

E3FIC - 2022 - 2023 - Mathématiques 3D - TD #1

Primitives 3D & Méthodes de traçage OpenGL associées (polygons & wireframes)

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)*

Objectifs du TD:

- Application du système de Coordonnées sphériques au développement d'une caméra orbitale
- Développement des structures C de définition d'objets primitifs 3D
- Développement et optimisation des méthodes de traçage de ces objets 3D, basées sur le rendu OpenGL "immédiat":

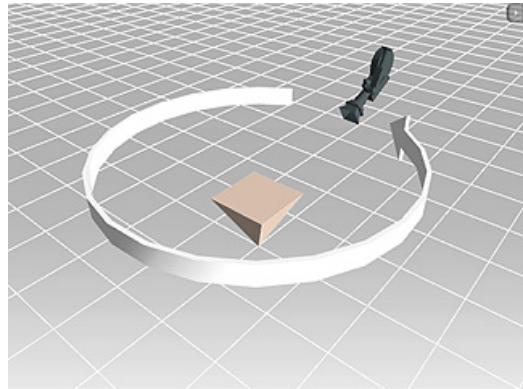
En prolongement du TD:

- Méthodes de traçage d'une portion de cylindre
- Méthodes de traçage d'une portion de sphère

◦

Application des coordonnées sphériques: développement d'une caméra orbitale

Une caméra orbitale permet à l'utilisateur d'une application 3D de contrôler à la souris le point de vue sur un objet en tournant autour de celui-ci: la caméra orbite autour de l'objet, son axe de regard dirigé à chaque instant vers l'objet. La trajectoire de la caméra se développe ainsi sur une sphère dont il est possible de modifier le rayon en se rapprochant ou s'éloignant de l'objet (zoom/unzoom à la molette).



Les coordonnées sphériques sont donc tout indiquées pour modéliser mathématiquement la position de la caméra à chaque instant.

Pour rappel:

- L'unité de la composante *rho* des coordonnées sphériques est le mètre (m)
- L'unité des composantes angulaires *theta* et *phi* est le radian

La caméra est initialisée en début de méthode *main*:

```
Vector3 cameraPos = { 8.0f, 15.0f, 14.0f }; // position initiale de la caméra
Camera camera = { 0 }; // la caméra
camera.position = cameraPos; // position de la caméra
camera.target = { 0.0f, 0.0f, 0.0f }; // la caméra regarde l'origine
camera.up = { 0.0f, 1.0f, 0.0f }; // définit la "verticalité" de la caméra
camera.fovy = 45.0f; // field of view (ouverture angulaire verticale)
camera.type = CAMERA_PERSPECTIVE; // projection perspective
SetCameraMode(camera, CAMERA_CUSTOM); // nous positionnons la caméra
```

Le point cible du regard de la caméra et l'orientation de celle-ci sont donc automatiquement gérés par *raylib*, il nous reste seulement à préciser sa position à chaque frame.

Les mouvements de la souris permettent de contrôler le mouvement de la caméra:

- le mouvement horizontal (bouton droit enfoncé) contrôle l'azimut *theta*, qui peut prendre n'importe quelle valeur réelle
- le mouvement vertical (bouton droit enfoncé) contrôle l'élévation *phi*, à limiter entre 1° et 179° (à convertir en radians). Pourquoi cette limitation ? En 0 et 180° apparaît le phénomène de Gimbal Lock qui perturbe grandement l'orientation de la caméra. Nous l'étudierons ultérieurement ensemble plus en détails.
- la molette contrôle la distance *rho* comprise entre deux valeurs limites *rhoMin* et *rhoMax*

Travail à réaliser: compléter la méthode

```
void MyUpdateOrbitalCamera(Camera* camera, float deltaTime)
```

(remarque: le paramètre *deltaTime* ne vous sera pas utile)

Les variables locales nécessaires au bon fonctionnement de cette caméra orbitale sont:

- la position en coordonnées sphériques de la caméra
 - **static Spherical sphPos = { 10,PI / 4.0f,PI / 4.0f };** // ici la position de départ de la caméra est rho = 10 m, theta = 45°, phi = 45°
 - le mot clé **static** permet à la variable locale de conserver sa valeur d'un appel à l'autre de la méthode
- la vitesse de déplacement en coordonnées sphériques de la caméra
 - **Spherical sphSpeed = { 2.0f,0.04f,0.04f };** // 2 m/incrément de molette et 0.04 radians/pixel
- les valeurs min et max du rayon d'orbite (*Spherical.rho*)
 - **float rhoMin = 4;** // 4 m
 - **float rhoMax = 40;** // 40 m
- la position courante de la souris:
 - **Vector2 mousePos;**
- la position précédente de la souris à l'écran, utile pour calculer le déplacement de la souris d'une frame à l'autre
 - **static Vector2 prevMousePos = { 0,0 };**
- le vecteur de déplacement de la souris à l'écran:
 - **Vector2 mouseVect;**
- le vecteur de déplacement de la caméra, en coordonnées sphériques
 - **Spherical sphDelta;**

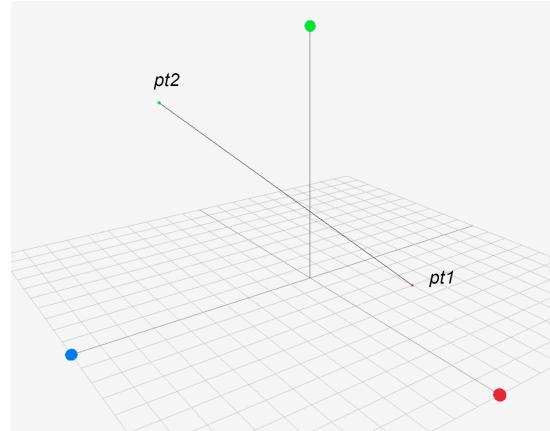
Les étapes de calcul sont les suivantes:

