Synthèse d'images par lancer de rayon
Rapport

BENKORT Matthias, DOLOU Clément,
GABORIT Nicolas, HERVE Maxime
Groupe 23

12 juin 2013

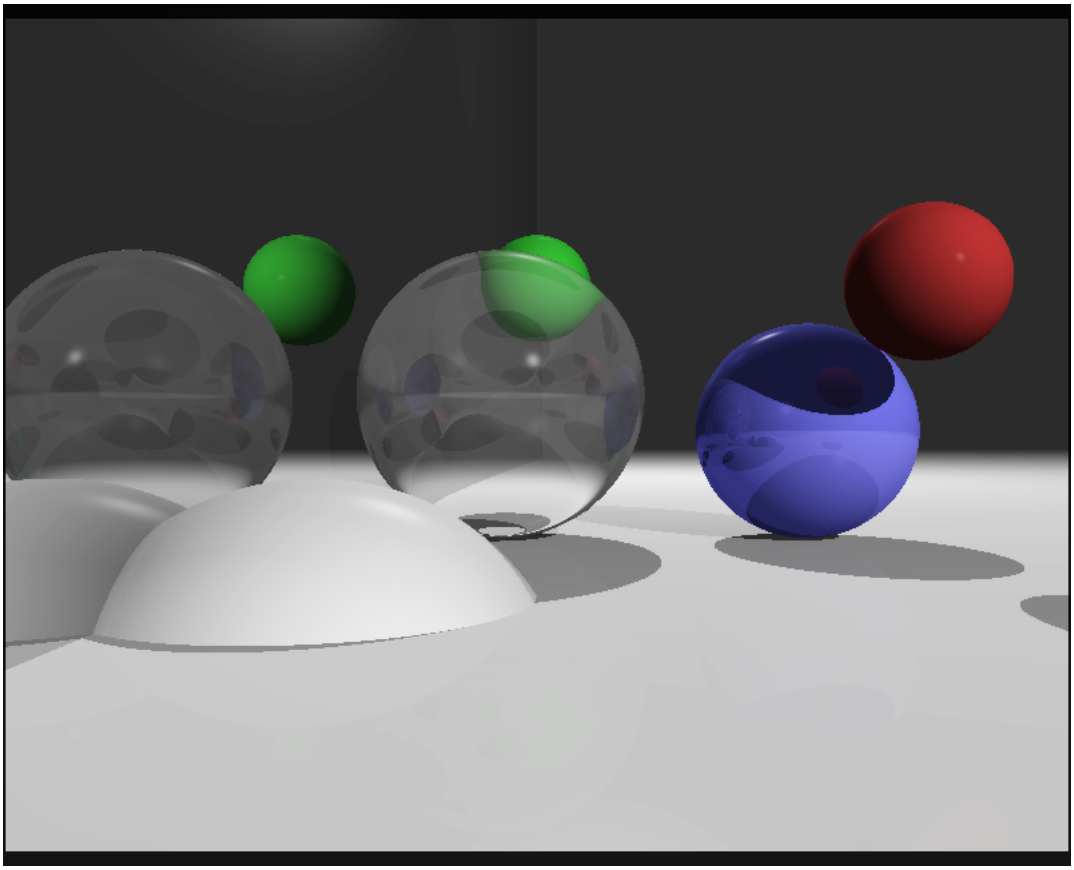

Table des matières

1 Analyse **3**
1.1 Modèle . . . 3
1.1.1 Disposition de la scène . . . 3
1.1.2 Observation de la scène . . . 4
1.2 Interface graphique . . . 5

2 Implémentation du modèle **6**
2.1 Utilitaires . . . 6
2.1.1 La classe Geometry . . . 6
2.1.2 Materiaux . . . 7
2.2 Disposition de la scène . . . 7
2.2.1 Les éléments . . . 7
2.2.2 Les sources de lumière . . . 13
2.2.3 Les points de vue . . . 13
2.3 Observation de la scène . . . 15

3 Implémentation de l'interface graphique **20**
3.1 La barre de menu . . . 20
3.2 Le corps de l'application . . . 21

4 Tests **21**
4.1 Tests . . . 21

Introduction

Le présent rapport est consacré à la présentation du travail effectué autour du projet de programmation orientée objet qui nous a été attribué. Le projet vise à développer une application Java de synthèse d'images à partir de scènes en trois dimensions.

