TP C#13 : Raymarcher

Consignes de rendu

A la fin de ce TP, vous devrez rendre une archive respectant l'architecture suivante :

```
rendu-tp-prenom.nom.zip
|-- prenom.nom/
    |-- README
    |-- Raymarcher/
        |-- Raymarcher.sln
        |-- Raymarcher/
            |-- Everything except bin/ and obj/
```

N'oubliez pas de vérifier les points suivants avant de rendre :

— Remplacez `prenom.nom` par votre propre login et n'oubliez pas le fichier `AUTHORS`.
— Les fichiers `AUTHORS` et `README` sont obligatoires.
— Pas de dossiers `bin` ou `obj` dans le projet.
— Respectez scrupuleusement les prototypes demandés.
— Retirez tous les tests de votre code.
— **Le code doit compiler !**

README

Vous devez écrire dans ce fichier tout commentaire sur le TP, votre travail, ou plus généralement vos forces / faiblesses, vous devez lister et expliquer tous les boni que vous aurez implémentés. Un `README` vide sera considéré comme une archive invalide (malus).

1 Introduction

1.1 Objectifs

Pour ce TP, vous n'allez pas apprendre de notions liées au C# mais plutôt un sujet qui va utiliser certaines de vos connaissances en mathématiques appliquées à certains algorithmes afin de faire un programme capable de faire le rendu d'une scène en 3D. Ce TP ne va cependant pas aborder la méthode de raytracing, mais une méthode plus simple, et plus adaptée à un TP d'une semaine. Une tarball sera fournie avec toute la structure du programme que vous aurez à écrire ainsi que toutes les parties qui ont été jugées superflues pour les mettre dans le sujet mais qui sont importantes dans la structure du programme (un parser pour gérer les fichiers des scènes et les primitives pour gérer la géométrie 3D principalement).

2 Cours

2.1 Un peu de théorie sur le rendu 3D

Il existe plusieurs techniques de rendu 3D, chacune avec leurs forces et leurs faiblesses. Nous allons commencer par une liste non exhaustive de certaines de ces techniques.

2.1.1 Rasterizing

Le rasterizing est une technique de rendu qui consiste à utiliser des opérations mathématiques pour transformer les coordonées en 3D d'une scène en coordonées en 2D à l'écran. Cette méthode possède un énorme avantage face aux autres méthodes qui seront évoquées dans cette liste : elle permet de faire un rendu extrêmement rapide car les calculs se font majoritarement sur un espace en 2D. C'est la méthode qui est utilisée pour la plupart des jeux vidéo. Elle est cependant un peu plus technique que les autres méthodes qui seront présentées car elle demande de faire beaucoup de calculs matriciels pour gérer le passage que la 3D à la 2D. La gestion des shaders (les ombres et lumières) est aussi plus complexe, ce qui pose problème si on veut obtenir un rendu photoréaliste. Les cartes graphiques sont souvent optimisées pour cette méthode de rendu et possèdent des instructions spécifiques pour accélérer le temps de rendu au maximum.

2.1.2 Raytracing

Le raytracing a un concept assez intuitif : on fait partir un rayon depuis la caméra à travers chaque pixel et on lui fait parcourir la scène jusqu'à ce qu'il rencontre un objet. On peut aussi le faire rebondir sur des surfaces ou simuler de la réfraction. Cette technique est relativement simple à implémenter mais est beaucoup plus lourde à calculer. Elle n'est donc pas adaptée à un rendu temps réel, sauf sur certaines cartes graphiques, qui sont optimisées pour en faire. Cette technique possède aussi le gros avantage qui est que les rendus sont de bien meilleure qualité, ce qui fait que cette méthode est la plus utilisée pour faire les rendus de films d'animation, qui parfois ont besoin d'un rendu photoréaliste.

