.assimp.org/



(a) (b) (c)

FIGURE 1 – Résultats de l'exercice 1.

## 2 Boîtes englobantes

Ça rame un peu non ? Comme vous l'avez remarqué, le calcul de l'image est bien trop long en l'état. En effet, pour chaque pixel, nous testons l'ensemble des primitives géométriques linéairement afin de trouver l'intersection la plus proche. Donc pour le lapin, nous testons plus de 30 000 triangles par pixel...

Dans cet exercice, nous allons utiliser un volume englobant autour de notre maillage dans le but de diminuer la complexité globale du calcul de l'image. Le principe est simple : avant de chercher une intersection avec les triangles composant le maillage, nous allons commencer par tester s'il y a une intersection avec le volume englobant du maillage. Si le rayon ne touche pas ce volume, alors il ne touche aucun triangle du maillage.

Plusieurs types de volumes englobants sont utilisables (sphères, enveloppes convexes,...). Généralement, nous utilisons des boîtes englobantes alignées aux axes (AABB : *Axis-Aligned Bounding Box*).

Une nouvelle classe est disponible dans le projet : `AABB`. Elle contient deux attributs (`_min` et `_max`) correspondant aux deux sommets formant les coins opposés de l'AABB. (Notez qu'on appelle aussi les AABB « boîtes min-max »).

1. Ajoutez un attribut `AABB _aabb` à votre classe `MeshTriangle`.
2. Modifiez la méthode `MeshTriangle::addVertex` pour modifier l'AABB en fonction du sommet ajouté (il va falloir implémenter la méthode `ABBB::extend`).
3. Implémentez `AABB::intersect` (mais comment on fait ? réfléchissez avant de regarder sur Internet svp).
4. Modifiez `MeshTriangle::intersect` pour tester l'intersection avec l'AABB avant de tester l'ensemble des triangles.
5. Testez votre code. Logiquement, le rendu est plus rapide.

## 3 Hiérarchie de volumes englobants

Si l'utilisation d'une boîte englobante par maillage nous permet d'éviter des calculs inutiles pour certains pixels, les rayons touchant l'AABB testent toujours les triangles du maillage de façon linéaire. Pour le lapin, ça passe encore, mais imaginez que notre maillage contienne plusieurs millions de triangles...

Pour réduire la complexité de la recherche d'intersection, une structure accélératrice est généralement utilisée. Plusieurs structures accélératrices ont été proposées telles que les grilles, les arbres BSP (BSP-tree), les arbres kd (kd-tree = BSP-tree dont les plans sont alignés aux axes)...

Dans cet exercice, nous allons utiliser une hiérarchie de volumes englobants ou BVH (*Bounding Volume Hierarchy*), comme vu dans le cours. Les volumes englobants seront des AABB. Nous allons créer un BVH basique... Notez cependant que de nombreuses améliorations ont été proposées dans la littérature.

Vous trouverez deux nouveaux fichiers dans le projet : `bvh.hpp/.cpp`.

La classe `BVH` est simplement composée d'un pointeur vers la liste des triangles du maillage (`_triangles`) et du nœud racine de l'arbre (`_root`).

La classe `BVHNode` correspond à un nœud de l'arbre. Elle contient une boîte englobante (`_aabb`), un pointeur vers ses fils gauche (`_left`) et droit (`_right`), ainsi que l'intervalle des indices des triangles qu'elle contient (`[_firstTriangleId, _lastTriangleId[`).

Nous allons construire l'arbre en choisissant en divisant récursivement en deux l'axe le plus grand de l'AABB du nœud. Ce choix de partition a l'avantage d'être simple, même s'il est loin de fournir un arbre idéal. Nous choisissons deux critères d'arrêt de la récursion : soit le nombre de triangles dans un nœud est inférieur au seuil donné (`_maxTrianglesPerLeaf`), soit la profondeur de l'arbre dépasse celle fixée (`_maxDepth`).