L'application doit permettre de créer une telle scène contenant des objets et sources de lumières, puis de placer un point de vue et enfin de calculer l'image de la scène observée depuis ce point. Les objets modélisés sont des sphères, des cubes, des facettes et des plans infinis. L'utilisateur doit pouvoir exploiter toutes les possibilités de l'application par l'intermédiaire d'une interface graphique conviviale.

Le modèle de Blinn-Phong sera utilisé pour le calcul des images. C'est un algorithme de lancer de rayons éprouvé. Il est ainsi possible d'obtenir de manière simple un rendu réaliste de la scène en trois dimensions.

Dans un premier temps nous décrivons notre démarche d'analyse du problème. Nous détaillerons ensuite les fonctionnalités de l'application finale, en commençant par le modèle (le lanceur de rayons) et en finissant par l'interface graphique. Nous conclurons sur la manière dont nous avons organisé notre travail, les tests effectués et la documentation du programme.

1 Analyse

Dans cette section, nous abordons la phase analytique du projet dont le support est le diagramme UML de l'application. Notre application est pensée selon le patron de conception *Modèle/Vue/Contrôleur*. Ceci nous permet, lors de la phase d'analyse, de nous concentrer sur la modélisation du domaine métier correspondant ici au modèle de notre application. Nous nous intéresserons donc d'abord à la modélisation de la scène. Nous exposerons ensuite notre analyse de la vue et du contrôleur en explicitant nos réflexions autour de l'interface graphique.

1.1 Modèle

Puisqu'elle nous parraissait au coeur du problème, nos premières réflexions ont porté sur la modélisation de la scène. Ce modèle comporte une partie disposition et une partie observation.

1.1.1 Disposition de la scène

Lors de notre analyse nous avons identifié tous les objets positionnables : les éléments , les sources de lumière (la lumière ambiante étant inhérente à la scène) et les observateurs. Nous avons décidé de créer une classe abstraite pour caractériser les objets positionnables, la classe `Positionable`. Bien que cette classe soit essentiellement un moyen de sous-typer, chaque positionnable possède une caractéristique commune : un nom, dont l'utilité sera développée dans la section relative à l'interface graphique.

Les éléments Premiers composants positionnables : les éléments. Nous devions être en mesure de créer des sphères, des plans et des cubes, ces derniers éléments étant composés de facettes, il fallait aussi pouvoir créer des facettes. Ainsi, nous avons créé les classes `Sphere`, `Plan`, `Triangle` et `Cube` qui héritent toutes de `Element`. Un élément est construit à partir d'un nom et d'un matériau (cf Utilitaires). La classe `Element` doit permettre d'accèder au nom de l'élément et à son matériau. De plus, un élément est avant-tout un positionnable susceptible d'être intersecté par un rayon. De même, chaque élément doit pouvoir donner sa normale ainsi que les rayons réfléchi et réfracté en un point. On retrouvera donc une série de méthodes communes à tous les éléments :

- `getIntersection` Obtenir le point d'intersection entre un élément et un rayon.
- `getNormal` Obtenir le vecteur normal unitaire, dirigé vers l'extérieur, en un certain point d'un élément.
- `getReflechi` Obtenir le rayon réfléchi en un point du matériau.
- `getRefracte` Obtenir le rayon refracté en un point du matériau.

Les sources de lumière Une autre catégorie d'éléments positionnables est celle des sources de lumière `SpotLight`. Une source de lumière est modélisée par un point et une intensité (i.e. une couleur). Il s'agit d'une classe relativement simple qui n'est composée que d'accesseurs et mutateurs.

Les points de vue La classe `ViewPoint` est créée pour contenir toutes les informations caractérisant une vue de la scène. Elle comporte donc principalement un point source (origine du regard) et un écran `Screen`.