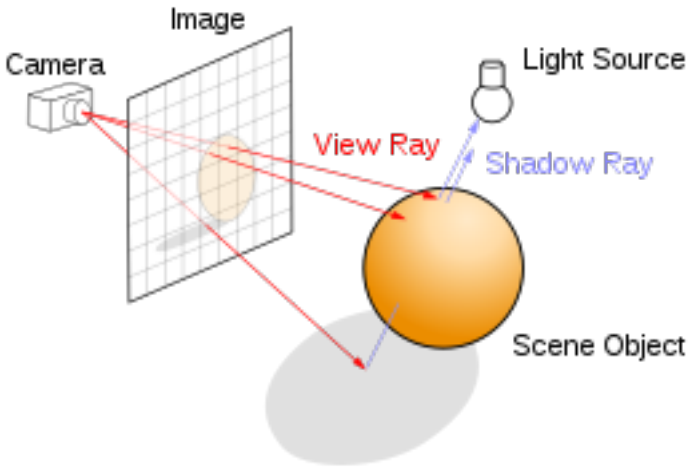


Illustration du raytracing sur wikipedia

2.1.3 Raymarching

Le raymarching est une version simplifiée du raytracing, et c'est cette méthode que vous aurez à implémenter pour ce TP. Le principe de base est le même : on fait partir un rayon de la caméra et qui passe à travers un pixel et on cherche une intersection. Cependant, cette intersection n'est pas calculée directement pour chaque triangle comme pour le raytracer. À la place, on avance petit à petit le long du rayon et on teste si le point que l'on obtient est dans un des modèles de la scène. Si le point est dans la géométrie de la scène, on s'arrête et on marque le point comme point d'intersection, sinon on avance et on refait un test. Cette méthode est très simple à implémenter et est très puissante quand on veut rendre certaines fractales ou une quantité infinie d'objets si on la mélange à une méthode qui permet de définir des formes en 3D par des fonctions mathématiques.

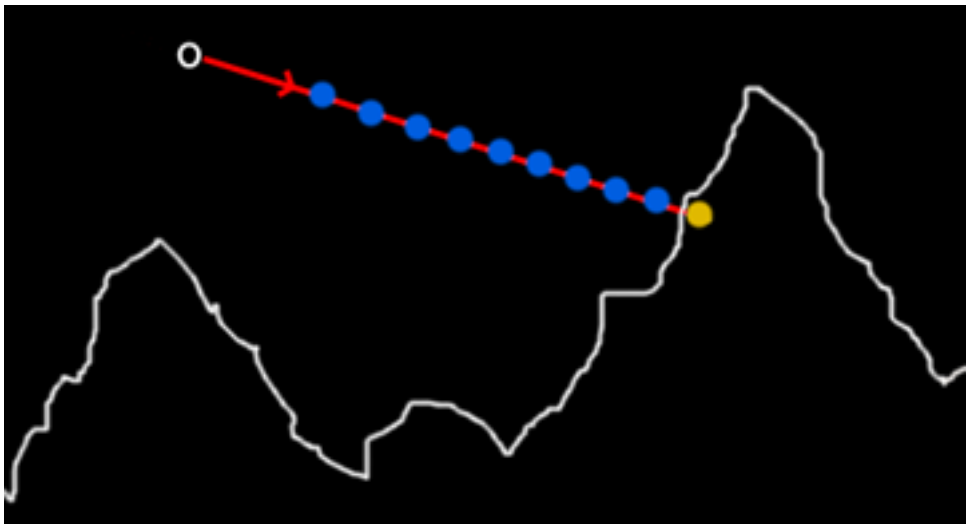


Illustration de la méthode de raymarching (Inigo Quillez)

2.2 Raymarching avec des fonctions de distance

2.2.1 Fonctions de distance signées

Une fonction de distance signée, abrégée SDF (Signed Distance Function) est une fonction mathématique qui permet de définir une forme géométrique (en 2D ou 3D) par une fonction

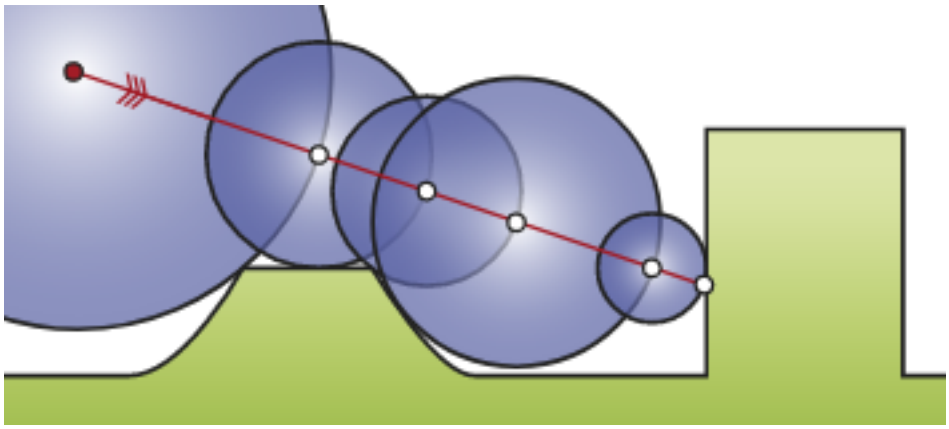
mathématique. La fonction de distance d'une sphère sera par exemple, pour un point p et une sphère de rayon r centrée sur l'origine :

$$d = \sqrt{x_p^2 + y_p^2 + z_p^2} - r$$

Ainsi, la fonction de distance retourne la distance entre le point qui lui est donné en paramètre et la surface de la forme qu'elle définit. Cette distance est positive si le point se trouve en dehors de la forme et négative si le point se trouve à l'intérieur.

