TP C# 15 : Raytracer

1 Règles de rendu

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

```
– rendu-login_x.zip
        |– rendu-login_x/
                |– AUTHORS
                |– README
                |– Raytracer/
                        |– Raytracer.sln
                        |– Raytracer/
                                |– Tout sauf bin/ et obj/
```

Bien entendu, vous devez remplacer *login_x* par votre propre login. N'oubliez pas de vérifier les points suivants avant de rendre :

- Le fichier AUTHORS doit être au format habituel : ** login_x$* où le caractère '$' représente un retour à la ligne.
- Le fichier README doit contenir les difficultés que vous avez rencontré sur ce TP, et indiquera les bonus que vous avez réalisé.
- Pas de dossiers bin ou obj dans le projet.
- **Le code doit compiler !**

2 Avant de commencer

Ceci est votre dernier TP de l'année. Pour cette occasion, vos charmants assistants ont préparé pour vous un sujet spécial qui contient tout ce que vous avez appris cette année. Ce TP sera sur 2 semaines avec un rendu intermédiaire à la fin de la première semaine pour voir votre avancée. Ce rendu sera une partie de votre note finale, **ne le sous-estimez pas !**

À partir de maintenant, vous devriez vraiment être à l'aise avec le code, particulièrement avec le C#. En effet, votre projet devrait maintenant être un peu développé et vous avez eu le temps d'apprendre beaucoup de choses sympatiques par vous même. Ce TP ne sera pas aussi dur qu'il peut sembler au premier abord, mais cela ne sera pas trivial non plus. Cependant, vous aimerez certainement ce que vous ferez et vous apprendrez aussi beaucoup de nouvelles choses intelligentes - nous sommes sûrs que vous savez déjà beaucoup de choses intelligentes (Ok, peut-être pas ...).

Comme ce sujet est un peu plus dur et plus long que les précédents, pour vous aider, vos assistants vous donneront un template avec quelques parties déjà codées : architecture d'heritage, prototypes, etc. Comprenez-le, utilisez-le et ne le modifiez pas. **Bon courage, et amusez vous bien !**

3 Courses

Dans cette premiere partie, nous allons vous expliquer ce qu'est un raytracer, ce qu'il fait et comment il le fait. Ne soyez pas étonné d'être perdu à la première lecture, ceci est totalement normal. Lisez-le de nouveau, à maintes reprises jusqu'à ce que vous compreniez avant de commencer à coder. En réalité, vous devriez lire ce TP comme ceci :

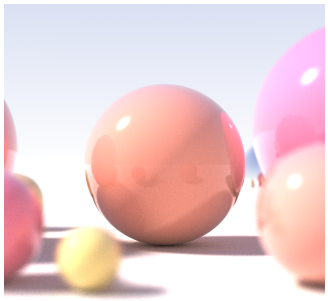
```
1 while (!i_understand_everything)
2 {
3         read_the_paper();
4 }
```

3.1 Raytraquoi ?

Pour ceux qui ne savent pas ce qu'est un raytracer, c'est un programme utilisant un algorithme de raytracing (surprenant hein !). Cet algorithme est utilisé pour produire des images graphiques en envoyant des "rayons" sur chaque pixel composant une image vide pour le remplir de la couleur appropriée. Il est généralement basé sur un système de coordonnées 3D et utilisé pour simuler des propriétés physiques complexes comme la réflexion, la réfraction ou la dispersion. Voici un exemple de ce que vous pouvez obtenir avec un raytracer :

Le raytracing peut créer des images réalistes (de Wikipédia)

Bien sûr, nous ne vous demanderons pas de créer un raytracer capable de produire des images comme celle-ci, c'est trop pour vous, particulièrement pour dans deux semaines. Cependant, si vous finissez entièrement le sujet, vous pourrez "raytracer" (comme disent les jeunes) quelques trucs comme ceci :



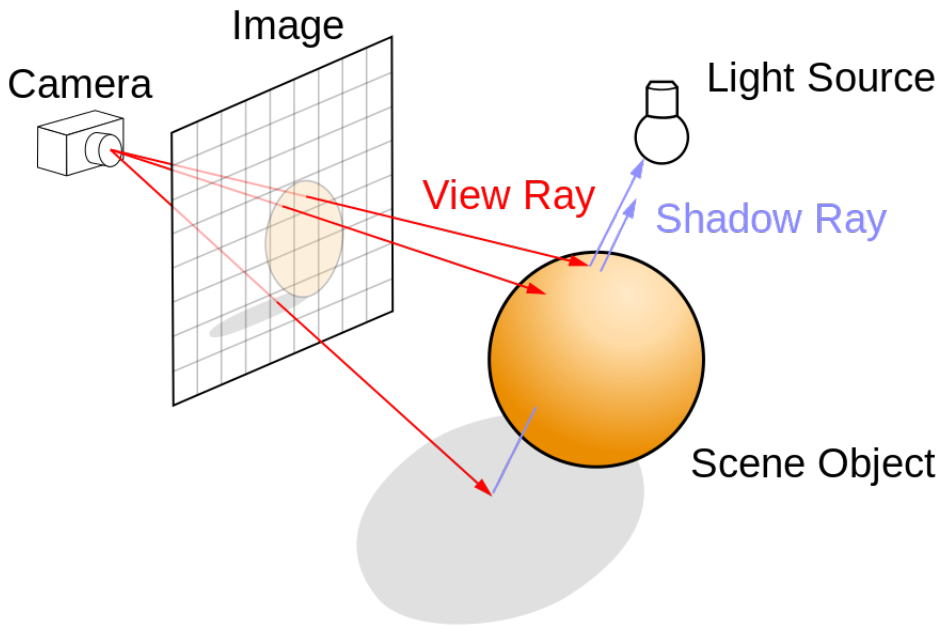
"Oh ! Quelle belle image !" (de Wikipedia)