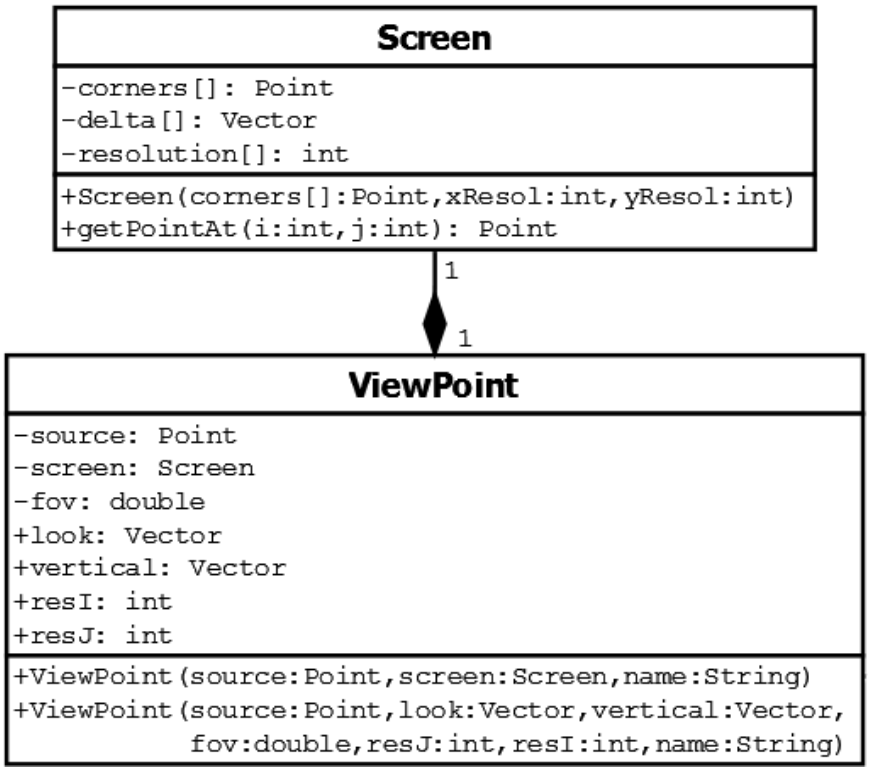

FIGURE 1 – Les points de vues : `ViewPoint.class` et `Screen.class`

Les objets `Screen` doivent représenter des matrices de points dans l'espace. Ils sont donc définis par trois coins nommés `corners` et une résolution en J et en I qui donnent la densité de points en largeur et en hauteur. On peut alors obtenir n'importe quel point de l'écran en utilisant la méthode `getPointAt`.

Au départ, un seul constructeur de `ViewPoint`, prenant en paramètre le point source et l'écran, était disponible. Cependant nous avons par la suite introduit un constructeur plus intuitif à utiliser, qui crée lui-même l'écran dont il sera composé. Ce constructeur prend en argument un point source, deux vecteurs indiquant la direction du regard (`look`) et la direction de la verticale (`vertical`), un angle de vision (`fov`), une résolution en J et en I. Ces arguments, bien que non nécessaires aux méthodes de ces classes, sont stockés en attributs pour qu'ils puissent être restitués lorsqu'une instance de `ViewPoint` est modifié au travers de l'interface graphique.

1.1.2 Observation de la scène

Le lancer de rayon en substance, pour décharger la classe `Scene` est exporté dans la classe abstraite `RayTracer`. Cette classe contient donc des méthodes de classe qui permettent de calculer la couleur vue par un rayon donné.

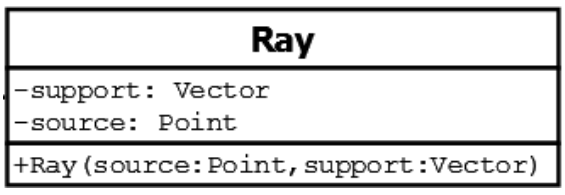

FIGURE 2 – Les rayons : `Ray.class`

Les rayons sont décrits par la classe `Ray` qui n'est que l'association d'un `Vector` et d'un `Point`.