1. Récupération de la position courante de la souris à l'écran, à l'aide de la méthode *GetPosition()*
2. Calcul du vecteur de déplacement de la souris par différence entre la position courante de la souris et la position précédente
 - utilisez la méthode *Vector2Subtract*
3. Mise à jour de la position précédente de la souris avec la position courante
4. Calcul du vecteur de déplacement de la caméra en coordonnées sphériques
 - la méthode *GetMouseWheelMove()* permet de connaître la magnitude de la rotation de la molette
 - *IsMouseButtonDown(MOUSE_RIGHT_BUTTON)* retourne *true* si le bouton droit de la souris est enfoncé
5. Calcul de la nouvelle position de la caméra en coordonnées sphériques, en prenant en compte les limitations sur les coordonnées *rho* et *phi*
 - la méthode *Clamp* permet de borner une valeur numérique dans un intervalle donné
 - la constante de proportionnalité *DEG2RAD* permet de convertir les angles de degré vers radian
6. Convertir en cartésien la nouvelle position et l'affecter à la caméra
 - utilisez *SphericalToCartesian*
 - la position de la caméra est accessible via *camera->position*

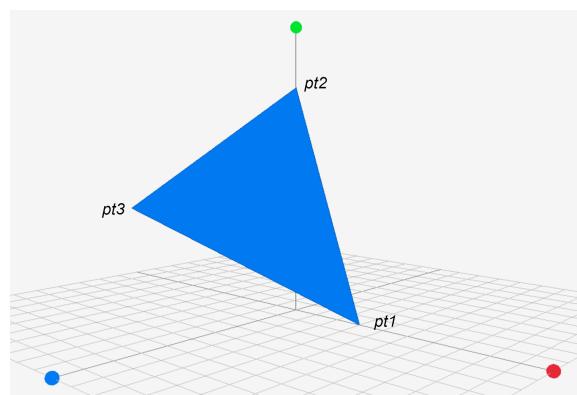
Objets primitifs 3D et structures de code

Les objets primitifs 3D sont des entités géométriques simples. Certaines seront modélisées mathématiquement en cours.

- le point, dont la position dans l'espace est définie par 3 coordonnées
 - soit 3 coordonnées cartésiennes x, y, z , implémentées dans raylib par le type **Vector3**
 - soit 3 coordonnées cylindriques **Cylindrical**: ρ, θ, y
 - soit 3 coordonnées sphériques **Spherical**: ρ, θ, ϕ
- la droite **Line**, définie par un point et un vecteur directeur. La droite est considérée orientée dans le sens de son vecteur directeur. Pour afficher une droite infinie dans la scène 3D, un Segment sera utilisé (voir ci-dessous). Il faudra faire un effort d'imagination pour étendre visuellement les limites du segment à l'infini.
 - **Vector3 pt;**
 - **Vector3 dir;**
- le **Segment**, défini par ses deux points *pt1* et *pt2*. Le segment est considéré orienté de *pt1* vers *pt2*.
 - **Vector3 pt1;**
 - **Vector3 pt2;**



- le **Triangle**, basé sur trois points de l'espace.
 - **Vector3 pt1;**
 - **Vector3 pt2;**
 - **Vector3 pt3;**
 - *On peut préférer un tableau de 3 points*
 - **Vector3[3] pts;**



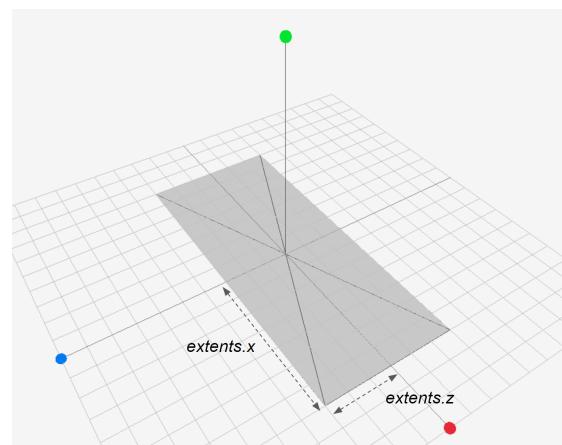
- le plan infini, nommé **Plane**, défini par un vecteur normal et une distance à l'origine signée. Le plan sera étudié en long et en large en cours. Pour afficher un plan infini dans la scène 3D, un *Quad* sera utilisé (voir ci-dessous). Il faudra faire un effort d'imagination pour étendre visuellement les limites du quad à l'infini.
 - *Vector3 n;*
 - *float d;*

Les objets suivants sont **munis d'un référentiel local**: une origine *O* et trois axes deux à deux perpendiculaires (*Ox*), (*Oy*) et (*Oz*) de vecteurs unitaires respectifs \vec{i} , \vec{j} , \vec{k} . Ils sont géométriquement **centrés sur l'origine** de leur référentiel, et d'orientation nulle (non tournés). L'axe (*Oy*) sera utilisé comme axe de symétrie pour les objets présentant une symétrie axiale. Pour translater et orienter l'objet, il suffira de translater l'origine du référentiel, et faire pivoter ses axes.

- En anglais référentiel se dit “reference frame”, la structure **ReferenceFrame** est constituée ainsi:
 - *Vector3 origin;*
 - *Vector3 i;*
 - *Vector3 j;*
 - *Vector3 k;*
 - *Quaternion q; // les explications viendront en temps et en heure...*
 - des constructeurs et méthodes appropriés permettent d'initialiser, de translater et de faire tourner aisément un référentiel (voir structure complète en Annexe 1)

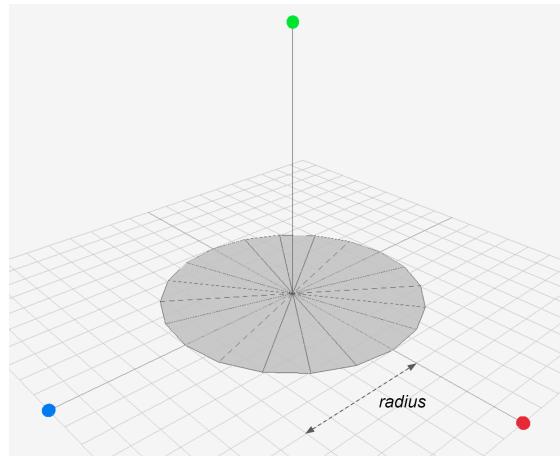
- le quadrilatère rectangle, appelé **Quad**, de vecteur normal \vec{j} , et dont la taille est définie par deux demi-longueurs, nommées les extensions (*extents*)