3.2 Principle

L'algorithme de coloration de base marche comme suit :

— Boucler sur chaque pixel de l'image de destination.
— Pour chaque pixel, lancer un rayon qui va de l'oeil vers le pixel dans l'espace 3D.
— Récuperer l'intersection avec le modèle/objet le plus proche de la camera.
— Mettre la couleur du pixel à celle de l'objet.

Si cela vous semble un peu confus, ne vous inquiétez pas et jetez un coup d'oeil à ce dessin qui décrit bien la situation :



Modelisation de l'algorithme de raytracing (de Wikipedia)

Comme vous pouvez voir, la caméra a une position 3D dans le monde, le plan sur lequel tous les détails sont projetés. Ce plan est représenté par un tableau de taille *largeur * hauteur* dans la mémoire. En fait, quand vous jouez a des jeux vidéo d'une certaine qualité graphique (on ne parle pas de Minecraft), vous jouez dans un monde 3D, mais ce que vous pouvez voir sur votre écran est en réalité la projection du monde 3D sur votre écran 2D (votre écran n'est rien de plus qu'un plan 2D dessinant des choses en 2D). Toutes les entités à l'intérieur de notre monde 3D seront représentées par des informations, comme une position, un rayon (pour une sphère) ou 4 quatre points (pour un plan). Pour résumer, une scène 3D est juste un ensemble de données (des points, etc ...), sur lequel nous appliquons des formules de maths (la détection d'intersection, etc ...).

Comme vous le voyez, notre algorithme de base ne prend pas en compte de lumières dans la scène. C'est parfaitement normal, nous essayons de vous amener chaque concept l'un après l'autre. Grâce à ce TP, vous allez apprendre à mettre en œuvre certaines de ces fonctionnalités plutôt sympathiques.

3.3 Des rayons, des rayons partout !

Comme son nom indique, un raytracer utilise des rayons virtuels pour produire des images. Chaque rayon peut être représenté par un vecteur 3D, partant de la position de la caméra, dirigé vers le pixel actuel de l'image. Ensuite, chaque rayon est prolongé dans la scène 3D pour voir si celui-ci rentre en intersection avec une ou plusieurs formes.

Pour l'instant, nous conviendrons qu'un rayon est representé par l'équation :

O + t * D

Où O est le point représentant l'origine du rayon - c'est-à-dire la position de la caméra -, t la distance entre l'origine du rayon et le point d'intersection et D le vecteur directionnel du rayon - **ce vecteur est normalisé !**

3.4 Formes et intersections

Pour faire apparaitre plusieurs formes dans votre scène, vous devrez vérifier l'intersection entre un rayon et chaque objet de la scene. Chaque intersection potentielle sera représentée par une équation polynômiale. Le degré de l'équation dépend du nombre d'intersection(s) possible avec l'objet. Les solutions de l'équation représentent la distance entre l'appareil photo et les points d'intersection. Donc, si la solution est négative, cela signifie que le point d'intersection est derrière l'appareil photo et ainsi, vous ne devez pas le prendre en compte. S'il n'y a aucune solution, cela signifie que le rayon ne croise pas d'objet dans la scène.

Pour votre raytracer, nous ne vous demanderons pas de produire des objets complexes. Vous devrez pouvoir afficher quelques formes géométriques comme :

- **Plan** : seulement une intersection possible.
- **Face d'un cube** : seulement une intersection possible.
- **Face d'une pyramide** : seulement une intersection possible.
- **Sphère** : deux intersections possibles.
- **Cube** : deux intersections possibles.
- **Pyramide** : deux intersections possibles.

Selon la représentation mathématique d'un rayon définie auparavant, voici les équations que vous devrez résoudre pour chaque forme.

Pour le plan, où celui-ci est représenté par l'équation mathématique $ax+by+cz+d = 0$ et $n = (a, b, c)$, sa normale :

t = - (n . O + d) / (n . D)

Pour la sphere, où C est le centre de la sphere et R son rayon :

D^2 . t^2 + 2 . D . (O - C) . t + $\|O - C\|^2$ - R^2 = 0

Pour la face d'une pyramide - i.e. un triangle -, où P est l'intersection entre le rayon et le plan contenant le triangle et A, B, C les points constituants le triangle :

P = A + r.|B - A| + s.|C - A|
P dans le triangle si 0 <= r <= 1 et 0 <= s <= 1 et 0 <= r + s <= 1

Pour la face d'un cube - i.e. un carré -, c'est presque la même chose que pour le triangle :

P = A + r.|B - A| + s.|C - A|
P dans le carré si 0 <= r <= 1 et 0 <= s <= 1

Finalement, le cube et la pyramide sont composés de carrés et de triangles. Donc, il n'y a aucune formule pour ces formes. Vous devez juste vérifier si le rayon croise une des sous-formes composant votre cube/pyramide.