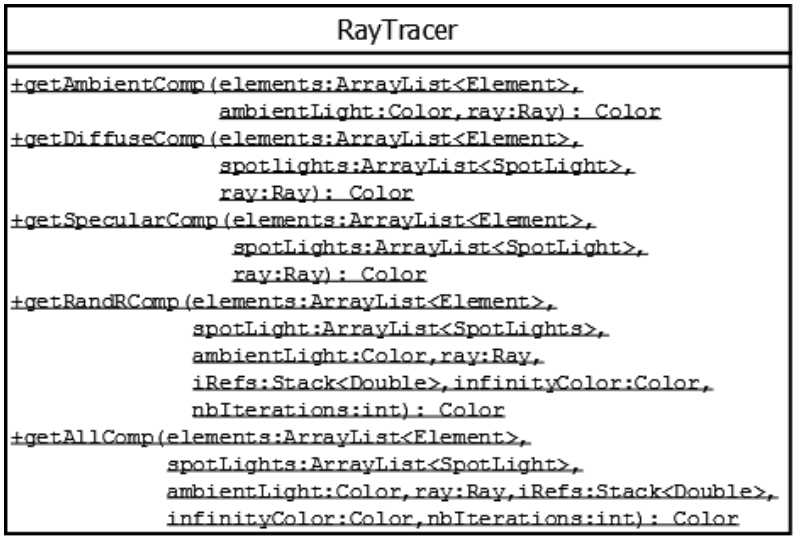

FIGURE 3 – Les rayons : `Ray.class`

`RayTracer` dispose de quatre méthodes qui calculent respectivement les composantes ambiantes, diffuse, spéculaire et réflechie/réfractée ainsi qu'une méthode `getAllComp` qui calcule la combinaison de toutes ces composantes mais de manière plus optimisée cette fois. Nous avons préféré séparer ces composantes dans un premier temps pour permettre une implémentation en profondeur de l'application. Nous pouvions ainsi avoir des premiers résultats avec la seule composante ambiante, puis implémenter par la suite les autres composantes. De manière générale, les arguments de ces méthodes correspondent aux informations nécessaires au calcul des intersections d'éléments (liste des éléments de la scène), au calcul des lumières visibles (liste des `SpotLight`), et d'autres caractéristiques de la scène comme la couleur de la lumière ambiante. Nous avons cependant ajouté à posteriori une pile d'indice de réfraction en argument de `getRAndRComp` (pour la composante réfléchie et réfractée). En effet, cette méthode récursive avait besoin de se souvenir des indices des milieux qui avaient été traversés. Ce point sera expliqué plus en détail dans la partie implémentation du rapport.

1.2 Interface graphique

L'interface homme-machine (IHM) de notre application repose sur l'interface *Swing* de la bibliothèque `javax`. Le but de cette IHM est de permettre aux utilisateurs de disposer des objets au sein d'une scène préalablement créée. L'interface doit donc être la plus simple possible. Ainsi, nous avons opté pour une fenêtre répartie en 3 sections : une barre de menu, une liste d'objets placés dans la scène, et un aperçu des différentes vues.