- *ReferenceFrame ref;*
- *Vector3 extents;*
- remarque: *extents.y* n'est pas utilisé



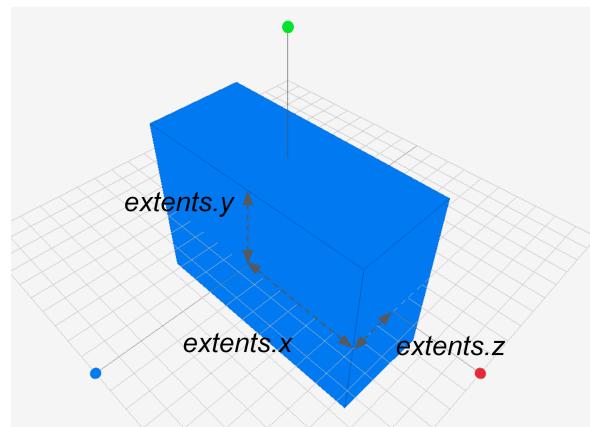
- le disque, nommé **Disk**, de vecteur normal \vec{j} , défini par son rayon

- *ReferenceFrame ref;*
- *float radius;*



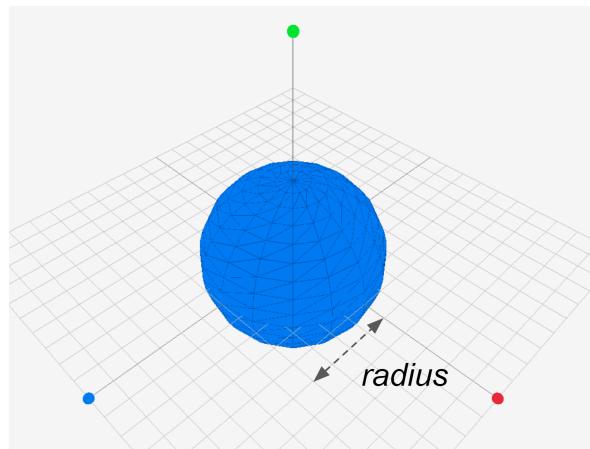
- le parallélépipède rectangle, appelé **Box**, défini par trois extensions

- *ReferenceFrame ref;*
- *Vector3 extents;*

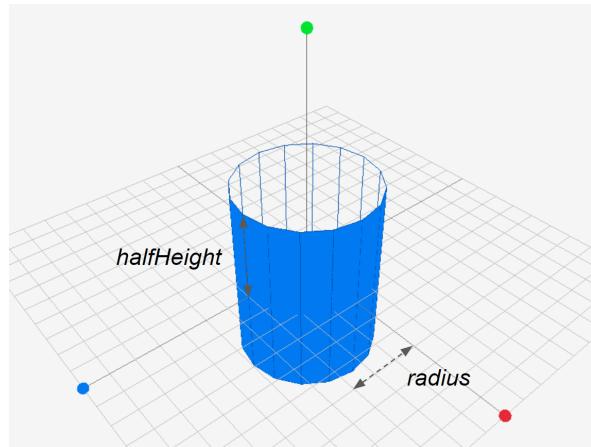


- la sphère **Sphere**, définie par son rayon

- *ReferenceFrame ref;*
- *float radius;*



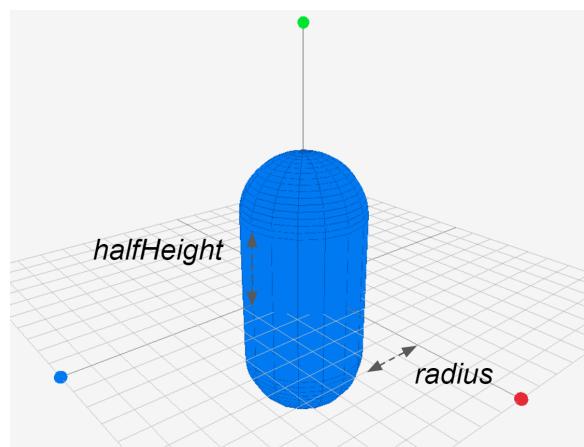
- le cylindre infini, ***InfiniteCylinder***, d'axe (*Oy*), défini par son rayon. Pour afficher un cylindre infini dans la scène 3D, un *Cylinder* ouvert sera utilisé (voir ci-dessous). Il faudra faire un effort d'imagination pour étendre visuellement les limites du cylindre à l'infini.
 - *ReferenceFrame ref;*
 - *float radius;*
- le cylindre ***Cylinder fini***, d'axe (*Oy*), défini par sa demi-hauteur et son rayon. Les extrémités peuvent être visuellement closes par deux extrémités discoïdes.
 - *ReferenceFrame ref;*
 - *float halfHeight;*
 - *float radius;*



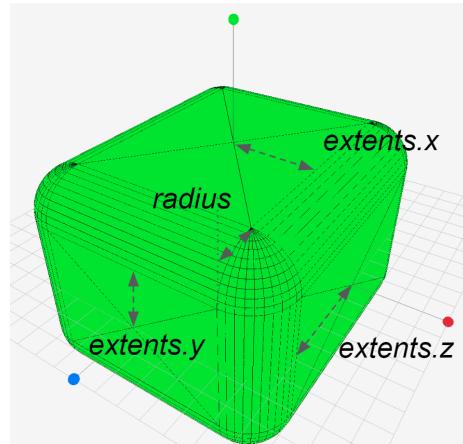
- On pourrait ajouter également le cône dans la liste des primitives 3D, mais celui-ci ne nous sera cependant d'aucune utilité.

Ajoutons à cette liste deux objets complexes

- La ***Capsule***, d'axe (*Oy*), constituée d'un corps cylindrique clos par deux hémisphères, et définie, comme le cylindre, par la demi-hauteur du corps cylindrique et son rayon. La hauteur totale de la capsule est donc $2 \times (\text{halfHeight} + \text{radius})$.
 - *ReferenceFrame ref;*
 - *float halfHeight;*
 - *float radius;*



- La boîte aux coins arrondis **RoundedBox**, définie par les extensions du noyau parallélépipédique et le rayon des coins arrondis. Le vecteur des extensions totales de la **RoundedBox** se calcule en ajoutant la valeur *radius* aux extensions du noyau parallélépipédique.
 - *ReferenceFrame ref;*
 - *Vector3 extents;*
 - *float radius;*



La **RoundedBox** sera utilisée lors des calculs de collision en tant que somme de *Minkovski* d'une **Box** et d'une **Sphere**.