Pour résumer, ce que vous devrez faire est trouver le rayon, pour chaque pixel de l'image. Si vous voulez produire une image *500 * 200*, vous devez calculer l'intersection pour *500 * 200 = 100 000* rayons.

3.5 Et la lumière fut

Dans le monde réel, il existe beaucoup de différents types de lumiere. Les chercheurs ont constaté qu'il était facile de modéliser plusieurs d'entre elles, en se basant uniquement sur certains critères :

- **La lumière ambiante** : cette lumière n'a pas de position, ni de direction. C'est juste une lumière pour éviter que les objets soient complètement noirs dans une obscurité totale (quand il n'y a aucune lumière dans la scène).
- **La lumière directionnelle** : cette lumière peut être assimilée au soleil. Elle a une direction, mais n'a pas de position. Si elle est proche d'un objet ou loin loin loin (loin), ça ne change rien, seulement la direction de la lumière est prise en compte.
- **La lumière ponctuelle** : vous pouvez la comparer à une ampoule. Elle émet dans toutes les directions, mais cette fois, la distance à l'objet est importante et plus loin est l'objet, plus sombre celui-ci apparaîtra.
- **Et beaucoup d'autres** : il y a d'autres types de lumières, comme la lumière de type "spot" (la même utilisée par Beyoncé pour centrer l'attention sur elle), la lumière de zone, émise par la fenêtre d'une maison, etc...

Calcul de la lumière diffuse à un point donné :

diffuse = shapeDiffuse * lightDiffuse
newDiffuse = diffuse * max(- lightDirection x surfaceNormal, 0)

Calcul de la lumière spéculaire à un point donné :

specularFactor = pow(max(dot(viewDirection, reflectedDirection), 0), materialShininess) ; specular = lightColor * specularFactor * material.specular ;

Calcul de l'influence totale de la lumière à un point donné :

newDiffuse + ambient + specular

Vous allez bientôt découvrir que certaine lumière ont de l'atténuation, calculable comme ceci :

attenuation = 1 / (0.5 + 0.09 * distance + 0.032 * distance * distance)
avec 'distance', la distance entre l'intersection et la position de la lumière

Vous devrez mettre en œuvre plusieurs de ces lumières. Elles seront plus détaillées plus tard dans ce sujet.

3.6 Résumé

Pour résumer l'algorithme complet, vous devez d'abord créer une boucle envoyant tous les rayons nécessaires. Pour chaque rayon, vous devez vérifier si une intersection se produit ou ou non. Si c'est le cas, appliquer la résultante de(des) lumière(s) en bouclant sur chacune d'elle afin de calculer la nouvelle couleur du pixel.

4 Exercices obligatoires - première semaine

Ne soyez pas effrayés (quoique), vous n'allez pas être livré à vous même, nous allons vous guider tout au long du TP. N'oubliez pas, les **ACDC's** sont là pour vous aider, posez leur toutes vos questions, il ne faut surtout pas hésiter même si celles-ci vous semble absurdes...

4.1 Vecteurs

Commençons le raytracer ! Avant de raytracer quoi que ce soit, vous devrez écrire quelques classes qui seront utiles pour le reste du sujet, en commençant par la classe `Vector3`. Un `Vector3` est représenté par 3 `double` : `x`, `y` et `z` (ses coordonnées).

Constructeurs

Écrivez les deux constructeurs de la classe `Vector3`, l'un ne prenant aucun argument et l'autre 3. Celui qui n'a pas d'argument initialise simplement le `Vector3` à (0, 0, 0). L'autre initialise le `Vector3` à (x, y, z).

```
1 public Vector3();
2 public Vector3(double x, double y, double z);
```

Méthodes

Écrivez les méthodes `Vector3` suivantes, respectant les prototypes déjà codés.

```
1 /* Retourne la norme du vecteur */
2 public double norm();
3
4 /* Normalise le vecteur */
5 public void normalize();
6
7 /* Retourne la distance entre les points 'p1' et 'p2' */
8 public static double distance(Vector3 p1, Vector3 p2);
```

Operateurs

Pour rendre la classe du `Vector3` plus lisible, nous allons surcharger quelques opérateurs. Voici les opérateurs que vous devez surcharger. Si vous ne savez pas ou ne vous rappelez pas des opérations sur les vecteurs, d'abord, c'est une honte et ensuite, **Google est votre ami**.

```
1 /* Retourne le produit scalaire des vecteurs 'v1' et 'v2' */
2 public static double operator |(Vector3 v1, Vector3 v2)
3
```

```
/* Returne la somme des vecteurs 'v1' et 'v2' */
public static Vector3 operator +(Vector3 v1, Vector3 v2)

/* Returne la soustraction des vecteurs 'v1' et 'v2' */
public static Vector3 operator -(Vector3 v1, Vector3 v2)

/* Retourne le produit vectoriel entre 'v1' et 'v2' */
public static Vector3 operator *(Vector3 v1, Vector3 v2)

/* Multiplie le vecteur 'v' par la constance 'd' */
public static Vector3 operator *(Vector3 v, double d)
public static Vector3 operator *(double d, Vector3 v)
```

Getters & Setters

Implémentez les getters et setters suivants pour les attributs `Vector3`.

```
/* Getters et setters */
public double X;
public double Y;
public double Z;
```

4.2 Couleurs

Pour que la scène apparaisse, nous allons avoir besoin d'une classe représentant la couleur de chaque pixel de l'image. En réalité, une classe existe déjà, dans le namespace `System.Drawing`, nommé `Color`. Ici, nous allons effectuer des opérations entre couleurs, donc chaque composante doit être un **flotant dont la valeur varie entre 0 et 1**. Donc, nous allons créer notre propre classe appelée `NormalizedColor`.

