E3FI - 2022 - 2023 - Mathématiques 3D - TD #2

Méthodes de calcul d'intersection entre un segment et divers objets 3D

par David Bilemdjian

Pré-requis:

- Projet de base *raylib* transmis en amont
- Développement des structures ***Cylindrical*** et ***Spherical***
- Développement des méthodes de conversion Cartesian ↔ Cylindrical et Cartesian ↔ Spherical
    - *Cylindrical CartesianToCylindrical(Vector3 cart)*
    - *Vector3 CylindricalToCartesian(Cylindrical cyl)*
    - *Spherical CartesianToSpherical(Vector3 cart)*
    - *Vector3 SphericalToCartesian(Spherical sph)*
- TD#1 suffisamment avancé pour disposer des méthodes d'affichage des primitives 3D utilisées dans le TD#2:
    - *Plane, Quad*
    - *Disk*
    - *Sphere*
    - *Cylinder*
    - *Capsule*
    - *Box*
    - *RoundedBox*

Objectifs du TD:

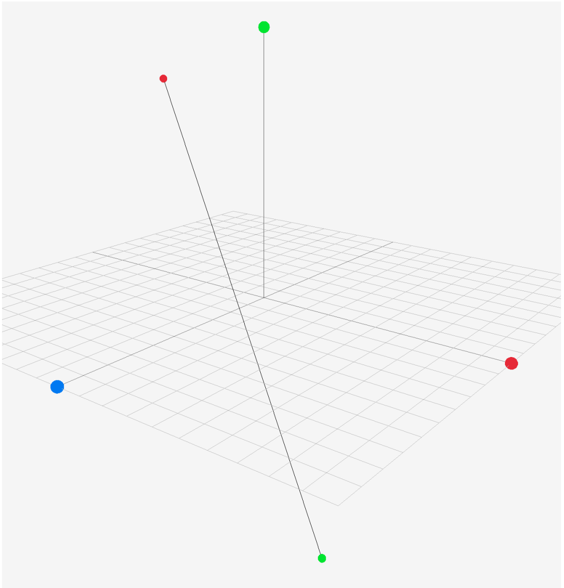
- Développement des méthodes de calcul d'intersection d'un segment avec divers objets 3D

Intersection d'un segment avec un objet 3D

Tout segment $[AB]$ sera ici considéré orienté de A vers B. le point A sera représenté par une petite sphère de couleur rouge (*Red* de la convention *RGB*), et le point B par une petite sphère de couleur verte (*Green* de la convention *RGB*).

Ci-contre, le segment $[AB]$ avec

- A de coordonnées (-5 ; 8 ; 0)
- B de coordonnées (5 ; -8 ; 3).



En ce qui concerne les objets 3D, nous ne nous intéresserons dans ce module qu'à leur enveloppe extérieure, pas à leur domaine intérieur.

Par exemple:

- pour une sphère nous ne considérons que son enveloppe sphérique, pas la boule intérieure
- pour une box, nous ne considérons que les 6 rectangles de son enveloppe extérieure, pas le parallélépipède intérieur
- etc

Ainsi, l'intersection d'un segment avec un objet 3D peut être:

- l'ensemble vide (aucune intersection)
  - $[AB]$ se situe totalement à l'extérieur de l'objet 3D, ou bien est totalement contenu dans celui-ci
- un ensemble de positions discrètes et de segments:
  - une position discrète lorsque le segment $[AB]$ transperce l'enveloppe de l'objet 3D
  - un segment lorsque tout ou partie du segment $[AB]$ est contenu dans l'enveloppe de l'objet

Notre objectif final, *i.e.* l'écriture d'un mini moteur physique d'une balle rebondissant sur des obstacles parallélépipédiques, nous autorise à limiter le calcul d'intersection aux positions discrètes. En effet, le cas où le segment $[AB]$ est contenu partiellement ou totalement dans une face de l'objet 3D correspond au cas limite d'une sphère en mouvement frôlant l'objet sans le pénétrer. (*cf. TD#3*)

Nous ne considérerons également que les cas d'intersection où le point A est à l'extérieur de l'objet. En effet, le cas où A est à l'intérieur correspond à la situation où la sphère a déjà pénétré l'obstacle… Une collision antérieure n'a donc pas été repérée, il s'agit d'une situation de bogue.

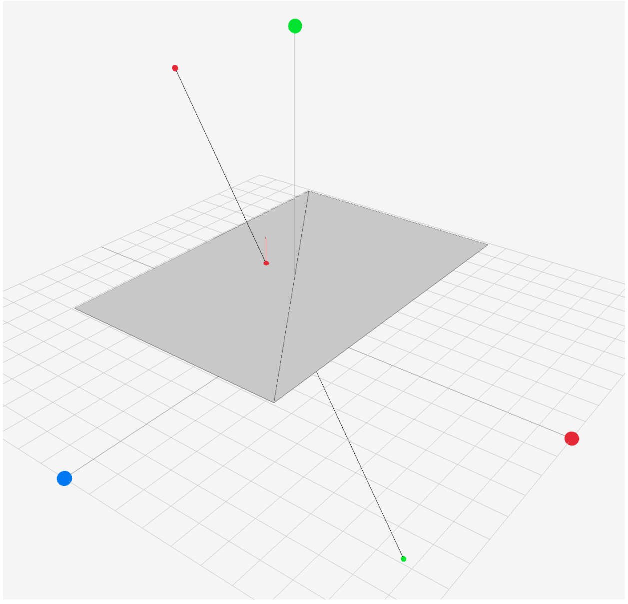
Au vu des objets 3D manipulés dans ce TD et des limitations définies ci-dessus, le nombre de positions discrètes contenues dans l'ensemble des intersections peut être 0, 1 ou 2.

Dans le cas où il y aurait deux points d'intersection, nous ne retiendrons que le point le plus proche de A.