2.2.2 En quoi les fonctions de distance peuvent nous aider ?

Toute la puissance du raymarching mélangé aux fonctions de distance est que pour chaque point de l'espace, on peut connaître avec précision à quelle distance ce point se trouve d'un objet. Par conséquent, l'algorithme pour avancer le long du rayon ne va pas utiliser une distance constante, mais le résultat calculé par la fonction de distance, ce qui fait que l'on va perdre beaucoup moins de temps à traverser les parties du rayon qui sont loin des objets de la scène.



Raymarching avec des fonctions de distance (*Enhanced Sphere Tracing*)

2.2.3 Les opérations sur les fonctions de distance

Les fonctions de distances sont utiles pour tracer une sphère, ou un cube, ou d'autres formes simples. Cependant, cela ne nous permet que de tracer une seule primitive (un seul objet "simple") sur la scène, ce qui est très limité.

Il existe une solution à ce problème. En effet, comme ce sont des fonctions, nous pouvons facilement appliquer des opérateurs simples à calculer sur ces fonctions. Les 3 opérateurs qui seront à votre disposition dans ce TP seront l'union, l'intersection, et la soustraction. Il existe d'autres opérateurs binaires qui sont des variations de ces 3 opérateurs.

Il existe aussi des opérations unaires sur les fonctions de distance, qui permettent de modifier la forme d'une autre fonction de distance. Cela permet d'augmenter la complexité des scènes produites. Grâce aux différentes primitives et aux opérations que l'on peut appliquer sur les fonctions de distance, il est possible de créer des scènes complexes.

Il est néamoins difficile de produire une scène complexe car la géométrie est implicite, et il est nécessaire de bien maîtriser les fonctions mathématiques qui sont utilisées pour obtenir le résultat souhaité. Par conséquent, les scènes fournies avec la tarball de ce TP seront simples, mais suffisantes pour voir si votre raymarcher fonctionne correctement. Cependant, rien ne vous empêche d'écrire de nouveaux fichiers pour créer vos propres scènes.

2.2.4 Comment utiliser une fonction de distance pour un objet qui ne se trouve pas à l'origine ?

Si on reprend la formule de la fonction de distance de la sphère :

$$d = \sqrt{x_p^2 + y_p^2 + z_p^2} - r$$

On peut constater que cette formule ne peut fonctionner que pour une sphère qui se trouve à l'origine du repère. Le problème est que nous voulons probablement créer des sphères dont le centre ne se trouve pas à l'origine du repère, mais la formule que nous avons est simple et nous ne voulons pas forcément la modifier. La solution est donc de transformer les coordonnées du point en entrée pour les passer dans un repère local centré sur la sphère. Ainsi, le code qui permet de gérer la sphère pour notre raymarcher ressemblera à :

```
1         public override float Evaluate(Vector3 point)
2         {
3             //Transform point coordinates into local coordinates
4             point = point - Center;
5             //Evaluate the distance function with the simple formula
6             return (point).Magnitude() - Radius;
7         }
```

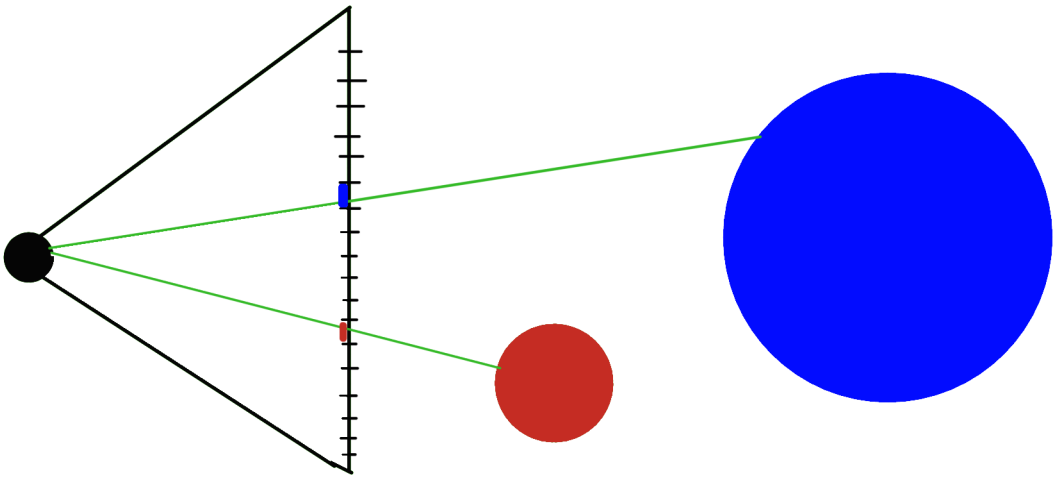
Comme vous pouvez le voir, le code n'est pas beaucoup plus compliqué, le passage aux coordonnées locales ne demande que des soustractions, qui sont simples et rapides à faire, et nous n'avons pas eu besoin de modifier la formule d'origine. Cela n'est pas trop visible sur une sphère car c'est une fonction très simple, mais sur des fonctions plus compliquées, la formule pourrait devenir beaucoup plus compliquée à écrire et le temps de calcul serait rallongé.