L'arbre se construit donc selon l'algorithme récursif suivant :

```
constr_rec_BVH(noeud, idPremierTri, idDernierTri)
    <calcul aabb noeud>
    if !<critere arret>
        axePartition = plus grand axe de aabb
        milieu = centre de axePartition
        idPartition = <partition(axePartition, milieu)>
        constr_rec_BVH(noeud.gauche, idPremierTri, idPartition)
        constr_rec_BVH(noeud.droit, idPartition, idDernierTri)
```

1. Implémentez les méthodes manquantes de `BVH`.
2. Testez votre code. Logiquement, le rendu est beaucoup plus rapide !
3. Chargez maintenant la scène `conference.obj`. Placez la caméra en $(-250, 500, 330)$ et faites la regarder en $(0, 350, 100)$. Placez un quad lumineux blanc, d'une puissance de 20, en position $(900, 600, -300)$ avec `_u`$= (-800, 0, 0)$ et `_v`$= (0, 0, 300)$. Lancez le rendu, vous devriez obtenir la figure 2.

FIGURE 2 – Conférence.

## 4 Soyons stratégiques ! (Optionnel)

1. Modifiez la construction de votre BVH pour utiliser l'heuristique SAH (*Surface Area Heuristics*) [MB90].

# A Configuration de la scène

```
// ====================================================================
// Add materials.
// ====================================================================
_addMaterial( new ColorMaterial( "RedColor", RED ) );
_addMaterial( new ColorMaterial( "GreenColor", GREEN ) );
_addMaterial( new ColorMaterial( "BlueColor", BLUE ) );
_addMaterial( new ColorMaterial( "GreyColor", GREY ) );
_addMaterial( new ColorMaterial( "MagentaColor", MAGENTA ) );
_addMaterial( new ColorMaterial( "YellowColor", YELLOW ) );
_addMaterial( new ColorMaterial( "CyanColor", CYAN ) );

// ====================================================================
// Add objects.
// ====================================================================
// OBJ.
loadFileTriangleMesh( "UVsphere", DATA_PATH + "uvsphere.obj" );
_attachMaterialToObject( "CyanColor", "UVsphere_defaultobject" );

// Pseudo Cornell box made with infinite planes.
_addObject( new Plane( "PlaneGround", Vec3f( 0.f, -3.f, 0.f ),
                                      Vec3f( 0.f, 1.f, 0.f ) ) );
_attachMaterialToObject( "GreyColor", "PlaneGround" );
_addObject( new Plane( "PlaneLeft", Vec3f( 5.f, 0.f, 0.f ),
                                    Vec3f( -1.f, 0.f, 0.f ) ) );
_attachMaterialToObject( "RedColor", "PlaneLeft" );
_addObject( new Plane( "PlaneCeiling", Vec3f( 0.f, 7.f, 0.f ),
                                       Vec3f( 0.f, -1.f, 0.f ) ) );
_attachMaterialToObject( "GreenColor", "PlaneCeiling" );
_addObject( new Plane( "PlaneRight", Vec3f( -5.f, 0.f, 0.f ),
                                     Vec3f( 1.f, 0.f, 0.f ) ) );
_attachMaterialToObject( "BlueColor", "PlaneRight" );
_addObject( new Plane( "PlaneFront", Vec3f( 0.f, 0.f, 10.f ),
                                     Vec3f( 0.f, 0.f, -1.f ) ) );
_attachMaterialToObject( "MagentaColor", "PlaneFront" );
_addObject( new Plane( "PlaneRear", Vec3f( 0.f, 0.f, -10.f ),
                                    Vec3f( 0.f, 0.f, 1.f ) ) );
_attachMaterialToObject( "YellowColor", "PlaneRear" );

// ====================================================================
// Add lights.
// ====================================================================
_addLight( new PointLight( Vec3f( 0.f, 3.f, -5.f ), WHITE, 100.f ) );
```

# Références

[MB90] David J. MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *Vis. Comput.*, 6(3) :153–166, May 1990.

[MT97] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *J. Graph. Tools*, 2(1) :21–28, October 1997.