Voici pour illustrer quelques situations d'intersection effective:

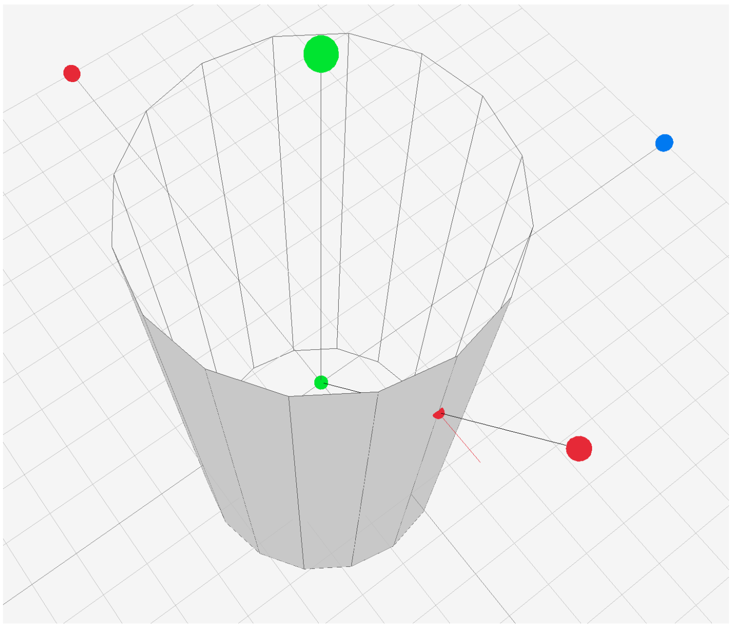
Intersection *Segment* ↔ *Quad*

Au maximum, un unique point d'intersection.



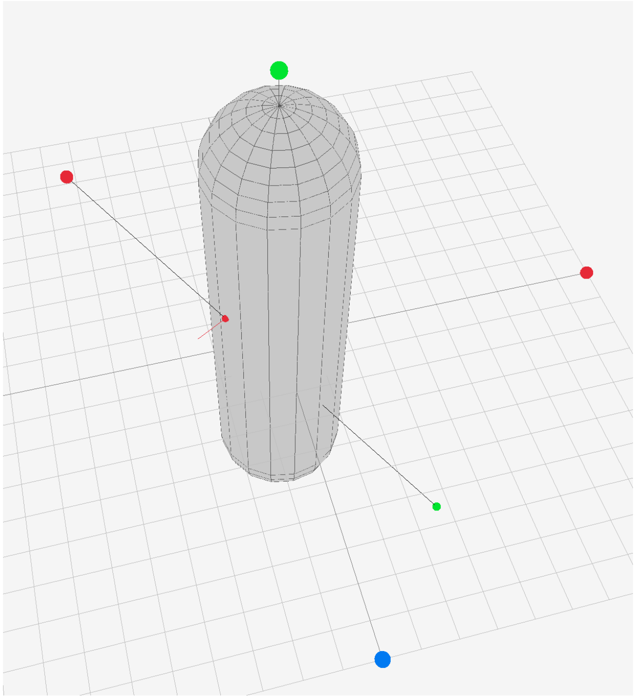
Intersection *Segment* ↔ *InfiniteCylinder*

Dans cette situation, le point B étant à l'intérieur du cylindre infini, il n'y a qu'un unique point d'intersection.



Intersection *Segment* ↔ *Capsule*

Le segment $[AB]$ transperce la capsule de part en part, il y a donc deux points d'intersection. Seul le point le plus proche de A est retenu.



Comme vous pouvez le remarquer sur les illustrations ci-dessus, le vecteur normal (synonyme d'orthogonal) unitaire à la surface de l'objet 3D au niveau du point d'intersection est représenté par un petit trait rouge.

A l'instar des coordonnées du point d'intersection, ce vecteur normal devra également être calculé. Il est essentiel pour calculer la direction de rebond de la sphère sur les obstacles (*cf. TD#3*)

Modélisation mathématique des intersections entre un segment et quelques objets 3D

Il s'agit ici de rappeler les éléments de modélisation mathématique de l'intersection d'un segment avec quelques primitives 3D: *Plane, Quad, Sphere, Disk, InfiniteCylinder*

Le segment $[AB]$ est modélisé par l'équation vectorielle de paramètre *t* réel suivante:

$$\overrightarrow{AM} = t \times \overrightarrow{AB},\ \ t \in [0,1]$$

écrit autrement $$\overrightarrow{OM} = \overrightarrow{OA} + t \times \overrightarrow{AB},\ \ t \in [0,1]$$

Intersection *Segment* ↔ *Plane*

Soit un plan de vecteur normal $\overrightarrow{n}$ et de distance à l'origine d.

Un plan est l'ensemble des points *M* de l'espace 3D tels que la projection du vecteur $\overrightarrow{OM}$ sur $\overrightarrow{n}$ est constante, égale à d.

L'équation vectorielle de ce plan s'écrit donc: $$\overrightarrow{OM} \cdot \overrightarrow{n} = d$$

Le système d'équations vectorielles qui décrit l'intersection *Segment↔Plane* est:

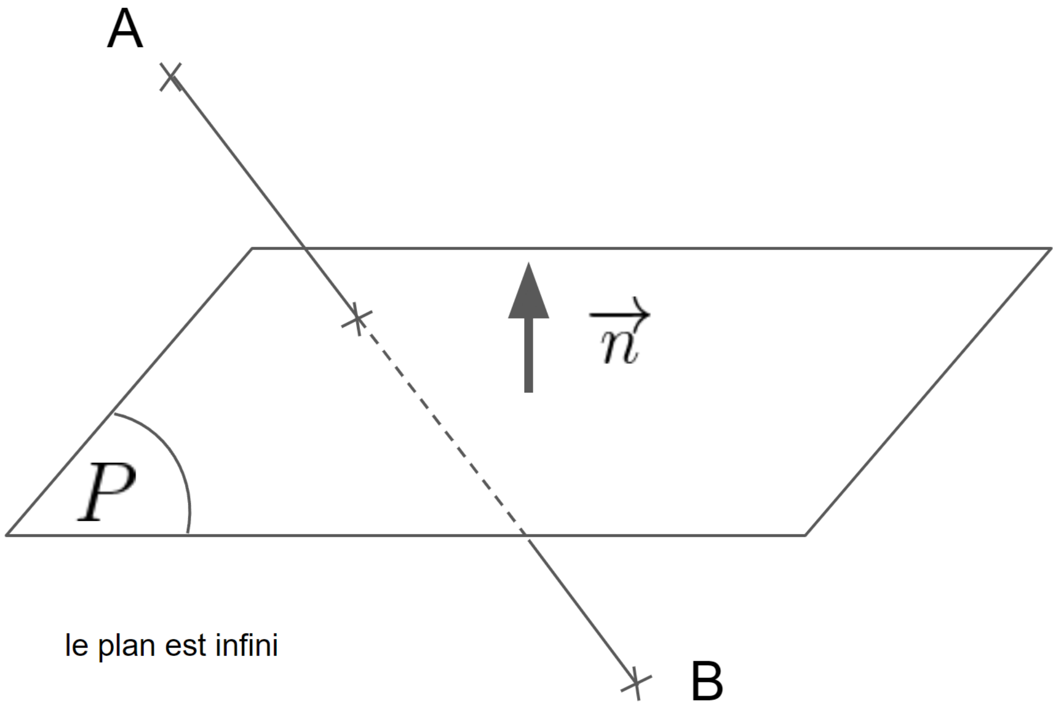
$$\begin{cases} \overrightarrow{OM} \cdot \overrightarrow{n} = d \\ \overrightarrow{OM} = \overrightarrow{OA} + t \times \overrightarrow{AB},\ \ t \in [0,1] \end{cases}$$

Ce système signifie que le point d'intersection appartient à la fois au segment et au plan.
Le point *M* solution de ce système, et qui vérifie donc les deux équations vectorielles, est le point d'intersection recherché.

Si $\overrightarrow{n}$ et $[AB]$ sont orthogonaux alors il ne peut pas y avoir de solution.

Si une solution est trouvée, le vecteur normal au point d'intersection sera:

- $\overrightarrow{n}$ si le vecteur $\overrightarrow{AB}$ pointe vers la face du plan d'où sort le vecteur $\overrightarrow{n}$
- l'opposé de $\overrightarrow{n}$ sinon



Intersection *Segment* ↔ *Quad*

Soit un quadrilatère basé sur un plan infini de vecteur normal $\vec{n}$ et de distance à l'origine d, et possédant des extensions.

La modélisation mathématique de l'intersection *Segment↔Quad* est similaire à celle de l'intersection *Segment↔Plane.*

Les étapes de calcul de l'intersection seront les suivantes:

- Rechercher le point d'intersection *Segment↔Plane* (le plan infini associé au quadrilatère contient le quadrilatère)
- Si ce point d'intersection existe,
  - calculer ses coordonnées dans le référentiel local du *Quad*
  - vérifier que ces coordonnées locales sont bien contenues dans les limites définies par les extensions du *Quad*

Intersection *Segment* ↔ *Sphere*

Soit une sphère de centre Ω et de rayon R. La sphère est l'ensemble des points *M* de l'espace 3D qui sont à la distance R de Ω.
Tous les points sur la sphère vérifient ainsi l'équation vectorielle suivante:

$$\Omega M = R$$

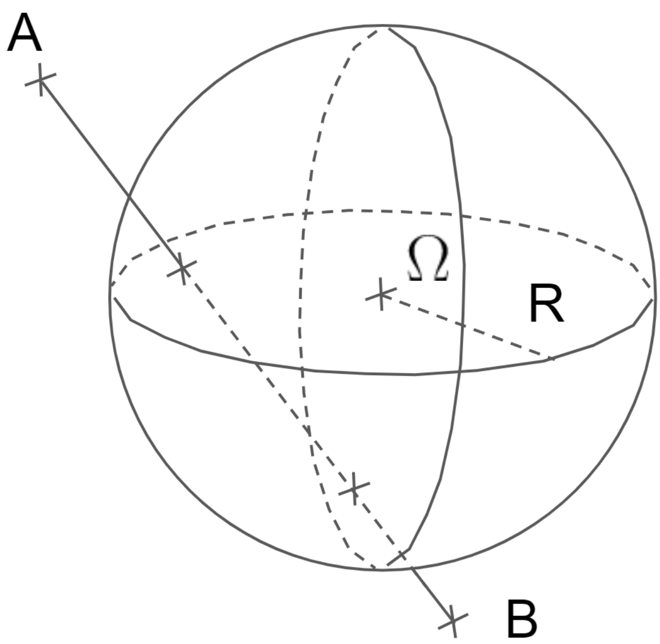
$$\Leftrightarrow \left\|\overrightarrow{\Omega M}\right\|^2 = R^2$$

$$\Leftrightarrow \overrightarrow{\Omega M} \cdot \overrightarrow{\Omega M} = R^2$$

Le système d'équations vectorielles qui décrit l'intersection *Segment↔Sphere* est:

$$\begin{cases} \overrightarrow{OM} = \overrightarrow{OA} + t \times \overrightarrow{AB},\ \ t \in [0,1] \\ \overrightarrow{\Omega M} \cdot \overrightarrow{\Omega M} = R^2 \end{cases}$$

Il signifie que le point d'intersection recherché appartient à la fois au segment et à la sphère. Ce système admet 0, 1 ou 2 solutions. Si 2 solutions sont trouvées, seul le point le plus proche de A est retenu.



Intersection *Segment* ↔ *Disk*

Soit un disque de centre Ω et de rayon R, s'appuyant sur un plan infini de vecteur normal $\overrightarrow{n}$ et de distance à l'origine d.

Ce calcul d'intersection est similaire au calcul de l'intersection *Segment↔Quad*. Seule diffère la manière de vérifier que le point d'intersection appartient bien au disque.

Le système vectoriel à résoudre est le suivant:

$$\begin{cases} \overrightarrow{OM} \cdot \overrightarrow{n} = d \\ \overrightarrow{OM} = \overrightarrow{OA} + t \times \overrightarrow{AB},\ \ t \in [0,1] \\ \overrightarrow{\Omega M} \cdot \overrightarrow{\Omega M} \leqslant R^2 \end{cases}$$

Il signifie que le point d'intersection recherché est à l'intersection d'un plan, d'un segment et d'une boule (l'intérieur de la sphère).