2.2.5 La gestion de la caméra

Représenter une scène dans l'espace n'est pas très compliqué : on peut utiliser des **meshs**, qui sont des listes de triangles dans l'espace avec un peu de données autour, ou bien des fonctions mathématiques comme pour le raymarcher, ou d'autres méthodes plus exotiques. Cependant, afin de pouvoir obtenir une image de cette scène, nous devons définir d'où nous observons la scène, dans quelle direction nous regardons, et à quel angle autour de nous nous pouvons voir la scène. C'est exactement pour atteindre ces objectifs que nous avons besoin d'une caméra. La caméra joue le rôle de nos yeux dans cet espace virtuel que nous essayons d'observer. Elle possède donc une position, une direction et un angle de vue, de la même façon qu'une caméra réelle est placée quelque part, regarde en direction de quelque chose, et peut voir avec un angle de vue plus ou moins important. Nous appellerons la position de la caméra son origine.

Afin de pouvoir lancer nos rayons et savoir dans quelle direction les lancer, nous avons besoin de positionner notre écran dans la scène. Cet écran est la représentation virtuelle de l'écran à travers lequel nous observons la scène dans cette dernière. L'écran est placé directement devant la caméra (en partant de l'origine, en avançant un peu en direction de là où regarde la caméra. Nous pourrons alors construire un plan avec son propre système de coordonnées pour connaître la position de chaque pixel de l'image dans l'espace. Ainsi, il nous suffit de calculer le vecteur partant de l'origine de la caméra et allant en direction d'un pixel précis pour obtenir le rayon qui passe par ce pixel.

rays.png



Chaque pixel correspond à un rayon partant de l'origine de la caméra, et le pixel prendra la couleur de la surface touchée.

2.2.6 Parcourir le rayon

Une fois le rayon pour un pixel obtenu, nous devons le parcourir, puis pour chaque point sur lequel on s'arrête, regarder si il est dans un objet ou non. Comme nous n'allons pas nécessairement croiser un objet, il est nécessaire de mettre en place des conditions d'arrêt. La plus simple est de s'arrêter après un nombre fixe d'étapes. On peut aussi de s'arrêter après avoir parcouru une certaine distance. Il est possible possible d'utiliser ces deux conditions d'arrêt en même temps. La condition d'arrêt pour quand on rencontre un objet est assez simple, elle devient vraie si on est assez proche de la limite de l'objet (quand la valeur retournée par la fonction de distance est assez proche de 0) ou si on se trouve dans un objet (si la valeur retournée par la fonction de distance est négative). La deuxième condition va devenir problématique si on peut gérer la réfraction ou d'autres types de shaders nécessitant de passer à travers un objet. Il faudra alors probablement retirer cette condition.

2.2.7 Gestion de la lumière

Une fois que l'on a trouvé un point d'intersection pour un pixel, on peut calculer sa couleur en fonction des conditions d'éclairage. Il faut considérer que toutes les couleurs sont gérées avec des nombres flottants entre 0 et 1 car cela permet d'avoir des équations qui fonctionnent mieux.

Lumière diffuse La méthode d'éclairage la plus simple est la lumière ambiante, elle consiste à ajouter une quantité fixe de lumière sur toute la scène (tous les pixels avec un point d'intersection reçoivent l'éclairage diffus, les autres restent noirs)

Lumière directionelle Une lumière directionnelle est une lumière qui possède seulement une direction. Elle éclaire uniformément la scène de la même intensité et cette intensité ne dépend pas de la distance à la source de lumière. On peut considérer qu'une lumière directionnelle agit comme le soleil.

Composition des couleurs Afin de mélanger la lumière diffuse à la lumière directionnelle, nous devons les composer. Ainsi, pour un pixel donné, en considérant la couleur initiale du matériaux **matColor**, **diffuse** la couleur de la lumière diffuse, et **ambient** la couleur de la lumière ambiente, sa couleur finale sera :

$$color = matColor * diffuse + matColor * ambient$$

Pour aller plus loin

Cette formule a été simplifiée pour fonctionner de façon simple avec notre raymarcher, qui ne contient pas grand chose. Si vous voulez avoir plus d'information sur le sujet de l'éclairage, vous pouvez faire des recherches sur le modèle **Phong**, qui est un des modèles utilisés pour la gestion de l'éclairage pour le rendu 3D.

2.3 Le format de fichier utilisé pour décrire une scène

Attention !

Les axes x et z sont les axes horizontaux et l'axe y est l'axe vertical.

Afin de pouvoir créer des scènes facilement, nous avons créé un format de fichier qui permettra de décrire chaque objet dans la scène, les opérateurs utilisés, la position et la direction de la caméra, et enfin la direction de la source de lumière.