Constructeurs

Écrivez les constructeurs suivant pour la clasee `NormalizedColor`.

```
/* Instancie une nouvelle couleur de valeurs (0, 0, 0) */
public NormalizedColor();

/* Instancie une nouvelle couleur de valeurs (r, g, b) */
public NormalizedColor(double r, double g, double b);

/* Instancie une nouvelle couleur à partir des valeurs de 'c' */
public NormalizedColor(Color c);
```

Méthode

Écrivez la méthode suivante de la classe `NormalizedColor`, nous permettant de convertir une `NormalizedColor` en une `Color` du namespace `System.Drawing`. N'oubliez pas, les composantes de nos couleurs se situent entre 0.0 et 1.0, alors que les couleurs de `System.Drawing` possèdent des entiers entre 0 et 255.

```
public Color to_color();
```

Opérateurs

Comme pour la classe `Vector3`, `NormalizedColor` il nous est possible d'effectuer des opérations sur les couleurs. Voici celles que l'on vous demande :

```
/* Multiplie chaque composante de 'c' par 'd', et retourne le resultat */
public static NormalizedColor operator *(NormalizedColor c, double d);

/* Multiplie chaque composante de 'c1' par celles de 'c2', et retourne le resultat */
public static NormalizedColor operator *(NormalizedColor c1,
                                         NormalizedColor c2);

/* Ajoute chaque composante de 'c1' à celles de 'c2', et retourne le resultat */
public static NormalizedColor operator +(NormalizedColor c1,
                                         NormalizedColor c2);
```

Getters

Écrivez les getters suivants pour la classe `NormalizedColor`.

```
/* Getters only */
public double R;
public double G
public double B;
```

4.3 Polymomes

Comme expliqué dans la partie précédente du sujet, vous devrez résoudre des équations polinômiales. Pour le faire correctement, nous allons créer une classe `Polynôme`. Cette classe sera représentée par un tableau de coefficients et le degré du polynôme.

Constructeur

Écrivez le constructeur de la classe `Polynomial` prenant en paramètres un tableau de coefficients et le degré du polynôme.

```
public Polynomial(double[] coefs, uint n);
```

Méthode

Écrivez la fonction suivante, en respectant le prototype, permettant de résoudre une équation quadratique. Cette méthode prend en argument un tableau dans lequel sont enregistrées les solutions - s'il y en a - de l'équation. Elle retourne le nombre de solutions trouvées.

```
public uint resolve_quadratic(ref double[] squares);
```

4.4 Camera

La classe `Camera` est très simple à implémenter, il n'y a aucune méthode à écrire. Cette classe sera definie par une position dans l'espace et par deux `Vector3`, u et v, représentant l'orientation de la caméra dans notre scène 3D.

Constructeur

Écrivez le constructor de la classe `Camera`, prenant en paramètres sa position et ses deux `Vector3` d'orientations, u and v.

```
public Camera(Vector3 pos, Vector3 u, Vector3 v);
```

Getters

Écrivez les getters suivants pour la classe `Camera` :

```
/* Getters seulement */
public Vector3 Pos;
public Vector3 U;
public Vector3 V;
```

4.5 Screen

Maintenant, écrivons la classe `Screen`. Celle-ci représente l'image que l'on veut raytracer. Elle a comme attributs les dimensions de l'image et le point central de l'écran.

Constructeur

Écrivez le constructeur de la classe `Screen`, prenant en paramètre les dimensions de l'image.

```
public Screen(int width, int height);
```

Méthode

Écrivez la methode suivante, calculant le centre de l'écran (représenté par un objet `Screen`) C selon la `Camera`, en utilisant ce système mathématique :

$$\begin{gathered} \text{C - Lw} = \text{camera\_pos} \\ \tan(\text{fov} \ / \ 2) = (\text{screen\_width} \ / \ 2) \ / \ \text{L} \end{gathered}$$

où C est le centre de l'écran, L la distance entre la camera et le point central, *camera_pos* la position de la caméra, fov le 'field of view' - **qui sera toujours de 45 degrés** - et w le produit vectoriel des vecteurs u et v de la caméra.

```
public void set_center(Camera cam);
```

Getters & Setters

Écrivez les getters et setters suivants pour la classe `Screen`.

```
/* Getters et setters */
public int Width;
public int Height;

/* Getter seulement */
public Vector3 Center;
```

4.6 Rayons

Même si des rayons virtuels pourraient être simplement définies par un vecteur, nous avons décidé de créer une classe spécialement pour eux (la chance !) pour une raison de clarté et d'organisation. Donc, voici ce que vous devez faire pour la classe `Ray`. Cette classe a pour attributs deux `Vector3`, le point d'origine du rayon et sa direction.