Intersection *Segment* ↔ *InfiniteCylinder*

Soit un cylindre infini, d'axe de vecteur directeur unitaire $\overrightarrow{u}$, et de rayon R. Soit C un point de l'axe du cylindre.

Le cylindre est l'ensemble des points *M* de l'espace 3D à égale distance de l'axe $(C, \overrightarrow{u})$.
Tous les points *M* du cylindre infini vérifient donc l'équation vectorielle suivante:

$$\begin{aligned} &\left\| \overrightarrow{CM} \wedge \overrightarrow{u} \right\| = R \\ \Leftrightarrow &\left(\overrightarrow{CM} \wedge \overrightarrow{u}\right) \cdot \left(\overrightarrow{CM} \wedge \overrightarrow{u}\right) = R^2 \end{aligned}$$

Il existe une $2^{\text{ème}}$ manière d'écrire cette équation vectorielle en utilisant le produit scalaire:

$$\left[\overrightarrow{CM} - \left(\overrightarrow{CM} \cdot \overrightarrow{u}\right)\overrightarrow{u}\right] \cdot \left[\overrightarrow{CM} - \left(\overrightarrow{CM} \cdot \overrightarrow{u}\right)\overrightarrow{u}\right] = R^2$$

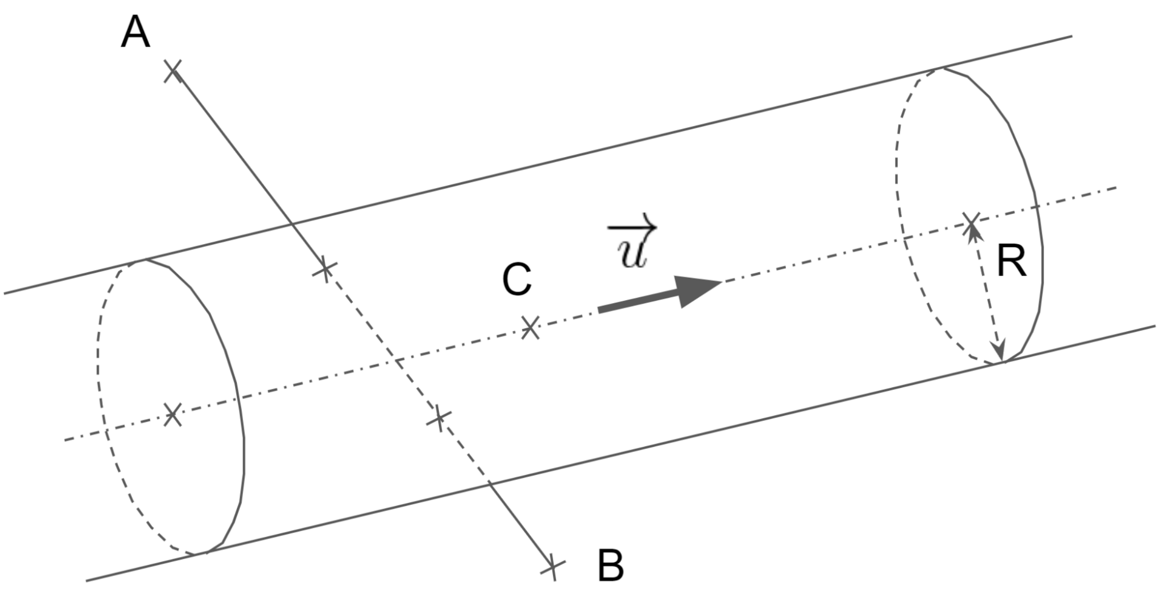
Les deux systèmes d'équations vectorielles possibles qui décrivent l'intersection *Segment*↔*InfiniteCylinder* sont:

$$\begin{cases} \left(\overrightarrow{CM} \wedge \overrightarrow{u}\right) \cdot \left(\overrightarrow{CM} \wedge \overrightarrow{u}\right) = R^2 \\ \overrightarrow{OM} = \overrightarrow{OA} + t \times \overrightarrow{AB}, \ \ t \in [0, 1] \end{cases}$$

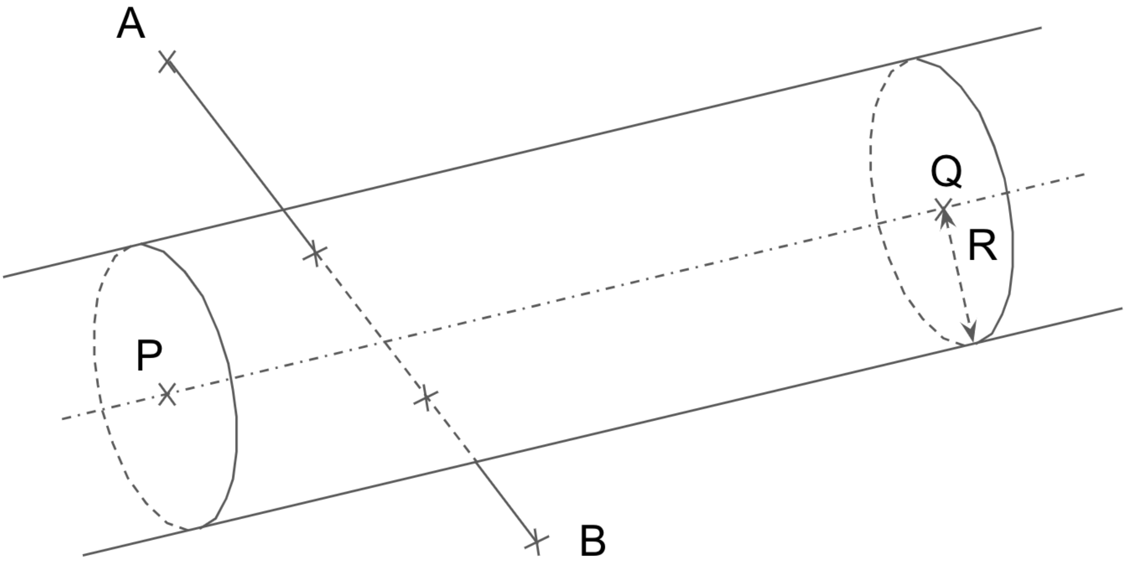
OU

$$\begin{cases} \left[\overrightarrow{CM} - \left(\overrightarrow{CM} \cdot \overrightarrow{u}\right)\overrightarrow{u}\right] \cdot \left[\overrightarrow{CM} - \left(\overrightarrow{CM} \cdot \overrightarrow{u}\right)\overrightarrow{u}\right] = R^2 \\ \overrightarrow{OM} = \overrightarrow{OA} + t \times \overrightarrow{AB}, \ \ t \in [0, 1] \end{cases}$$