Certaines de ces primitives seront étudiées mathématiquement en détails: **Line**, **Segment**, **Plane**, **Sphere & Cylinder**

Travail à réaliser: à l'exception de *Vector3* prédéfini dans raylib, et de *ReferenceFrame* donnée en Annexe 1, créez toutes les structures C nécessaires pour représenter les entités susmentionnées. Vous pouvez ajouter ces structures sous le code afférent aux systèmes de coordonnées *Cylindrical* et *Spherical*.

Mais vous pouvez également organiser votre code comme bon vous semble. Plutôt que de tout développer dans le fichier *core_basic_window_cpp.cpp*, n'hésitez pas à développer toutes ces structures dans un fichier *My3DPrimitives.h*.

Dans la section suivante vous allez développer les méthodes de traçage de ces primitives 3D.

Tracer des objets 3D en mode “immédiat” avec OpenGL

raylib s'appuie sur l'API graphique OpenGL pour rendre graphiquement les objets à l'écran.

L'enveloppe visible d'un objet 3D s'appelle un **maillage polygonal**, “3D mesh” en anglais, il est constitué de **faces** polygonales: quadrilatères ou **triangles**. Nous utiliserons des triangles. Chaque triangle est défini par 3 sommets nommés **vertices**. Les **arêtes** des faces peuvent être affichées en “fil de fer” (**wireframe**) pour une meilleure lisibilité de la perspective 3D de l'objet.

OpenGL est capable d'afficher à l'écran des points, des lignes, des triangles, des quadrilatères,etc:

- [GL_POINTS](#)
- [GL_LINES](#)
- [GL_TRIANGLES](#)
- [GL_QUADS](#)
- ...
- <https://docs.gl/gl3/glBegin>

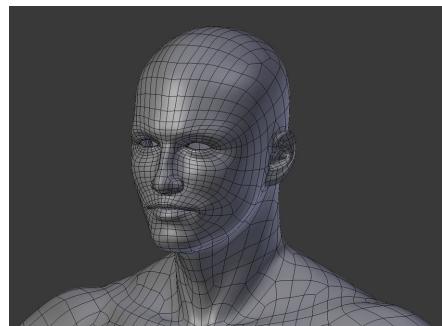
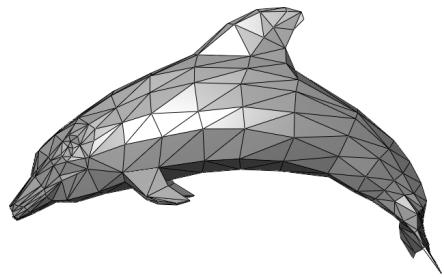
Pour visualiser nos objets, nous afficherons donc des triangles pour les faces, et des lignes pour les arêtes.

Deux modes d'affichage sont disponibles:

- un mode “immediate”: l'objet graphique 3D n'est pas défini et stocké en amont, les instructions pour son rendu graphique sont envoyées séquentiellement à la carte graphique en temps réel. Mode peu performant mais qui a l'avantage de la simplicité. Nous utiliserons ce mode d'affichage “**immediate**”.
- un mode “retained”: les données (vertices et faces) de l'objet 3D sont pré-calculées et stockées en mémoire. Ces données sont ensuite transmises d'un bloc à la carte graphique pour être rendues à l'écran. Gain de performance mais complexité de code accrue.

Les méthodes *DrawLine3D* et *DrawTriangle3D* prédéfinies par *raylib* permettent de tracer respectivement un segment et un triangle. Elles pourront être utilisées telles quelles si besoin. En voici les implémentations, qui illustrent l'utilisation des méthodes d'OpenGL, ainsi que les bases de l'affichage en “fil de fer” (**wireframe**) des arêtes, et de l'affichage polygonal des faces:

```
void DrawLine3D(Vector3 startPos, Vector3 endPos, Color color)
{
    rlBegin(RL_LINES);
    rlColor4ub(color.r, color.g, color.b, color.a);
    rlVertex3f(startPos.x, startPos.y, startPos.z);
    rlVertex3f(endPos.x, endPos.y, endPos.z);
```



```

    rlEnd();
}

```

Une séquence d'instructions de traçage commence par un appel à la méthode *rlBegin* et se termine par un appel à *rlEnd*. Le type d'objet tracé, RL_LINES ou RL_TRIANGLES est passé en paramètre de la méthode *rlBegin*. La méthode *rlVertex3f* envoie à la carte graphique les coordonnées d'une vertice. Un segment étant défini par deux vertices, deux appels à la méthode *rlVertex3f* sont donc nécessaires pour tracer le segment.

```

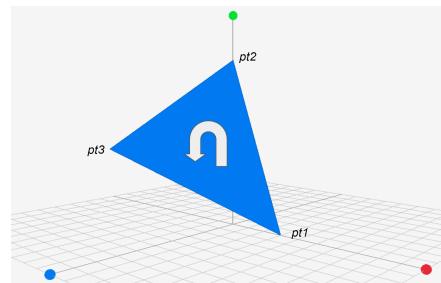
void DrawTriangle3D(Vector3 v1, Vector3 v2, Vector3 v3, Color color)
{
    if (rlCheckBufferLimit(3)) rlglDraw();

    rlBegin(RL_TRIANGLES);
    rlColor4ub(color.r, color.g, color.b, color.a);
    rlVertex3f(v1.x, v1.y, v1.z);
    rlVertex3f(v2.x, v2.y, v2.z);
    rlVertex3f(v3.x, v3.y, v3.z);
    rlEnd();
}

```

Une face triangulaire étant définie par trois vertices (sommets), trois appels à la méthode *rlVertex3f* sont donc nécessaires.

Les triangles sont orientés et ne sont visibles que d'un seul côté. Dans un espace muni d'un référentiel direct, les sommets du côté visible du triangle doivent être rentrés dans le sens trigonométrique.