Le format de fichier utilisé pour ce tp est composé d'un en-tête comportant la position et la direction de la caméra, l'angle de vue de la caméra et la direction de la lumière. Cet en-tête est suivi d'une ligne vide et de la description de la scène, qui suit la syntaxe suivante :

```
function ::= object | operator | modificator
object ::= (object_name [param*])
operator ::= (op_name [param*] (function)(function))
modificator ::= (mod_name [param*] (function))
```

Ainsi, un fichier de scène peut ressembler à ceci :

```
0 0 0 //camera position
1 0 0 //camera direction
90 //camera field of view
1 -1 0.5 //light direction

(union
    (box 0 -3 0 60 2 60)
    (union
        (box 10 0 0 0.5 4 10)
        (box 5 0 5 10 4 0.5)))
```

Voici la liste des objets et des opérateurs fournis dans le parser de la tarball. Rien ne vous empêche d'ajouter d'autres primitives/opérateurs/modificateurs en ajoutant les classes correspondantes et en les ajoutant dans la classe **NodeFactory** si vous le souhaitez.

Sphère

```
(sphere <centerX> <centerY> <centerZ> <radius>)
```

Boite

```
(box <centerX> <centerY> <centerZ> <width> <height> <depth>)
```

Boite arrondie

```
(rbox <centerX> <centerY> <centerZ> <width> <height> <depth> <radius>)
```

Union

```
(union (<left>)(<right>))
```

Soustraction

```
(sub (<left>)(<right>))
```

Intersection

```
(inter (<left>)(<right>))
```

Conseil

Des fichiers de scènes sont fournis dans la tarball, mais nous vous invitons à en ajouter pour avoir vos propres cas de tests. Les rotations des objets n'ont pas été gérées dans les fichiers fournis.

3 Exercices

Pour ce TP, vous devez commencer par télécharger la tarball fournie sur l'intranet, elle contient tous les fichiers dont vous aurez besoin ainsi que toutes les classes qui permettent de gérer la scène (le parser et les différentes fonctions de distances ainsi que quelques fonctions utilitaires qui pourront vous servir). Le main est fourni, la commande pour lancer votre programme est :

```
mono raymarcher.exe <input_file> <res_x> <res_y> <output_file>
```

Avec **input_file** le nom du fichier décrivant la scène, **res_x** et **res_y** la résolution de l'image en largeur (x) et en hauteur (y), et **output_file** le nom du fichier en sortie. Il vous sera demandé d'utiliser la classe **Bitmap** ; par conséquent, le fichier de sortie sera une bitmap.

3.1 Vecteurs

Commençons par un outil essentiel de votre raymarcher, qui vous sera utile pendant toute la suite du TP : les vecteurs. Vous aurez donc à remplir la classe **Vector3**. Les fonctions qui vous seront demandées sont :

— **Dot :** Produit scalaire entre deux vecteurs $\vec{a}$ et $\vec{b}$

$$\vec{a}.\vec{b} = a_x * b_x + a_y * b_y + a_z * b_z$$

— **Cross :** Produit vectoriel entre deux vecteurs

$$\vec{a} \times \vec{b} = \begin{pmatrix} a_y * b_z - a_z * b_y \\ a_z * b_x - a_x * b_z \\ a_x * b_y - a_y * b_x \end{pmatrix}$$

— **Magnitude :** Retourne la norme du vecteur

$$\|\vec{v}\| = \sqrt{v_x^2 + v_y^2 + v_z^2}$$

— **Normalized :** Retourne le vecteur normalisé

$$\vec{n} = \begin{pmatrix} v_x / \|\vec{v}\| \\ v_y / \|\vec{v}\| \\ v_z / \|\vec{v}\| \end{pmatrix}$$

Nous vous demandons aussi d'implémenter les surcharges d'opérateurs suivantes :

```
1  //Addition between 2 vectors
2   public static Vector3 operator +(Vector3 a, Vector3 b);
3
4  //Scale a vector, when the scalar is on the right
5  public static Vector3 operator *(Vector3 a, float b);
6
7  //Scale a vector, when the scalar is on the left
8  public static Vector3 operator *(float a, Vector3 b);
9
10 //A shortcut for cross product
11 public static Vector3 operator *(Vector3 a, Vector3 b);
12
13 //Returns a - b
14 public static Vector3 operator -(Vector3 a, Vector3 b);
15
16 //Returns -a
17 public static Vector3 operator -(Vector3 a);
```

Attention !

Chaque fonction qui renvoie un vecteur doit retourner un **nouveau** vecteur et ne doit pas modifier ses opérandes.

Les fonctions **Max** et **Up** sont utilisées dans les fonctions de distances, vous ne devez donc pas les modifier.

3.2 Camera

Vous allez maintenant réellement commencer à travailler sur votre raymarcher, en implémentant la caméra. En effet, c'est grâce à la caméra que vous pourrez récupérer les rayons que vous allez parcourir et que vous pourrez les lier à des pixels de l'image.