Chaque système signifie que le point d'intersection recherché appartient à la fois au segment et au cylindre.
Ce système admet 0, 1 ou 2 solutions. Si 2 solutions sont trouvées, seul le point le plus proche de A est retenu.



Dans le cas d'un cylindre dont l'axe est défini par deux points *P* et *Q* (comme vu en cours):



Les systèmes d'équations vectorielles s'écrivent:

$$\begin{cases} \dfrac{\left(\overrightarrow{PM}\wedge\overrightarrow{PQ}\right)\cdot\left(\overrightarrow{PM}\wedge\overrightarrow{PQ}\right)}{\overrightarrow{PQ}\cdot\overrightarrow{PQ}} = R^2 \\ \overrightarrow{OM} = \overrightarrow{OA} + t\times\overrightarrow{AB},\ \ t\in[0,1] \end{cases}$$

OU

$$\begin{cases} \left[\overrightarrow{PM} - \frac{\overrightarrow{PM}\cdot\overrightarrow{PQ}}{\overrightarrow{PQ}\cdot\overrightarrow{PQ}}\overrightarrow{PQ}\right]\cdot\left[\overrightarrow{PM} - \frac{\overrightarrow{PM}\cdot\overrightarrow{PQ}}{\overrightarrow{PQ}\cdot\overrightarrow{PQ}}\overrightarrow{PQ}\right] = R^2 \\ \overrightarrow{OM} = \overrightarrow{OA} + t\times\overrightarrow{AB},\ \ t\in[0,1] \end{cases}$$

(vous reconnaissez le système d'équations écrit en cours au tableau)

Développement de quelques méthodes outils de la 3D

Le développement des méthodes de calcul d'intersection entre un segment et diverses primitives 3D nécessite quelques méthodes outils. Il vous faut donc les développer en amont.

Méthodes de changement de référentiel

Pour un vecteur:

- du référentiel local au référentiel global:
    - *Vector3 LocalToGlobalVect(Vector3 localVect, ReferenceFrame localRef)*
- du référentiel global au référentiel local:
    - *Vector3 GlobalToLocalVect(Vector3 globalVect, ReferenceFrame localRef)*

Pour une position:

- du référentiel local au référentiel global:
    - *Vector3 LocalToGlobalPos(Vector3 localPos, ReferenceFrame localRef)*
- du référentiel global au référentiel local:
    - *Vector3 GlobalToLocalPos(Vector3 globalPos, ReferenceFrame localRef)*

Méthodes géométriques diverses

- Méthode retournant le projeté orthogonal d'un point sur une droite
    - *Vector3 ProjectedPointOnLine(Vector3 linePt, Vector3 lineUnitDir, Vector3 pt)*
- Méthode retournant la distance au carré d'un point à un segment
    - *float SqDistPointSegment(Segment seg, Vector3 pt)*
- Méthode qui permet de déterminer si un point est situé à l'intérieur d'une Box
    - *bool IsPointInsideBox(Box box, Vector3 globalPt)*

Développement des méthodes d'intersection d'un segment avec des objets 3D

Toutes les méthodes d'intersection auront une signature similaire. Voici les prototypes des méthodes qui devront être développées:

- *bool IntersectLinePlane(Line line, Plane plane, float& t, Vector3& interPt, Vector3& interNormal)*
- *bool IntersectSegmentPlane(Segment seg,Plane plane, float& t, Vector3& interPt, Vector3& interNormal)*
- *bool IntersectSegmentQuad(Segment seg, Quad quad, float& t, Vector3& interPt, Vector3& interNormal)*
- *bool IntersectSegmentDisk(Segment segment, Disk disk, float& t, Vector3& interPt, Vector3& interNormal)*
- *bool IntersectSegmentSphere(Segment seg, Sphere s, float& t, Vector3& interPt, Vector3& interNormal)*
- *bool IntersectSegmentInfiniteCylinder(Segment segment, InfiniteCylinder cyl,float& t, Vector3& interPt, Vector3& interNormal)*
- *bool IntersectSegmentCylinder(Segment segment, Cylinder cyl, float& t, Vector3& interPt, Vector3& interNormal)*
    - le segment peut transpercer le corps cylindrique et/ou les extrémités discoïdes
- *bool IntersectSegmentCapsule(Segment seg,Capsule capsule,float& t, Vector3& interPt, Vector3& interNormal)*
    - le segment peut transpercer le corps cylindrique et/ou les extrémités hémisphériques
- *bool IntersectSegmentBox(Segment seg, Box box, float& t, Vector3& interPt, Vector3& interNormal)*
- *bool IntersectSegmentRoundedBox(Segment seg, RoundedBox rndBox, float& t, Vector3& interPt, Vector3& interNormal)*
    - le segment peut transpercer une des faces rectangulaires, les arêtes cylindriques et les coins sphériques

Chaque méthode retourne une valeur de type *bool*:
- *true* si un point d'intersection a été trouvé
- *false* sinon

Les paramètres de chaque méthode sont:
- *Segment seg* , le segment $[AB]$ (excepté pour la 1ère méthode qui propose un paramètre de type Line)
- l'objet 3D, de type *Plane, Quad, Disk, Sphere, InfiniteCylinder, Cylinder, Capsule, Box* ou *RoundedBox*
- les trois paramètres suivants sont passés par référence (&) et permettent de récupérer en sortie les données d'intersection calculées par la méthode
    - *t* paramètre d'interpolation entrant dans l'écriture de l'équation vectorielle du segment $[AB]$

$$\overrightarrow{AM} = t \times \overrightarrow{AB},\ \ t \in [0, 1]$$

- *interPt* point d'intersection (coordonnées données dans le référentiel global)
- *interNormal* vecteur normal à la surface de l'objet 3D au niveau du point d'intersection (coordonnées données dans le référentiel global)

Votre réflexion doit s'appuyer sur des schémas. Il est impossible de raisonner de manière totalement abstraite. Pour chaque objet 3d, dessinez toutes les configurations possibles du segment, y compris les cas particuliers. Ce sont vos schémas qui vous permettront de développer des algorithmes pertinents et optimisés.