La méthode *rlCheckBufferLimit* assure que la carte graphique n'est pas surchargée d'instructions de rendu. Si tel est le cas, un appel à la méthode *rlglDraw* permet de libérer la mémoire tampon en déclenchant le rendu graphique des instructions transmises préalablement. Le paramètre quantitatif de la méthode *rlCheckBufferLimit* correspond au nombre d'appels à la méthode *rlVertex3f* qui seront effectués dans la méthode de traçage en cours d'exécution. **Il est donc important de pouvoir en calculer le nombre exact.**

Pour les objets 3D munis d'un référentiel, la translation et la rotation de l'objet induites par ce référentiel local seront prises en charge par les méthodes *r/Translatef* et *r/Rotatef* d'OpenGL. La taille de l'objet pourra également être modifiée via la méthode *r/Scalef*. Les instructions de traçage devront alors être encadrées par un appel aux méthodes *r/PushMatrix* et *r/PopMatrix*.

Pour illustrer ceci, vous trouverez en Annexe 2 les méthodes d'affichage d'un *Quad*:

- `void MyDrawPolygonQuad(Quad quad, Color color = LIGHTGRAY)`
 - méthode de traçage des deux polygones triangulaires du *Quad*
- `void MyDrawWireframeQuad(Quad quad, Color color = DARKGRAY)`
 - méthode de traçage des 5 arêtes du *Quad*
- `void MyDrawQuad(Quad quad, bool drawPolygon = true, bool drawWireframe = true, Color polygonColor = LIGHTGRAY, Color wireframeColor = DARKGRAY)`
 - méthode générale de traçage du *Quad* (polygones + arêtes)

Toutes vos méthodes de traçage d'objets munis d'un référentiel seront développées selon le même modèle.

Vous remarquerez que le *Quad* effectivement tracé est normalisé, ses sommets ont des coordonnées égales à 0, 1 ou -1. Le dimensionnement du *Quad* est pris en charge par la méthode OpenGL *r/Scalef* : vous appliquerez ce principe aux objets 3D qui supportent le redimensionnement, soit tous sauf la *Capsule* et la *RoundedBox*.

Il ne vous reste plus qu'à développer vos propres méthodes de traçage:

Disk:

- `void MyDrawPolygonDisk(Disk disk, int nSectors, Color color = LIGHTGRAY)`
- `void MyDrawWireframeDisk(Disk disk, int nSectors, Color color = DARKGRAY)`
- `void MyDrawDisk(Disk disk, int nSectors, bool drawPolygon = true, bool drawWireframe = true, Color polygonColor = LIGHTGRAY, Color wireframeColor = DARKGRAY)`

Box:

- `void MyDrawPolygonBox(Box box, Color color = LIGHTGRAY)`
- `void MyDrawWireframeBox(Box box, Color color = DARKGRAY)`
- `void MyDrawBox(Box box, bool drawPolygon = true, bool drawWireframe = true, Color polygonColor = LIGHTGRAY, Color wireframeColor = DARKGRAY)`

Sphere:

- `void MyDrawPolygonSphere(Sphere sphere, int nMeridians, int nParallels, Color color = LIGHTGRAY)`
- `void MyDrawWireframeSphere(Sphere sphere, int nMeridians, int nParallels, Color color = DARKGRAY)`
- `void MyDrawSphere(Sphere sphere, int nMeridians, int nParallels, bool drawPolygon = true, bool drawWireframe = true, Color polygonColor = LIGHTGRAY, Color wireframeColor = DARKGRAY)`

Cylinder:

- void **MyDrawPolygonCylinder**(Cylinder cylinder, int nSectors, bool drawCaps = false, Color color = LIGHTGRAY)
- void **MyDrawWireframeCylinder**(Cylinder cylinder, int nSectors, bool drawCaps = false, Color color = LIGHTGRAY)
- void **MyDrawCylinder**(Cylinder cylinder, int nSectors, bool drawCaps=false, bool drawPolygon = true, bool drawWireframe = true, Color polygonColor = LIGHTGRAY, Color wireframeColor = DARKGRAY)

Capsule:

- void **MyDrawPolygonCapsule**(Capsule capsule, int nSectors, int nParallels, Color color = LIGHTGRAY)
- void **MyDrawWireframeCapsule**(Capsule capsule, int nSectors, int nParallels, Color color = LIGHTGRAY)
- void **MyDrawCapsule**(Capsule capsule, int nSectors, int nParallels, bool drawPolygon = true, bool drawWireframe = true, Color polygonColor = LIGHTGRAY, Color wireframeColor = DARKGRAY)

RoundedBox:

- void **MyDrawPolygonRoundedBox**(RoundedBox roundedBox, int nSectors, Color color = LIGHTGRAY)
- void **MyDrawWireframeRoundedBox**(RoundedBox roundedBox, int nSectors, Color color = LIGHTGRAY)
- void **MyDrawRoundedBox**(RoundedBox roundedBox, int nSectors, bool drawPolygon = true, bool drawWireframe = true, Color polygonColor = LIGHTGRAY, Color wireframeColor = DARKGRAY)

Appuyez-vous sur les méthodes mathématiques de *raylib* du fichier *raymath.h*. Créez vos propres méthodes mathématiques si besoin.

Certaines de ces méthodes de traçage existent déjà dans le fichier *models.c*, vous pouvez vous en inspirer. Sachez cependant qu'elles ne sont pas optimisées et n'utilisent pas les systèmes de coordonnées *Cylindrical* et *Spherical*, il est donc préférable (obligatoire en fait) de les réécrire. Utilisez pour chaque objet 3D le système de coordonnées approprié.

- coordonnées cartésiennes pour *Box*,
- coordonnées cylindriques pour *Disk & Cylinder*,
- coordonnées sphériques pour *Sphere*

La méthode *rlVertexf* n'acceptant cependant que les coordonnées cartésiennes, vous n'aurez pas le choix *in fine* que d'effectuer une conversion finale vers le système cartésien grâce à vos méthodes de conversion.

Un bon principe d'optimisation de vos méthodes: le même calcul ne devrait jamais être réalisé deux fois. Stockez sa valeur s'il doit être réutilisé.

Les méthodes de traçage se ressemblent parfois fortement, utilisez la fonctionnalité d'édition copier/coller pour gagner du temps.