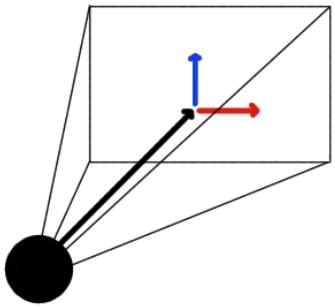
Attention !

Cette classe a un rôle très important, qui a un impact sur la suite du TP. Nous vous conseillons **très fortement** de la tester et de vous assurer qu'elle fonctionne correctement et que les formules mathématiques sont correctes avant de passer à la suite afin de vous faciliter le débug sur les prochaines étapes.

3.2.1 Constructeur

```
1  public Camera(Vector3 origin, Vector3 forward,  float fov);
```

Vous devez commencer par le constructeur, qui est appelé par le constructeur de la scène et qui vous permet de configurer la position, la direction et le FOV (champ de vision) de votre caméra. Le constructeur doit aussi calculer les vecteurs **up** et **right**, qui vont définir le plan représentant l'image dans l'espace en 3D.



Le vecteur noir épais est le vecteur forward, le vecteur bleu est le vecteur up et le vecteur rouge est le vecteur right.

Le vecteur **right** est simple à calculer, il suffit d'avoir un vecteur horizontal normal au vecteur **forward**. Cela suppose que la caméra ne peut pas subir de rotation autour de son axe **forward**.

$$\vec{right} = \begin{pmatrix} -forward_z \\ 0 \\ forward_x \end{pmatrix}$$

Le vecteur up nécessite de faire un produit vectoriel entre le vecteur **right**, que vous venez de calculer, et le vecteur **forward** de la caméra.

$$\vec{up} = \vec{right} \times \vec{forward}$$

Attention !

Le produit vectoriel n'est pas commutatif, ce qui signifie que $\vec{a} \times \vec{b} \neq \vec{b} \times \vec{a}$. Assurez vous donc que votre calcul du vecteur **up** est correct.

Attention !

Tous les vecteurs qui servent à décrire la caméra (**forward**, **right** et **up**) doivent être normalisés.

Pour vous aider

Un constructeur par défaut de la classe **Camera** est fourni dans la tarball, vous pouvez l'utiliser pour tester vos fonctions en ayant la certitude que la caméra est correctement construite.

3.2.2 ComputeFOV

```
1  public void ComputeFOV(float fov);
```

Cette fonction dois calculer la taille prise par l'image sur le plan qui la représente dans l'espace. Ce calcul doit se faire en fonction du FOV donné en paramètre. Comme le FOV est le même en hauteur et en largeur, le plan de l'image sera nécessairement carré.

$$width = height = |\tan(fov/2)| * 2$$

Attention !

La fonction **Math.tan** prend une valeur en radians. La valeur du champ de vision est donnée en degrés dans le fichier de la scène, vous devez donc penser à la convertir avant de faire vos calculs.

3.2.3 SetScreenData

```
1  public void SetScreenData(int width, int height);
```

Cette fonction prend en paramètres la largeur et la hauteur de l'image en pixels. Elle doit modifier la taille verticale du plan de l'image pour que le plan de l'image ait le même rapport *height*/*width* que l'image elle même. Ensuite, vous devrez calculer la taille d'un pixel sur le plan de l'image (si on a un plan de 10 * 10 et que l'image fait 10 * 10 pixels, chaque pixel fera 1 * 1). Les variables pour la taille d'un pixel sont **stepx** et **stepy**.

La fonction doit aussi calculer l'origine de l'image dans l'espace. Ce point correspond au coin haut-gauche de l'image, et le centre de l'image doit se trouver au bout du vecteur **forward**. Vous devrez calculer la position de ce point grâce à la position de la caméra, aux vecteurs **forward**, **right** et **up** et à la taille du plan que vous avez calculé.

$$\vec{screenOrigin} = \vec{origin} + \vec{forward} + \vec{up} * \mathbf{sizeY}/2 - \vec{right} * \mathbf{sizeX}/2$$

3.2.4 GetRay

```
1  public Vector3 GetRay(int x, int y);
```

Vous êtes maintenant prêts à écrire la fonction qui construit un rayon partant de l'origine de la caméra et passant par un pixel précis à partir des coordonnées de ce pixel sur l'image. Cette fonction prend les coordonées du pixel et doit renvoyer un vecteur **non normalisé** qui part de l'origine de la caméra et qui s'arrête sur ce pixel dans le plan de l'image.

Indication

Cette fonction est simple, elle est faisable en une ligne assez facilement. N'hésitez pas à faire un schéma pour vous aider.