Veillez à optimiser les méthodes afin de minimiser la quantité de calculs effectués. Écartez le plus tôt possible les cas de non intersection.
Vos algorithmes doivent viser à écarter ces cas de non intersection dans un ordre qui optimise globalement la méthode, les cas fréquents de non intersection étant en général traités en premier.
Voir en exemple en Annexe 1 l'implémentation d'une méthode de calcul d'intersection *Line ↔ Plane.* Vous pouvez remarquer que le test du parallélisme entre la droite et le plan est effectué dès le début de la méthode. Ce test est cependant obligatoire pour éviter la division par zéro donc il n'illustre pas véritablement l'optimisation.
Un autre exemple sera plus pertinent, celui de l'intersection Segment↔*Sphere.* Le calcul du point d'intersection passe par la résolution numérique d'une équation polynomiale du second degré. Ce calcul est coûteux car basé sur la fonction "racine carrée". Il est possible d'optimiser la méthode en écartant en amont les cas où le vecteur $\overrightarrow{AB}$ "s'éloigne" du centre de la sphère. Je vous laisse le soin d'interpréter mathématiquement l'éloignement d'un vecteur par rapport à un point.

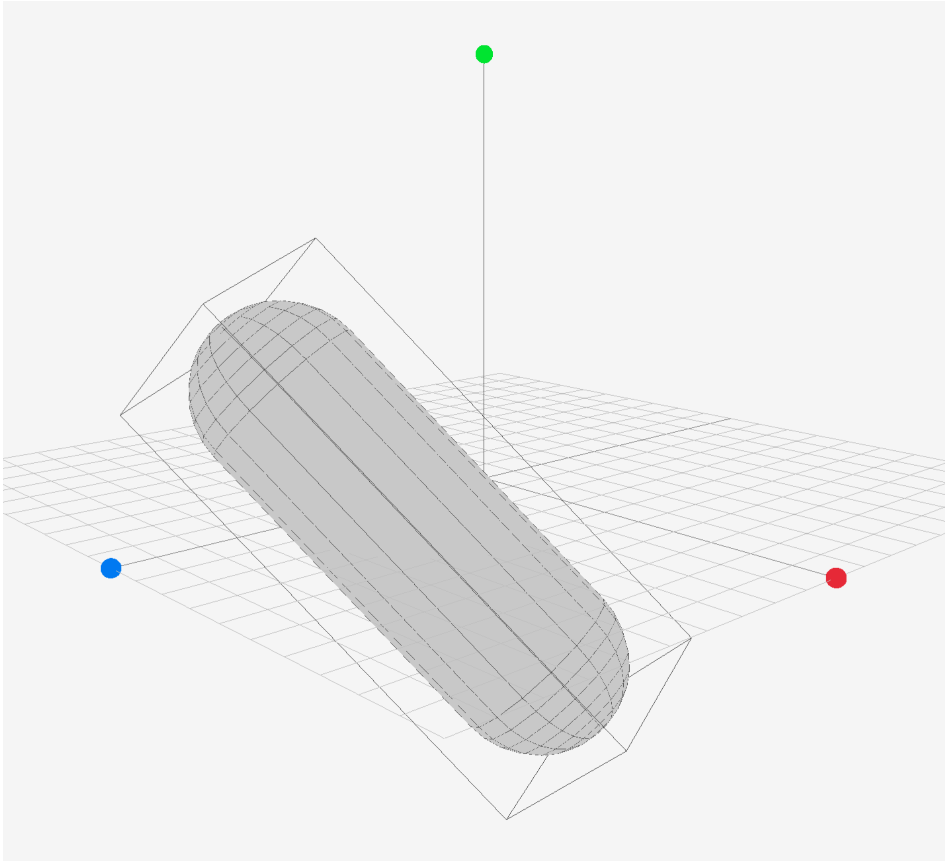
Plus l'objet est complexe, moins l'optimisation est triviale. L'optimisation sera donc complexe pour les méthodes d'intersection *Segment↔Cylinder, Segment↔Capsule* et *Segment↔RoundedBox.*

Un premier niveau d'optimisation est de vérifier si le segment $\overrightarrow{AB}$ intersecte la boîte englobante orientée de l'objet 3d. Cette boîte englobante, nommée *OBB* (Oriented Bounding Box) est le plus petit parallélépipède qui englobe l'objet 3d. Si $\overrightarrow{AB}$ n'intersecte pas l'*OBB* de l'objet, alors il ne risque pas s'intersecter l'objet 3d lui-même.

Pour chaque objet 3D, l'*OBB* est facile à construire, car il s'agit d'une *Box* qui partage le même référentiel avec l'objet, et dont les extensions permettent d'englober l'objet.

- Par exemple, soit une Capsule définie ainsi
    - Référentiel *ref*
    - demi-hauteur: *halfHeight*
    - rayon: *radius*
- Son *OBB* sera une *Box* aux caractéristiques suivantes:
    - Référentiel: *ref*
    - extents = *{radius , halfHeight+radius , radius}*

Si vous mettez en place quelques critères d'optimisation, dont l'*OBB*, ce sera déjà très bien: ne cherchez pas de solution optimale, vous y passeriez beaucoup trop de temps.