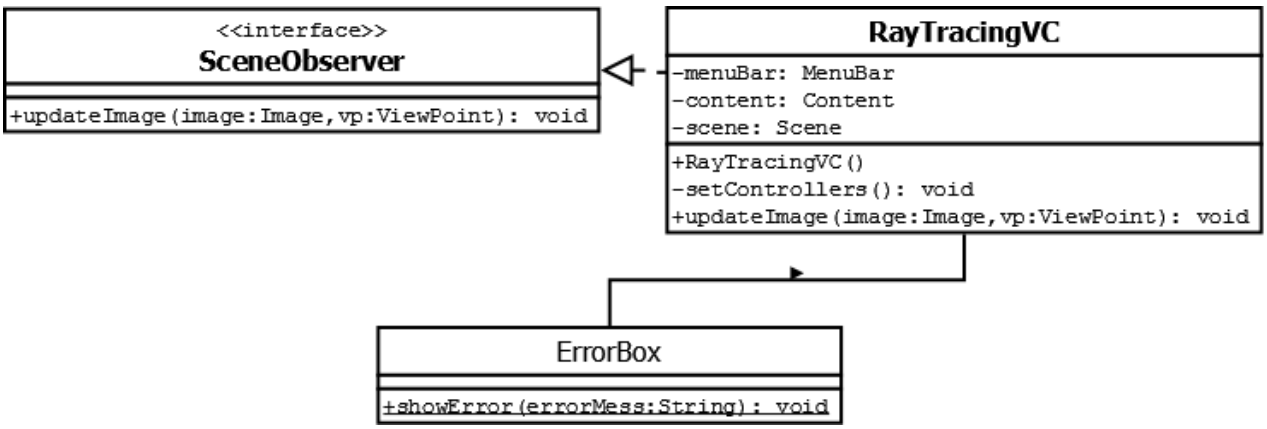

FIGURE 4 – La vue et contrôleur principale : `RayTracingVC.class`

Nous voulions ensuite que la vue présente certaines fonctionnalités essentielles et d'autres plus annexes. Par manque de temps, nous n'avons pu implémenter toutes les fonctionnalités annexes voulues ; l'interface reste cependant fonctionnelle. Parmi ces fonctionnalités, nous retrouvons l'ajout d'éléments, de points de vue et de sources de lumière, mais également la suppression et la modification de ces différents objets positionnables. De plus, il est également possible d'exporter une vue au format `.ppm`. N'ayant pu réaliser un processus d'enregistrement/ouverture de "projet", l'exportation s'effectue dans le dossier d'exécution.

2 Implémentation du modèle

2.1 Utilitaires

Afin d'implémenter tout le code relatif au tracé de rayon, nous avons déporté certains calculs dans des classes externes dites classes utilitaires.

2.1.1 La classe Geometry

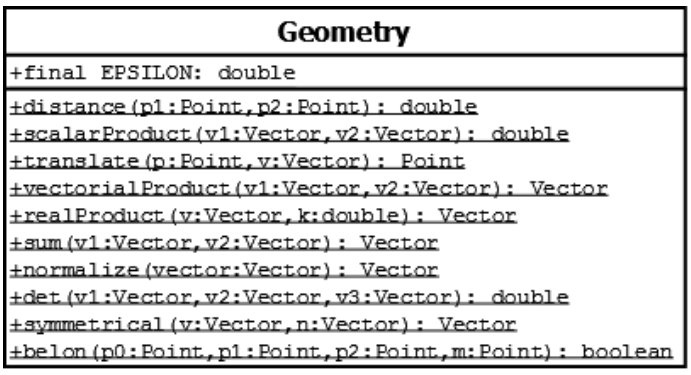

FIGURE 5 – La manipulation des points et vecteurs : `Geometry.class`

L'outil géométrie, incarné par la classe abstraite `Geometry`, est utile à tous les étages du programme. Elle doit permettre d'effectuer certains calculs vectoriels de base tels que le produit vectoriel, le produit scalaire ou le calcul de déterminant. La classe `Geometry` manipule donc des `Vector` et des `Point` qui représentent les objets vecteur et point au sens mathématiques.

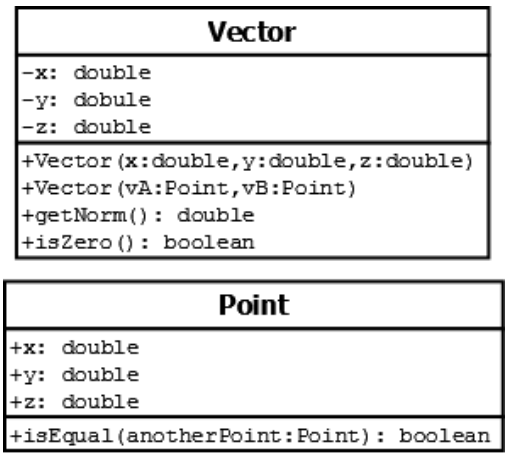

FIGURE 6 – Les points et vecteurs : `Vector.class` et `Point.class`

2.1.2 Materiaux

La classe `Material` nous permet également d'externaliser le stockage des divers coefficients à fournir (absorption, réfléctance, indice de réfraction) pour caractériser un élément. De fait, elle ne possède aucune réélle commande et les différentes requêtes servent à obtenir les valeurs des coefficients.

2.2 Disposition de la scène

2.2.1 Les éléments

Un élément se caractérise avant-tout par sa capacité à répondre aux différentes questions du traceur (par exemple, fournir sa normale depuis un point d'intersection). Cependant, seul le calcul des rayons réfléchi et réfracté s'effectue sans connaissance a priori de l'élement.