3.3 Scene

Pour la classe **scene** nous vous fournissons le constructeur et la fonction **FloatEq**, qui permet de tester l'égalité entre deux flottants. C'est dans cette classe que se trouvera votre algorithme de raymarching. La classe **SDF** est la classe utilisée pour représenter les fonctions de distance.

3.3.1 Raymarch

```
1  public Vector3 RayMarch(SDF sdf,
2                          Vector3 ray,
3                          Vector3 origin,
4                          out int nbSteps,
5                          out float dist);
```

Cette fonction doit avancer le long du rayon **ray**, en partant de l'origine **origin**. Elle doit retourner le dernier point testé, mettre le nombre d'étapes prises sur le rayon dans **nbSteps** et la dernière valeur retournée par **sdf** dans **dist**. L'algorithme utilisé dans cette fonction est très simple :

```
current = origin
dist = sdf.evaluate()
while nbSteps < maxStep and not in an object:
    set current to the next position on the ray
    dist = sdf.evaluate()
    increment nbSteps
```

Conseil

Il est fortement conseillé de se référer au cours pour savoir quelle condition d'arrêt utiliser pour cet algorithme.

3.3.2 RenderScene

```
1  public void RenderScene(string fileName, int w, int h);
```

C'est ici que votre raymarcher va prendre vie ! Cette fonction va envoyer un rayon pour chaque pixel dans l'image, déterminer si il y a eu une intersection et régler la couleur de chaque pixel en fonction de ces informations. À la fin de cette fonction un nouveau fichier bitmap doit être créé avec le nom passé en paramètre **fileName**.

Par rapport aux bitmaps...

Afin de pouvoir créer une image et la sauvegarder sous le format bitmap, vous devrez utiliser la classe **Bitmap**. Pensez à ajouter **System.Drawing** dans les références du projet s'il ne trouve pas la classe **Bitmap** (même si la tarball devrait être correctement configurée, on n'est jamais à l'abri d'un problème).

L'algorithme pour cette fonction est le suivant :

```
Bitmap image = new Bitmap(...)
for each pixel:
    get a ray from the camera
    set the ray origin at the camera's origin
    normalize the ray
    search for an intersection along the ray (use the Raymarch function)
    if an intersection is found: put a white pixel
    else: put a black pixel

save image into a file
```

Attention !

Si vous décidez d'implémenter le shader, vous devrez appeler le shader à la place de mettre un pixel blanc mais vous devrez continuer de mettre un pixel noir si aucun point d'intersection n'a été trouvé sur le rayon.

3.4 Shader

Si vous êtes arrivés là, vous devez avoir la capacité de créer des images, mais elles sont un peu ennuyeuses. En effet, il n'y a aucun relief, juste du blanc ou du noir, ce n'est pas très intéressant. C'est dans cette partie que votre raymarcher va devenir intéressant ! Ici, vous allez calculer les effets d'ombres et de lumières sur vos objets. Nous allons commencer par prendre en compte la lumière directionnelle qui se trouve dans la scène, puis nous ajouterons le calcul des ombres projetées par les différents objets sur la scène.

Pour cette partie la fonction **CalculateNormal** est fournie, elle permet de calculer le vecteur normal à la surface touchée par un rayon au point indiqué. Elle prend en paramètre la fonction de distance de la scène et le point à partir duquel la normale doit être calculée.

3.4.1 Shade

```
1  public static Color Shade(Vector3 lightDir, Vector3 point, SDF sdf, int steps);
```

Vous devrez implémenter la fonction shade, qui va calculer la couleur du pixel (plus ou moins sombre) en fonction de la direction de la lumière et de celle de la normale. Vous devrez aussi ajouter de la lumière ambiante. La lumière ambiante est simple à calculer. Pour la lumière directionnelle, vous avez besoin de la normale et de la direction de la source de lumière et faire un produit scalaire entre les deux pour connaître l'intensité de la lumière qui frappe l'objet au point d'intersection, avec **ambient** l'intensité de la lumière ambiante, et **maxIntensity** l'intensité maximale de la lumière directionelle :

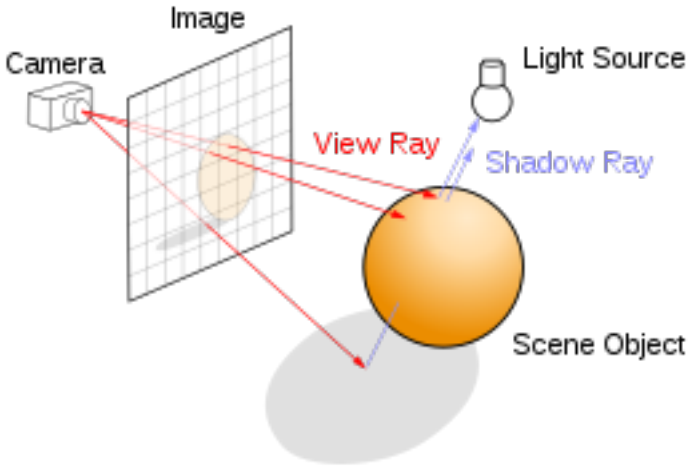
$$pixelColor = ambient + maxIntensity * (-\vec{normal}.\vec{lightRay})$$

Attention !