Organisez votre code comme bon vous semble. Plutôt que de tout développer dans le fichier *core_basic_window_cpp.cpp*, n'hésitez pas à développer toutes ces méthodes de traçage dans un fichier *MyDrawingMethods.h*.

Pour chaque méthode de traçage, écrivez un code de test correspondant entre les appels aux méthodes *BeginMode3D* et *EndMode3D* de la méthode *main*. En Annexe 2 figure l'exemple du code de test pour l'affichage d'un *Quad*.

En Annexe 4 figurent quelques éléments de langage C++ dont vous pourrez avoir besoin.

Pour aller plus loin:

Une *RoundedBox* est visuellement constituée de portions de cylindres et de portions de sphères au niveau de ses arêtes et de ses sommets. Vous afficherez dans un premier temps des cylindres et des sphères complètes, mais il serait bon ultérieurement d'optimiser cet affichage en n'affichant que ce qui est nécessaire. Il vous faudra donc développer des méthodes qui tracent une portion de cylindre (ouvert, pas de chapeau) et une portion de sphère. En voici les prototypes:

Portion de cylindre

- `void MyDrawPolygonCylinderPortion(Cylinder cylinder, int nSectors, float startTheta, float endTheta, Color color = LIGHTGRAY)`
- `void MyDrawWireframeCylinderPortion(Cylinder cylinder, int nSectors, float startTheta, float endTheta, Color color = LIGHTGRAY)`
- `void MyDrawCylinderPortion(Cylinder cylinder, int nSectors, float startTheta, float endTheta, bool drawPolygon = true, bool drawWireframe = true, Color polygonColor = LIGHTGRAY, Color wireframeColor = DARKGRAY)`

Portion de sphère

- `void MyDrawPolygonSpherePortion(Sphere sphere, int nMeridians, int nParallels, float startTheta, float endTheta, float startPhi, float endPhi, Color color = LIGHTGRAY)`
- `void MyDrawWireframeSpherePortion(Sphere sphere, int nMeridians, int nParallels, float startTheta, float endTheta, float startPhi, float endPhi, Color color = LIGHTGRAY)`
- `void MyDrawSpherePortion(Sphere sphere, int nMeridians, int nParallels, float startTheta, float endTheta, float startPhi, float endPhi, bool drawPolygon = true, bool drawWireframe = true, Color polygonColor = LIGHTGRAY, Color wireframeColor = DARKGRAY)`

Lorsque vous aurez développé ces méthodes, vous pourrez les appeler pour simplifier les méthodes de traçage du cylindre entier et de la sphère entière.

Annexe 1

Structure *ReferenceFrame*

```
struct ReferenceFrame {
    Vector3 origin;
    Vector3 i, j, k;
    Quaternion q;

    ReferenceFrame()
    {
        origin = { 0,0,0 };
        i = { 1,0,0 };
        j = { 0,1,0 };
        k = { 0,0,1 };
        q = QuaternionIdentity();
    }

    ReferenceFrame(Vector3 origin, Quaternion q)
    {
        this->q = q;
        this->origin = origin;
        i = Vector3RotateByQuaternion({ 1,0,0 }, q);
        j = Vector3RotateByQuaternion({ 0,1,0 }, q);
        k = Vector3RotateByQuaternion({ 0,0,1 }, q);
    }

    void Translate(Vector3 vect)
    {
        this->origin = Vector3Add(this->origin, vect);
    }

    void RotateByQuaternion(Quaternion qRot)
    {
        q = QuaternionMultiply(qRot, q);
        i = Vector3RotateByQuaternion({ 1,0,0 }, q);
        j = Vector3RotateByQuaternion({ 0,1,0 }, q);
        k = Vector3RotateByQuaternion({ 0,0,1 }, q);
    }
};
```

Annexe 2

Les méthodes d'affichage d'un Quad:

- `void MyDrawPolygonQuad(Quad quad, Color color = LIGHTGRAY)`
- `void MyDrawWireframeQuad(Quad quad, Color color = DARKGRAY)`
- `void MyDrawQuad(Quad quad, bool drawPolygon = true, bool drawWireframe = true, Color polygonColor = LIGHTGRAY, Color wireframeColor = DARKGRAY)`

```
//QUAD
void MyDrawPolygonQuad(Quad quad, Color color = LIGHTGRAY)
{
    int numVertex = 6;
    if (rlCheckBufferLimit(numVertex)) rlglDraw();

    // BEGINNING OF SPACE TRANSFORMATION INDUCED BY THE LOCAL
    // REFERENCE FRAME
    // methods should be called in this order: rlTranslatef, rlRotatef & rlScalef
    // so that transformations occur in the opposite order: scale, then rotation, then
    translation
    rlPushMatrix();

    //TRANSLATION
    rlTranslatef(quad.ref.origin.x, quad.ref.origin.y, quad.ref.origin.z);

    //ROTATION
    Vector3 vect;
    float angle;
    QuaternionToAxisAngle(quad.ref.q, &vect, &angle);
    rlRotatef(angle * RAD2DEG, vect.x, vect.y, vect.z);

    //SCALING
    rlScalef(quad.extents.x, 1, quad.extents.z);
    // END OF SPACE TRANSFORMATION INDUCED BY THE LOCAL REFERENCE
    FRAME

    rlBegin(RL_TRIANGLES);
    rlColor4ub(color.r, color.g, color.b, color.a);

    rlVertex3f(1,0,1);
    rlVertex3f(1,0,-1);
    rlVertex3f(-1,0,-1);

    rlVertex3f(1, 0, 1);
    rlVertex3f(-1, 0, -1);
    rlVertex3f(-1, 0, 1);

    rlEnd();
```

```

//EVERY rlPushMatrix method call should be followed by a rlPopMatrix method call
rlPopMatrix();

}

void MyDrawWireframeQuad(Quad quad, Color color = DARKGRAY)
{
    int numVertex = 10;
    if (rlCheckBufferLimit(numVertex)) rlglDraw();

    rlPushMatrix();
    rlTranslatef(quad.ref.origin.x, quad.ref.origin.y, quad.ref.origin.z);

    Vector3 vect;
    float angle;
    QuaternionToAxisAngle(quad.ref.q, &vect, &angle);
    rlRotatef(angle * RAD2DEG, vect.x, vect.y, vect.z);

    rlScalef(quad.extents.x, 1, quad.extents.z);

    rlBegin(RL_LINES);
    rlColor4ub(color.r, color.g, color.b, color.a);

    rlVertex3f(1, 0, 1);
    rlVertex3f(1, 0, -1);

    rlVertex3f(1, 0, -1);
    rlVertex3f(-1, 0, -1);

    rlVertex3f(-1, 0, -1);
    rlVertex3f(1, 0, 1);

    rlVertex3f(-1, 0, 1);
    rlVertex3f(1, 0, 1);

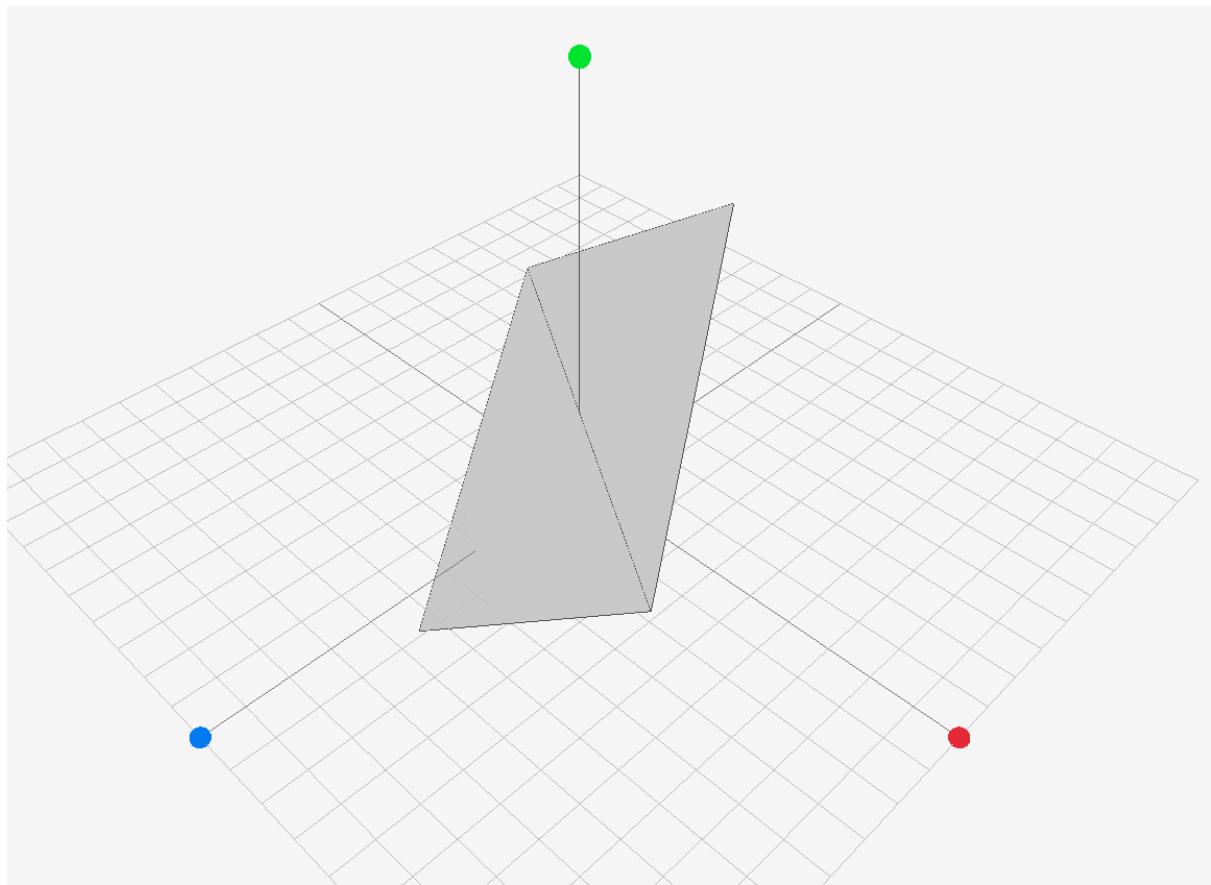
    rlEnd();
    rlPopMatrix();
}

void MyDrawQuad(Quad quad, bool drawPolygon = true, bool drawWireframe = true,
Color polygonColor = LIGHTGRAY, Color wireframeColor = DARKGRAY)
{
    if (drawPolygon) MyDrawPolygonQuad(quad, polygonColor);
    if (drawWireframe) MyDrawWireframeQuad(quad, wireframeColor);
}

```

Voici un bout de code pour tester ces méthodes, à insérer entre les appels aux méthodes *BeginMode3D* et *EndMode3D* de la méthode *main*, juste avant les instructions de traçage du référentiel:

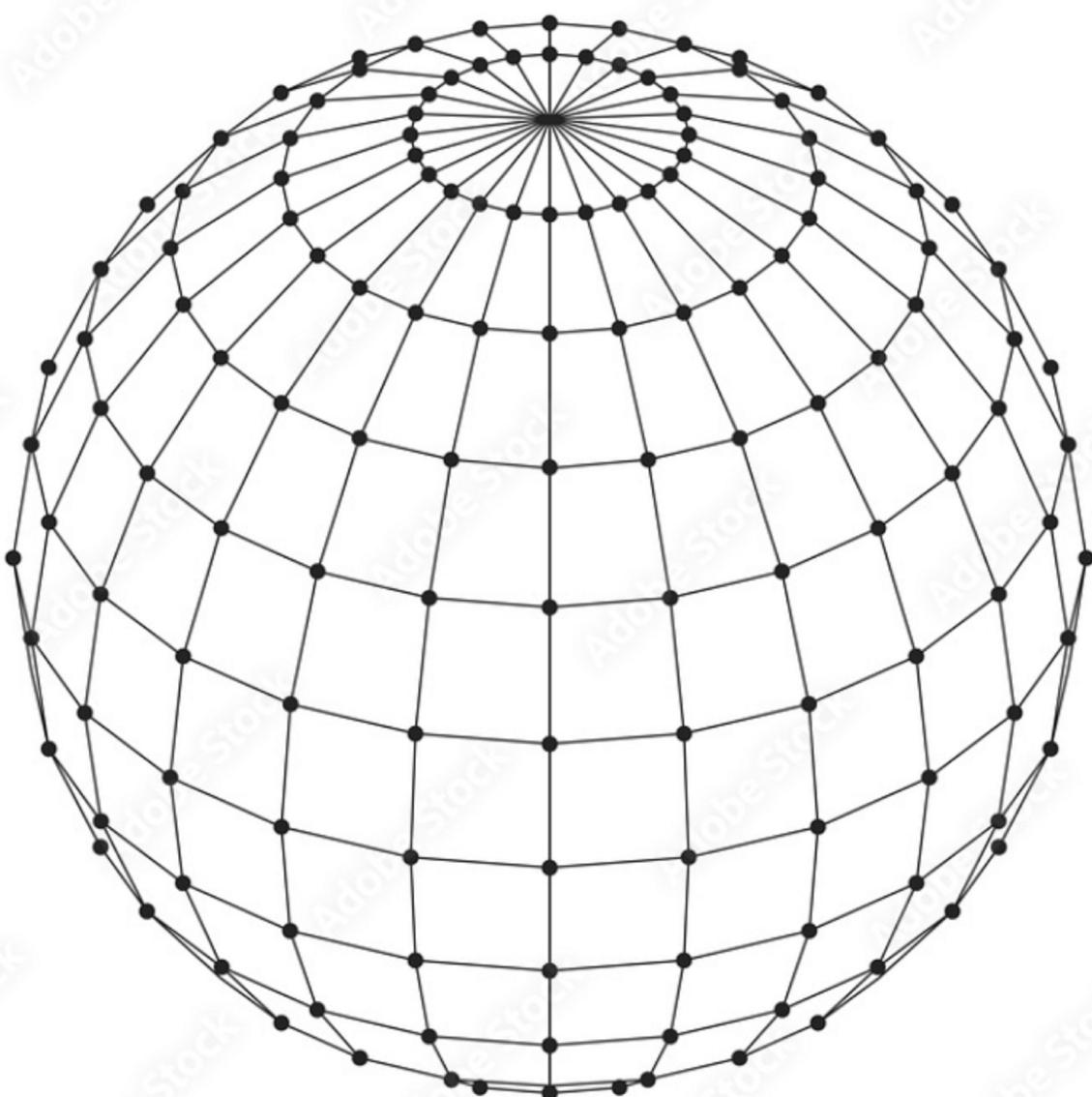
```
// QUAD DISPLAY TEST
ReferenceFrame ref = ReferenceFrame(
    { 0,2,0 },
    QuaternionFromAxisAngle(Vector3Normalize( { 1,1,1 } ), PI / 4));
Quad quad = { ref,{3,1,5} };
MyDrawQuad(quad);
```



Annexe 3

Une sphère 3D standard. Seuls les quadrilatères ont été dessinés pour des raisons de lisibilité, mais vous devez imaginer que chaque quadrilatère est en réalité constitué de deux triangles.

Utilisez ce schéma pour vous aider dans la réflexion, la conception et l'**optimisation** des algorithmes de traçage de la sphère.



Annexe 4

Quelques éléments de langage utiles et spécifiques (?) au C/C++. Cette liste n'est pas exhaustive, n'hésitez pas à me faire part d'autres éléments de langage qui vous auront été utiles.

- Nous n'utilisons aucune fonctionnalité Orientée Objet du C++.
 - Il ne sera donc pas utile de créer des classes
- Des structures suffisent.
- Vous pouvez ajouter des méthodes dans vos structures: voir pour cela l'exemple de la structure *ReferenceFrame* en Annexe 1.
- Au sein d'une méthode de structure, utilisez “*this->*” pour accéder à un champ de la structure si un paramètre de la méthode porte le même nom que le champ: voir pour cela l'exemple de la structure *ReferenceFrame* en Annexe 1.
- Nous n'utilisons pas les pointeurs.
- Nous utiliserons les *références* pour les algorithmes de calcul d'intersection, mais pas pour les méthodes de traçage, donc pour le moment vous ne vous en souciez pas. La totalité des paramètres de nos méthodes sont donc passés pour le moment par valeur

Initialisation d'une variable d'un type *struct*

Écrivez les valeurs entre accolades, dans l'ordre de définition des champs de la structure.

Exemple avec la structure *Plane*

```
struct Plane
{
    Vector3 n;
    float d;
};
```

L'initialisation d'une variable de type *Plane* pourra ressembler à cela:

Plane plane = { {0,1,0} , 3 }; // en 1er le vecteur normal, puis la distance signée

Initialisation d'un tableau de taille fixe

Exemple avec un tableau de *Vector3*

```
Vector3 vertices[2] = { {0,1,0}, {-5,1,0} };
```

Tableau dynamique

Le type C++ `std::vector<T>` permet de créer des tableaux dynamiques pour stocker tous types d'éléments

Un exemple avec un tableau de *float*

```
std::vector<float> array = { 1.0f, -3.5f, 9.7f };           // initialisation
array.push_back(2.0f);          // ajout d'un élément en fin de tableau
printf("size of the array = %d\n", array.size());        // taille du tableau
for (int i = 0; i < array.size(); i++)
    printf("element at index %d: %f\n", i, array[i]);      // accesseur
array.clear();            // vider le tableau

std::fill(array.begin(), array.end(), -1.0f);       // remplir un tableau avec une valeur
```

Valeurs min & max du type *float*

float maxFloatValue = std::numeric_limits<float>::max(); // utile pour initialiser une
“distance minimum” dans une boucle de recherche de l’objet le plus “proche”

float minFloatValue = std::numeric_limits<float>::min(); // utile pour initialiser une
“distance maximum” dans une boucle de recherche de l’objet le plus “éloigné”

Méthodes mathématiques usuelles

Utilisez les méthodes manipulant le type *float*, nous n’avons pas besoin de la précision mathématique apportée par le type *double*: *fabsf*, *sqrif*, *cosf*, *sinf*, *tanf*, *acosf*, *asinf*, *atanf*,...

<https://en.cppreference.com/w/c/numeric/math>