Constructeur

Écrivez le constructeur la classe `Ray`, prenant comme arguments son origine et sa direction.

```
public Ray(Vector3 origin, Vector3 dir);
```

Methode

Pour la classe `Ray`, la seule méthode nécessaire est celle qui nous permet d'obtenir un rayon (objet de la classe `Ray`) selon le pixel actuel de l'image. Pour calculer ce rayon, nous aurons besoin de la `Camera` et du `Screen`. Voici comment nous allons calculer le `Ray`) - **n'oubliez pas de normaliser D** - :

$$O = \text{Cam.Pos}$$
$$D = \text{Screen.Center} + \text{Cam.U} * dx + \text{Cam.V} * dy - \text{Cam.Pos}$$
$$dx = x - \text{Screen.Width} / 2 \text{ and } dy = y - \text{Screen.Height} / 2$$

Ensuite, c'est au tour du prototype de cette méthode, x et y étant les coordonnées du pixel actuel sur l'image.

```
public static Ray get_ray(int x, int y, Screen screen, Camera cam);
```

Getters

Écrivez les getters suivants pour la classe `Ray` :

```
/* Getters Seulememt */
public Vector3 Origin;
public Vector3 Dir;
```

4.7 Material

Pour définir l'aspect de nos formes, nous avons décidé de créer une classe `Material`. Cette classe sera presente dans la classe `Shape`, que l'on vous expliquera plus tard. Elle a pour attributs la couleur diffuse de la `Shape`, sa couleur spéculaire et sa brillance. Comme la `Camera`, la classe `Material` est vraiment facile a coder puisqu'elle contient seulement un constructeur et des getters.

Constructeur

Écrivez le constructeur de la classe `Material`, prenant comme arguments la couleur diffuse, la couleur spéculaire et la brillance de la `Shape`.

```
1 public Material(NormalizedColor diff, NormalizedColor spec, float shin);
```

Getters & Setters

Écrivez les getters et setters suivants de la classe `Material` :

```
1 /* Getters et setters */
2 public NormalizedColor Diffuse;
3 public NormalizedColor Specular;
4 public float Shininess;
```

4.8 Forme

Comme il a été dit au début de ce sujet, nous allons mettre en oeuvre différents objets. Tous hériteront de `Shape`. Le seul attribut que les objets auront en commun est leur `Material`. Donc, ce sera le seul attribut de la classe `Shape`.

Constructeur

Écrivez the constructeur de la classe `Shape`, prenant comme argument son `Material`.

```
1 public Shape(Material mat);
```

Methodes

Il n'y a rien à implementer ici. En effet, les deux méthodes que nous allons voir sont abstraites, leur implémentation sera donc dans les sous-classes.

La première méthode que nous allons voir est celle qui nous permet de vérifier si un `Ray` croise la `Shape` courante. Bien sûr, chaque `Shape` a son propre algorithme. Référez-vous à la partie de cours du sujet. Le deuxième retourne la normale à un point surfacique de la `Shape`. Comme il a été dit plusieurs fois, le `Vector3` doit être normalisé (nor...ma...li...sé ! Tous vos bugs vont venir d'ici de toutes façons).

```
1 public abstract Vector3 intersect(Ray ray);
2 public abstract Vector3 normal_at_point(Vector3 point);
```

Getter

Écrivez le getter suivant de la classe `Shape`.

```
/* Getter seulement */
public Material Mat;
```

4.9 Sphere

Nous sommes maintenant prêts à implementer les différentes formes, en commençant par la classe `Sphere`. Celle-ci sera définie par une position et un rayon.

Constructeur

Écrivez le constructeur de la classe `Sphere`, prenant en parametre son `Material`, sa position et son rayon.

```
public Sphere(Material mat, Vector3 position, float radius);
```

Methodes

Comme cela a été dit dans la partie précédente, la classe `Sphere` hérite de `Shape`. Donc, vous devez implémenter les méthodes `intersect` et `normal_at_point` en utilisant la partie cours et la classe `Polynomial`.

4.10 Light

Maintenant, il est temps d'ajouter de la lumière (et donc des variations de couleurs) à notre scène. Pour ce faire, nous allons utiliser la classe `Light` . Comme la classe `Shape`, `Light` est la classe dont `Directional` et `PointLight` héritent. Cette classe aura pour attribut une `NormalizedColor`, couleur diffuse de la lumière.

Constructeur

Écrivez le constructeur de la classe `Light`, prenant en paramètre sa couleur.

```
public Light(NormalizedColor c);
```

Méthodes

Comme pour la classe `Shape`, les deux méthodes de `Light` sont abstraites, leur implementation sera donc faite dans les sous-classes `Directional` et `PointLight`. Cependant, vous allez devoir implémenter `specular_get`, utilisée pour calculer le facteur spéculaire à un point d'intersection donné (mais ceci sera à faire pour la deuxième semaine).

La première méthode `apply_lighting` retourne la nouvelle couleur du pixel au point d'intersection trouvé entre le `Ray` courant et la `Shape`. Il prend comme argument le `Material` de la `Shape`. La couleur retourné prend en compte la lumière courante appliquée, ainsi que le `Material` de la forme. Il s'agit donc d'appliquer ici l'algorithme de composition des couleurs :

```
/* La formule pour obtenir la couleur du nouveau pixel */
(newDiffuse + specularColor + ambient);
```

Dans `apply_lighting`, vous devez donc calculer la diffuse et la spéculaire engendrée par la lumière, et de sommer le tout avec la couleur ambiante. Pour le moment, il n'est pas nécessaire de calculer la spéculaire (ceci sera fait à la deuxième semaine). Veillez simplement à utiliser la diffuse et l'ambiante.