Une capsule et son OBB

Annexe 1

Méthode *IntersectLinePlane* de calcul d'intersection *Line* ↔ *Plane*

```
bool IntersectLinePlane(Line line, Plane plane, float& t, Vector3& interPt, Vector3& interNormal)
{
	// no intersection if line is parallel to the plane
	float dotProd = Vector3DotProduct(plane.normal, line.dir);
	if (fabsf(dotProd) < EPSILON) return false;

	// intersection: t, interPt & interNormal
	t = (plane.d - Vector3DotProduct(plane.normal, line.pt)) / dotProd;
	interPt = Vector3Add(line.pt, Vector3Scale(line.dir, t));	// OM = OA+tAB
	interNormal = Vector3Scale(plane.normal,
Vector3DotProduct(Vector3Subtract(line.pt, interPt), plane.normal) < 0 ? -1.f : 1.f);
	return true;
}
```

Code de test de la méthode à insérer dans la méthode *main* au sein du bloc délimité par les appels aux méthodes *BeginMode3D* et *EndMode3D*;

```
//TESTS INTERSECTIONS
Vector3 interPt;
Vector3 interNormal;
float t;

//THE SEGMENT
Segment segment = { {-5,8,0},{5,-8,3} };
DrawLine3D(segment.pt1, segment.pt2, BLACK);
MyDrawPolygonSphere({ {segment.pt1,QuaternionIdentity()},.15f }, 16, 8, RED);
MyDrawPolygonSphere({ {segment.pt2,QuaternionIdentity()},.15f }, 16, 8, GREEN);

// TEST LINE PLANE INTERSECTION
Plane plane = { Vector3RotateByQuaternion({0,1,0}, QuaternionFromAxisAngle({1,0,0},time * .5f)), 2 };
ReferenceFrame refQuad = { Vector3Scale(plane.normal, plane.d),
				QuaternionFromVector3ToVector3({0,1,0},plane.normal) };
Quad quad = { refQuad,{10,1,10} };
MyDrawQuad(quad);

Line line = { segment.pt1,Vector3Subtract(segment.pt2,segment.pt1) };

if (IntersectLinePlane(line, plane, t, interPt, interNormal))
{
	MyDrawPolygonSphere({ {interPt,QuaternionIdentity()},.1f }, 16, 8, RED);
	DrawLine3D(interPt, Vector3Add(Vector3Scale(interNormal, 1), interPt), RED);
}
```

Annexe 2

Je vous propose ici une liste subjective des méthodes très utilisées au cours de ce projet. N'hésitez pas cependant à parcourir le fichier *raymath.h* pour prendre connaissance de l'entièreté de la boîte à outils mathématique de *raylib*.

Les matrices sont en général très utilisées en 3D, mais comme nombre d'entre vous n'ont pas encore suivi de cours d'algèbre linéaire, je les ai omises à dessein. N'hésitez pas cependant à les utiliser le cas échéant.

Prototype de la méthode	**Description**	**Exemple d'utilisation**
Méthodes sur le type *float*		
float Clamp(float value, float min, float max)	Restreint une valeur à une plage définie par les valeurs min et max	*float v = 4.0f;* *v = Clamp(v,1,3);* *// v vaut 3*
float Lerp(float start, float end, float amount)	Interpolation linéaire entre *start* et *end* par le paramètre d'interpolation *amount* $result = start + (end - start) \times amount$	*float v = Lerp(3,5,.75f);* *// v vaut 4.5*
Méthodes sur le type *Vector3* *Remarque: A l'exception de Vector3CrossProduct* et *Vector3RotateByQuaternion, les méthodes existent également pour le type* ***Vector2***		
Vector3 Vector3Zero(void)	Retourne un vecteur nul	*Vector3 v = Vector3Zero();* *// v vaut {0,0,0}*
Vector3 Vector3One(void)	Retoune un vecteur de coordonnées {1,1,1} Attention: le vecteur n'est pas unitaire car sa norme vaut $\sqrt{3}$	*Vector3 v = Vector3One();* *// v vaut {1,1,1}*
Vector3 Vector3Add(Vector3 v1, Vector3 v2)	Somme de deux vecteurs	*Vector3 u = {1,2,3};* *Vector3 v = {4,-3,-7};* *Vector3 w = Vector3Add(u,v); // vaut {5,-1,-4}*

Vector3 Vector3Subtract(Vector3 v1, Vector3 v2)	Soustraction de deux vecteurs	*Vector3 u = {1,2,3};* *Vector3 v = {4,-3,-7};* *Vector3 w = Vector3Subtract(u,v); // vaut {-3,5,10}*
Vector3 Vector3Scale(Vector3 v, float scalar)	Multiplication d'un vecteur par un scalaire. Très utilisé pour construire un vecteur à partir d'un vecteur unitaire (de norme 1) et d'une norme.	*Vector3 dir = Vector3Normalize({1,2,3});* *float speed = 4.0f;* *Vector3 velocity = Vector3Scale(dir,speed);* *// velocity vaut environ {1.069045, 2.138090, 3.207135}*
Vector3 Vector3CrossProduct(Vector3 v1, Vector3 v2)	Produit vectoriel entre deux vecteurs, utilisable uniquement en 3D. Le vecteur résultat est orthogonal aux deux vecteurs opérandes, et sa norme est égale au produit des normes multiplié par le sinus de l'angle entre les deux vecteurs. Utile par exemple pour trouver le vecteur normal à un plan défini par deux vecteurs, ou pour calculer l'aire d'un triangle (voir exemple ci-contre).	*Vector3 A = {1,2,3};* *Vector3 B = {4,5,6};* *Vector3 C = {-5,-6,-7};* *float areaABC = .5f*Vector3Length(Vector3CrossProduct(Vector3Subtract(B,A),Vector3Subtract(C,A)));* *// areaABC vaut environ 7.348469 unités de longueur au carré*
float Vector3Length(const Vector3 v)	Norme d'un vecteur	*Vector3 v = {1,2,3};* *float vectModulus = Vector3Length(v);* *// vectModulus vaut* $\sqrt{14}$*, soit environ 3.74165*

float Vector3LengthSqr(const Vector3 v)	Norme au carré d'un vecteur.
	Certaines formules mathématiques font appel à la norme au carré d'un vecteur.
	Et pour comparer deux normes de vecteurs, il est parfois plus performant de comparer les normes au carré, car cela évite le calcul coûteux de la racine carrée (voir exemple ci-contre).
		Vector3 Omega = {1,2,3}; // center of sphere
		float radius = 4; // radius of sphere
		Vector3 A = {4,5,6}; // any point
		