Si le résultat du produit scalaire est négatif, la source de lumière directionelle ne doit pas changer la couleur du pixel.

3.4.2 Shade V2

Maintenant que votre scène est éclairée, il ne nous manque plus que les ombres. L'ajout d'ombres est simple à faire. Il suffit de lancer un rayon depuis le point d'intersection en direction de la lumière et de vérifier si il rencontre un objet sur le chemin.



Pour calculer les ombres, on fait partir un rayon du point d'intersection vers la source de lumière et on regarde si il y a un objet entre les deux.

Cependant, il sera nécessaire de partir un peu avant la surface de l'objet pour éviter de commencer le rayon à l'intérieur de l'objet. Par conséquent, vous devrez vous éloigner légèrement de la surface (en utilisant la normale) pour placer le point d'origine du rayon. La direction du rayon est simple à calculer : c'est le vecteur qui va dans le sens inverse du rayon de la lumière. Si un point d'intersection a été trouvé, la lumière directionnelle ne doit pas modifier la couleur du pixel. Sinon, la lumière directionnelle doit agir normalement.

4 Bonus

Maintenant que vous avez un raymarcher capable de faire un rendu d'une scène en 3D et qui gère les ombres correctement, vous pouvez, si vous le souhaitez, aller plus loin en faisant des bonus (vous n'êtes absolument pas obligés d'en faire, la partie obligatoire s'arrête à Shade V2) :

De meilleures ombres : Nous pouvons vous suggérer de gérer des ombres douces, ce qui est simple à faire avec un raymarcher.
Vous pouvez vous aider de cet article :
`https://www.iquilezles.org/www/articles/rmshadows/rmshadows.htm`

Des modificateurs, d'autres opérateurs, et de nouvelles primitives : Vous aurez probablement remarqué qu'il n'y a aucun modificateur dans la tarball. Le parser est prêt à les supporter, il suffit d'ajouter les classes correspondantes et les ajouter dans la liste des objets de la NodeFactory.
Voici un lien utile avec une liste de primitives, d'opérateurs, et de modificateurs :
`https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm`

Autres idées : Vous pourriez aussi gérer la spécularité, l'occlusion ambiante, la rotation des objets, les réflections, ou toute autre chose qui vous semble intéressante à ajouter à votre raymarcher. Vous pouvez aussi tenter d'optimiser votre vitesse de rendu, mais sans dégrader la qualité de l'image.

5 Bibliographie

Parce que ce sujet a demandé une quantité importante de recherches et que nous souhaitons encourager les plus curieux à aller un peu plus loin, voici une petite bibliographie des différents articles, publications, et sites qui nous ont permis d'acquérir le savoir nécessaire à la réalisation de ce sujet :

- Fonctions de distance (Inigo Quillez) :
  `https://iquilezles.org/www/articles/distfunctions/distfunctions.htm`
- Ombres douces (Inigo Quillez) :
  `https://iquilezles.org/www/articles/rmshadows/rmshadows.htm`
- Un article d'introduction au raymarching (Jamie Wong) :
  `http://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/`
- Un article sur le rendu de fractales en 3D grâce au raymarching :
  `http://blog.hvidtfeldts.net/index.php/2011/08/distance-estimated-3d-fractals-iii-fold`
- Une discussion sur un forum spécialisé à propos d'opimisation d'un raymarcher :
  `http://www.pouet.net/topic.php?which=7920&page=69`
- Un rapport de thèse sur le raymarching : *Enhanced Sphere Tracing*, Benjamin Keinert, Henry Schäfer, Johann Korndörfer, Urs Ganse, Marc Stamminger :
  `http://erleuchtet.org/~cupe/permanent/enhanced_sphere_tracing.pdf`
- Encore une discussion sur l'optimisation :
  `http://www.fractalforums.com/mandelbulb-implementation/major-raymarching-optimization/`
- Une autre discussion sur des améliorations pour le raymarching :
  `http://www.fractalforums.com/fragmentarium/fast-fake-montecarlo-for-raymarching-(not-/`
- Un autre article d'introduction aux techniques de raymarching :
  `http://9bitscience.blogspot.com/2013/07/raymarching-distance-fields_14.html`
- Un autre rapport de thèse : *GPU Ray Marching of DistanceFields*, Lukasz Jaroslaw Tomczak :
  `http://www2.compute.dtu.dk/pubdb/views/edoc_download.php/6392/pdf/imm6392.pdf`
- Un autre article d'introduction au raymarching (Michael Walczyk) :
  `http://www.michaelwalczyk.com/blog/2017/5/25/ray-marching`

I see no point in coding if it can't be beautiful.