Le deuxième, `get_direction`, retourne le vecteur directionnel - **normalisé** - entre la position de la lumière et le point d'intersection (pour les lumières sans position, il s'agit simplement de la direction actuelle de la lumière).

Getters

Écrivez le getter suivant de la classe `Light`.

```
1 /* Getter seulement */
2 public NormalizedColor Color;
```

4.11 Lumière Directionelle

Maintenant que nous avons notre classe `Light`, il est temps d'implémenter la classe `Directionnal`. Cette `Light` est définie par un `Vector3`, sa direction.

Constructeur

Écrivez le constructeur de la classe `Directional`, prenant en paramètre sa couleur et sa direction.

```
1 public Directional(NormalizedColor c, Vector3 dir);
```

Méthodes

Implémentez les deux methodes `apply_lighting` et `get_direction` en vous réferrant à la partie du cours sur les lumières.

Getters

Écrivez le getter de la classe `Directional` suivant.

```
1 /* Getter seulement */
2 public Vector3 Dir;
```

4.12 Algorithme de Ray Tracing

Pour remplir l'image, vous devez remplir le fichier `Raytracer.cs`, situé à la racine du projet. Vous devrez remplir la methode `render_image` qui applique l'algorithme.

Trouver la composante en z (non, ça n'est pas un titre de film)

Pour trouver la direction du rayon actuel, vous devrez trouver le vecteur de l'origine au pixel 3D et le normaliser. Vous pouvez facilement trouver les composantes x et y, en utilisant la largeur d'image et la hauteur, mais trouver le z est un peu délicat.

En utilisant seulement des maths de base, nous pouvons trouver cette composante z (la distance au plan le plus proche) grâce à l'angle - le champ de vision - qui est égal à `45°` pour nous, humains.
Nous savons trouver le `fov`, nous pouvons donc déduire la composante z :

$$z = \text{width} \ / \ \tan((45 * \pi) \ / \ 180))$$

Méthode

Maintenant que vous savez trouver la position 3D du pixel en fonction de la camera, vous devez compléter la fonction `render_image` qui retourne l'image finale.

```
/* Cette fonction prend la liste des formes, la liste des lumières et la
lumière ambiante */
public Bitmap render_image(Camera cam, Screen screen, List<Shape> nodes,
List<Light> lights, NormalizedColor ambient, ProgressBar pb)
```

Nous vous rappellons que cette fonction doit fonctionner de la maniere suivante :

— **1.** Boucler sur tous les pixels (attention aux bornes des deux boucles).
— **2.** Pour chaque pixel, boucler sur toutes les formes pour determinrer la couleur de la forme intersectée la plus proche de la caméra, s'il y en a une.
— **3.** Boucler sur toutes les lumieres de facon à appliquer l'eclairage de la couleur précedemment trouvée.

5 Exercices obligatoires - Deuxieme semaine

5.1 Plan

Commençons cette deuxieme semaine avec une forme simple : le plan. Un `Plane` sera défini par 4 nombres flotants a, b, c et d représentant l'equation planaire suivante $ax + by + cz + d = 0$.

Constructeur

Implémentez le construteur de `Plane`, qui prend 4 constantes en argument definissant son équation ainsi que son `Material`.

```
public Plane(Material mat, double a, double b, double c, double d)
```

Méthodes

Comme pour la `Sphere`, implémentez les deux méthodes `intersection` et `normal_at_point`, en suivant les indications données dans la partie correspondante du cours.

5.2 Face d'un cube

Cette forme est aussi simple qu'un carré. En fait, elle n'a aucun interêt lorsqu'elle est utilisée seule. Cependant, nous allons calculer un cube ou une pyramide plus tard. Cette forme sera composee de 3 points - le quatrième pourvant etre déterminé depuis les 3 autres, ainsi que d'un `Material`.

Constructeur

Écrivez le constructeur de `CubeFace`, prenant en arguments 3 `Vector3` et un `Material`.

```
1 public CubeFace(Material mat ,Vector3 point_a, Vector3 point_b, Vector3 point_c)
```

Méthodes

Implémentez les deux méthodes `intersection` et `normal_at_point` en vous referrant au cours.

5.3 Cube

De façons à créer notre `Cube`, nous allons avoir besoin de 6 `CubeFace`, definie depuis le centre du `Cube` et de sa taille. En effet, il est possible de trouver 6 `CubeFace` composant un cube seulement avec ces informations.

Constructeur

Écrivez le constructeur de `Cube`, prenant en arguments la position du centre du cube, sa taille et son `Material`.

```
1 public Cube(Material mat, Vector3 pos, float s)
```

Méthodes

Implémentez les deux methodes `intersection` et `normal_at_point` en vous réferrant au cours.

5.4 Face de Pyramide

Comme pour la `CubeFace`, la `PyramidFace` est simplement un triangle en 2D. Nous aurons besoins de créer une `Pyramid`. La `PyramidFace` est créée depuis ses 3 vertices.

Constructeur

Écrivez le constructeur de `Cube`, prenant en arguments la position de ses trois sommets, sa taille et son `Material`.

```
1 public PyramidFace(Material mat, Vector3 point_a, Vector3 point_b, Vector3 point_c)
```

Méthodes

Implémentez les deux methodes `intersection` et `normal_at_point` en vous referrant au cours.