		Vector3 OmegaA = Vector3Subtract(A,Omega);
		if(Vector3LengthSqr(OmegaA)<radius*radius)
		    printf("A is inside sphere");

float Vector3DotProduct(Vector3 v1, Vector3 v2)	Produit scalaire entre deux vecteurs. Utile pour de nombreuses applications impliquant le calcul d'une quantité de projection d'un vecteur sur un axe.	*Vector3 u = {1,2,3};* *Vector3 v = {4,-3,-7};* *if(fabsf(Vector3DotProduct(u,v))<EPSILON)* *printf("u and v are orthogonal");*
float	Calcule la distance entre deux points	*float dist = Vector3Distance({0,3,1},{-1,-2,-4});*

Vector3Distance(Vector3 v1, Vector3 v2)		*// dist vaut 7.141428*

Vector3 Vector3Normalize(Vector3 v)	Retourne un vecteur normalisé, c'est-à-dire le vecteur de norme 1 de même direction que le vecteur passé en paramètre. Non calculable si le vecteur opérande est de norme nulle.
	Très utile car de très nombreuses formules vectorielles et certaines méthodes informatiques sont basées sur des vecteurs unitaires.
		*Vector3 velocity = {2,3,4};
		float angle = acosf(Vector3DotProduct({1,0,0},Vector3Normalize(velocity)));
		// l'angle positif le plus court entre le vecteur velocity et l'axe (Ox) vaut ici 1.190290 radians*

Vector3 Vector3Negate(Vector3 v)	Retourne le vecteur opposé	*Vector3 u = Vector3Normalize({1,2,3});
		printf("%f", Vector3DotProduct(u, Vector3Negate(u)));
		// la console affiche logiquement -1*

Vector3 Vector3RotateByQuaternion(Vector3 v, Quaternion q)	Retourne le vecteur auquel a été appliqué une rotation définie par un quaternion. Très utile pour faire tourner une direction dans l'espace.	*Plane plane = { Vector3RotateByQuaternion({0,1,0}, QuaternionFromAxisAngle({1,0,0},time * .5f)), 2 }; // définit un plan dont le vecteur normal initial $\vec{\jmath}$ tourne autour de l'axe (Ox) avec le temps*

Vector3 Vector3Reflect(Vector3 v, Vector3 normal)	Retourne le vecteur réfléchi par rapport à un vecteur normal.
	Utile pour calculer la direction de rebond d'une balle entrant en collision avec un obstacle.
	(https://docs.unity3d.com/ScriptReference/Vector3.Reflect.html)
		*Vector3 velocity = {1,1,0};
		Vector3 normal = {0,1,0};
		velocity = Vector3Reflect(velocity, normal);
		// velocity vaut logiquement {-1,1,0}*

Méthodes sur le type *Quaternion*		
Quaternion QuaternionIdentity(void)	Retourne le quaternion identité, correspondant à une rotation nulle. Utilisé en général pour initialiser tout quaternion.	*Quaternion q = QuaternionIdentity(); // q vaut {x = 0 , y = 0 , z = 0 , w = 1}*
Quaternion QuaternionFromAxisAngle(Vector3 axis, float angle)	Initialisation d'un quaternion de rotation à partir du vecteur définissant l'axe de rotation, et de l'angle de rotation.	*Quaternion q1 = QuaternionFromAxisAngle({ 1,0,0 }, PI); // q1 représente la rotation autour de l'axe (Ox) d'angle 45°*

*void QuaternionToAxisAngle (Quaternion q, Vector3 *outAxis, float *outAngle)*	Récupération des vecteur et angle de la rotation définie par le paramètre *q*.	*// q est un quaternion d'orientation quelconque
		Vector3 vect;
		float angle;
		QuaternionToAxisAngle(q, &vect, &angle);
		rlRotatef(angle * RAD2DEG, vect.x, vect.y, vect.z);
		// rlRotatef permet de transformer l'espace 3D en rotation avant d'envoyer à la carte graphique des instructions de traçage OpenGL*

Quaternion QuaternionMultiply(Quaternion q1, Quaternion q2)	Multiplication de deux quaternions: correspond géométriquement à la transformation en séquence de deux rotations:
	● Dans le référentiel global: la rotation définie par *q2* suivie de la rotation définie par *q1*
	● Dans le référentiel local: la rotation définie par *q1* suivie par la rotation définie par *q2*
		*Quaternion q1 = QuaternionFromAxisAngle({ 1,0,0 }, PI / 4); // rotation autour de l'axe (Ox) d'angle 45°
		Quaternion q2 = QuaternionFromAxisAngle({ 0,1,0 }, PI / 4); // rotation autour de l'axe (Oy) d'angle 45°
		Quaternion q3 = QuaternionMultiply(q1, q2);
		ReferenceFrame ref = ReferenceFrame({0,0,0},q3);
		// l'orientation du référentiel ref est la concaténation des rotations définies, dans le référentiel global, par q2 puis par q1*

Quaternion QuaternionFromVector3ToVector3(Vector3 from, Vector3 to)	Définit un quaternion de rotation qui permet de faire coïncider deux vecteurs (ici les paramètres *from* et *to*). Les deux vecteurs doivent être unitaires. Utile par exemple pour redresser verticalement un objet.	*// ref est un référentiel quelconque
		Quaternion qUprightRot = QuaternionFromVector3ToVector3(ref.j,{0,1,0});
		ref.q = QuaternionMultiply(qUprightRot ,ref.q);
		ref.i = Vector3RotateByQuaternion({ 1,0,0 }, ref.q);
		ref.j = Vector3RotateByQuaternion({ 0,1,0 }, ref.q);
		ref.k = Vector3RotateByQuaternion({ 0,0,1 }, ref.q);
		// le référentiel est redressé verticalement*