5.5 Pyramide

Une `Pyramide` sera definie par sa base, une `CubeFace`, et ses 4 `PyramidFaces`. Celles ci seront calculees a partir de la position du centre de sa base, de la taille de la base et de la hauteur de la `Pyramid`.

Constructeur

Écrivez le constructeur de `Cube`, prenant en arguments la position de sa base, la taille de la base, sa hauteur et son `Material`.

```
1 public Pyramid(Material mat, Vector3 pos, float base_size, float height)
```

Méthodes

Implémentez les deux méthodes `intersection` et `normal_at_point` en vous referrant au cours.

5.6 Un point avant de continuer

Avant de continuer, vous devriez tester que vos lumières fonctionnement sur vos formes et que vos lumières peuvent être cumulés sur les objets présents dans votre scène. Si vous n'avez pas codé comme des gorets, il ne devrait pas y avoir de soucis.

Et maintenant... vous allez enfin pouvoir entrevoir vos formes avec... **DE LA SPECULAIRE !** Implémentez la fonction `get_specular` définie dans la classe `Light`, en utilisant la formule du cours.

```
1 // 'init_dir' représente la direction de la lumière
2 // 'point' représente le point d'intersection
3 // 'normal' représente la normal à la surface au point d'intersection
4 // 'shininess' le coefficient spéculaire associé au Material de la forme courante
5 public double specular_get(Ray ray, Vector3 init_dir, Vector3 point,
6                             Vector3 normal, float shininess);
```

N'oubliez pas de modifier les functions `apply_lightning` de la classe `Directional`. Maintenant, l'algorithme de coloration devrait prendre la lumière spéculaire en compte.

5.7 Lumiere d'attenuation

Le fichier `AttenuationLight.cs` est à moitié rempli et n'attend que vous pour être terminé ! Cette classe est utilisée comme une `Interface`, qui ne peut pas être instanciée, car une `AttenuationLight` n'existe pas dans le monde réel. Elle existe sous certaines formes (point light, spot light,...).

Juste pour la théorie, toutes les lumieres qui seront soumises à une atténuation auront 3 constantes. `Pourquoi utiliser 3 constantes et pas un simple facteur lineaire ?` Excellente question ! (ne mentez pas !, vous vous la posiez). En fait, ces 3 constantes permettent un résultat plus lisse/doux au regard voyez vous ? Et cela permet d'éviter un gros "trou" qui serait visible entre une surface eclairée et une surface sombre.

Méthodes

Complétez la fonction `attenuation_get` qui va calculer la distance entre la lumière et le point donné en paramètre en appliquant la formule d'atténuation donnée dans le cours.

5.8 Point Light

Comme expliqué au début de ce TP, une 'Point Light' émet dans toutes les directions avec une atténuation croissante en fonction de la distance. Complétez le fichier `PointLight.cs` de façons à mettre des lumières aussi jolies que celles de Unity !

Méthodes

Comme pour la classe `DirectionalLight`, vous devez implémenter la fonction `apply_lightning`. Son implémentation est très proche de ce que vous avez fait. Mais faites attention, rappellez vous que c'est une `AttenuationLight`, donc n'oubliez pas de prendre cette atténuation en compte dans votre calcul.

Vous aurez aussi à implémenter la fonction `get_direction` qui retourne le vector **normalise** entre la position de la lumière et le point d'intersection. Faites attention ici, nous demandons `position - point`, dans cet ordre la, qui est super mega mega mega mega mega mega mega mega mega mega mega important (mega important (genre, vraiment, (genre vraiment, vraiment)).

5.9 Spot Light

Un Spot emets un cercle 2D de lumière, atténuant la lumière quand l'intersection est proche des bords de la forme. Complétez le fichier `SpotLight.cs`, en utilisant la formule (contenant l'angle du cone, etc...)

Méthodes

Comme pour la lumière précédente, vous devez implémenter la fontion `apply_lightning`.

Vous devez également implémenter la fonction `get_direction` qui retourne le vecteur **normalise** entre la position de la lumière et le point d'intersection. Faites attention, nous demandons toujours `position - point`, dans cet ordre la, je me repète mais c'est important (genre vraiment, oui bon, j'arrête) ;)

5.10 Ombres

Vous devriez maintenant avoir un Raytracer rendant des images assez séduisantes. Cependant, nous le savons, ce n'est pas assez pour vous. Vous cherchez toujours une façon d'améliorer votre projet et il est temps de nous montrer que vous voulez avoir un raytracer aussi bon que celui présenté dans la section de cours.

Principe

Pour rendre quelques ombres vous pouvez utiliser algorithme simple comme suivant :

— **1.** Quand vous bouclez sur chaque lumière de la scène après avoir trouvé le point d'intersection, bouclez sur chaques formes - **de nouveau !** - et utilisez la fonction `get_direction` de la classe pour déduire le vecteur de direction entre la direction de la lumière et ce point d'intersection.

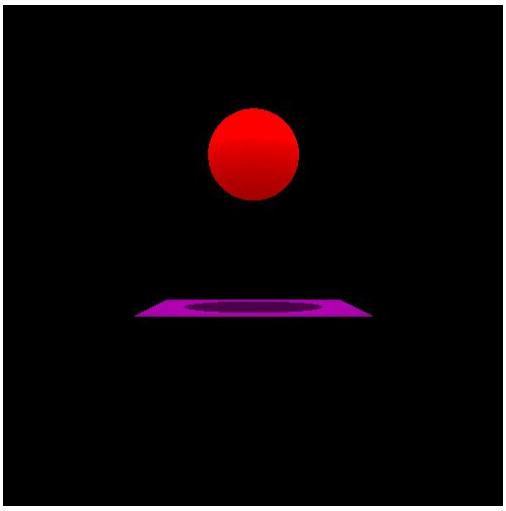
— **2.** En utilisant cette direction, créez un nouveau `Ray` et vérifiez l'intersection entre la forme actuelle (de la nouvele boucle bien sûr) et ce `Ray`.

— **3.** Si un point d'intersection est trouvé, simplement obscurcir le pixel en le multipliant par une petit constante (e.g : `0.45d`)

Méthodes

Complétez la methode `render_image` dans le fichier `Raytracer.cs` en modifiant son contenu de facon à appliquer l'algorithme d'ombre au code précedent.

Résultat attendu



Resultat utilisant le fichier `cube_sphere_point_light_shadows.xml`

6 Bonus - On commence le vrai boulot

Nous allons maintenant vous présenter quelques caractéristiques bonus que vous pouvez mettre en œuvre pour votre raytracer. **Vous devez finir la partie obligatoire avant de commencer la partie de bonus !** De plus, les caractéristiques suivantes ne sont triées dans aucun ordre, difficulté ou autre. Vous êtes, donc, libres de mettre en œuvre ceux de votre choix.

Remarquez que les bonus sont le début de la partie intéressante de ce projet. **Faites les !**. De plus, n'oubliez pas d'indiquer les bonus que vous faites dans votre README.

6.1 Anti-aliasing

Vous pouvez avoir remarqué que notre raytracer calcule des images contenant des formes courbées. Le problème est comment faire pour dessiner des images contenant de belles courbes sans avoir pour autant cet effet escalier ?

Grâce à la méthode d'aplanissement, vous pouvez obtenir des courbes plus lisses. Cette méthode, qui est vraiment coûteuse, consiste dans la division de chaques pixels dans des sous-pixels. Plus vous avez de sous-pixels, plus l'image sera clairs. Une fois que vous avez calculé la couleur de chaque sous-pixel, la couleur du pixel final est la moyenne de ses sous-pixels.

6.2 Multi-threads

L'algorithme de raytracing n'est certainement pas un algorithme optimisé, comme vous pouvez l'avoir remarqué. Ainsi, la seule façon d'améliorer la vitesse de votre "traceur de raie" est de le multi-threader.

Prenez garde, nous parlons de CPUs. Donc, la façon la plus efficace de multi-threader votre programme est de créer 8 threads, calculant en même temps la couleur des pixels.

6.3 Plus de formes

Pendant ce TP, vous avez dû créer plusieurs formes simples. Celles-là étaient tracées avec des équations de polynôme quadratiques. Cependant, il existe plus de formes qui peuvent être faites avec ce type d'équation : cône, cylindre, etc.

Si vous êtes vraiment motivés, vous pouvez aussi essayer de tracer des formes plus difficiles avec un degré plus haut d'équation. Le plus connu est le tore, tracé avec des équations de polynôme de degré 4.

6.4 Constructive Solid Geometry - C.S.G.

Une fois que vous avez mis en œuvre toutes les formes simples possibles - ou pas - vous pouvez pour vous demander : comment puis-je faire des formes plus compliquées ?

C.S.G permet de créer des formes compliquées a partir de formes simples. Ce processus utilise des operateurs de théorie des ensembles. En effet, une forme peut être une union, une différence ou une intersection de plusieurs autres formes.

Si vous êtes intéressés par ce bonus, référez-vous à la page de Wikipédia et faites quelques recherches. Ce concept est bien documenté sur Internet.

6.5 Texturing

Pour faire de plus belles images, vous pouvez utiliser des textures. Vous pouvez texturer vos formes en utilisant des images simples (en puissance de 2 généralement) et pourquoi pas implémenter un effet de Normal Mapping, afin de donner du relief à vos textures.

Le Texture Mapping vous permet d'appliquer une image représentant une texture sur une forme. Le Normal Mapping vous permet de créer des perturbations colorées sur des surfaces de forme, en utilisant des textures donnant des informations quand à l'orientation des normales d'une texture.

Encore une fois, Google est votre meilleur ami. La documentation et les formules sont faciles à trouver sur Internet.

6.6 Reflexion

Si vous voulez mettre en oeuvre un effet vraiment cool, il y a l'effet de miroir, aussi appelé la réflexion. Le principe est vraiment simple : une fois que vous obtenez le point d'intersection d'une forme, vous devez calculer le vecteur de réflexion, le suivre jusqu'à ce que vous croisiez une autre forme et obteniez la couleur à ce point d'intersection.

En effet, le principe est simple, mais quelques cas délicats doivent être traités. Par exemple, si vous avez 2 formes réfléxives dans votre scène, vous devez faire attention pour ne pas tomber dans une récursion infinie.

6.7 Et plein d'autres...

Les bonus précédents sont juste une petite partie de ce que vous pouvez faire. **impressionnez nous !** Il y a tant de choses que vous pouvez mettre en œuvre : parallélisme GPU avec CudaFy, un éditeur d'images en temps réel, un créateur de GIF au raytracer, etc.

Essayez d'avoir le plus impressionnant et complet des raytracer. Mais, n'oubliez pas, **deadline is coming**.