Listing 1 – Méthode getReflected

```
public Ray getReflected(Ray incRay, Point intersection){
    Vector normal = this.getNormal(intersection);
    double rayScalarNormal = Geometry.scalarProduct(
        incRay.getSupport(), normal
    );

    if(rayScalarNormal > 0)
        normal = Geometry.realProduct(normal, -1);

        return new Ray(intersection, Geometry.symmetrical(
            incRay.getSupport(), normal)
        );
}
```

Le calcul du rayon réfléchie utilise la classe `Geometry` pour effectuer le calcul du symmétrique d'un vecteur par rapport à la normale. Une vérification sur le sens de cette normale est toutefois effectuée avant calcul. En effet, la formule fournie est exacte uniquement pour des rayons externes aux objets. Le problème se pose donc lors du calcul de la composante réfractée alors que les rayons sont susceptibles de traverser les éléments.

Listing 2 – Méthode getRefracted

```
public Ray getRefracted(Ray incRay,Point intersection,Stack<Double> iRefs){
    Vector normal = this.getNormal(intersection);
    double rayScalarNormal = Geometry.scalarProduct(
        incRay.getSupport(),
        normal
    );

    // Récupération des indices de réfractions, et mise à jour de la pile.
    double nRef = iRefs.peek();
    double nRefPrim = this.getMaterial().getIRefraction();

    //rayScalarNormal > 0 => on quitte l'element.
    if(rayScalarNormal > 0){
        normal = Geometry.realProduct(normal, -1);
        rayScalarNormal *= -1;
        iRefs.pop();
        try{
            nRefPrim = iRefs.peek();
        }catch(EmptyStackException e){
            throw new RuntimeException(
                "Point de vue à l'interieur d'un objet");
        }
    }else{
        iRefs.push(nRefPrim);
    }

    // Calculer le rayon refracté
    double coeffNormal = (nRefPrim*nRefPrim)/(nRef*nRef) +
        rayScalarNormal*rayScalarNormal -1;

    coeffNormal = -(rayScalarNormal + Math.sqrt(coeffNormal));

    Vector vRefrac = Geometry.realProduct(
        Geometry.sum(incRay.getSupport(),
            Geometry.realProduct(normal, coeffNormal)),
        nRef/nRefPrim
    );

    return(new Ray(intersection, vRefrac));
}
```

Le calcul du rayon réfracté est un peu plus complexe. Il nécessite notament la manipulation d'une pile d'indices de réfraction. C'est d'ailleurs au sein de cette méthode que s'opére toutes les modifications de cette pile. De fait, lorsqu'un rayon entre dans un élément, son indice de réfraction est ajouté au sommet de la pile. Lorsqu'il en sort, on retire le sommet de la pile (correspondant à l'indice de l'élément). L'entrée et la sortie sont déterminées par un produit scalaire entre le rayon incident et la normale (par convention extérieure) à l'élément. Bien que cette méthode soit relativement simple, elle possède une faiblesse. Aucun point de vue ne peut-

être placé à l'intérieur d'un élément transparent (les plans infinis sont en réalité considérés comme des zones définissant un demi-espace). Une fois les indices de réfraction entrant et sortant calculés, le rayon réfracté est calculé à l'aide de la formule fournie.

En outre, un élément doit également pouvoir renseigner son point d'intersection par un rayon, ou encore sa normale en un point donné. Ces méthodes étant propres à l'élément choisi (cube, sphère ou plan), elles ont simplement été déclaré `abstract` pour forcer leur réécriture par les classes filles.

La sphère La sphère a été le premier positionnable à avoir été implémenté de part sa simplicité (vis à vis des autres positionnables). En effet, les calcul de la normale ou de l'intersection peuvent s'écrire assez rapidemment avec toutefois certaines précautions. La détermination de l'intersection est notament sujette à confusion. La formule fournie permet en effet d'obtenir l'intersection entre une sphère et une droite. On considère cependant ici des demi-droites : les rayons ont un point source et un sens de propagation. D'autres problèmes se posent lorsque le point source du rayon se trouve sur la sphère (notament lors du calcul de la composante diffuse). Il a donc fallu être prudent dans le developpement de la méthode `getIntersection`.

Listing 3 – Méthode getRefracted

```
public Point getIntersection(Ray ray){
    Vector SC =  new Vector(ray.getSource(), this.center);
    double dScalSC = Geometry.scalarProduct(ray.getSupport(),SC);
    double delta = this.radius*this.radius+dScalSC*dScalSC-
        Math.pow(SC.getNorm(),2);

    if(delta < 0)
        throw new NoIntersectionException();

    double t1 = dScalSC-Math.sqrt(delta);
    double t2 = dScalSC+Math.sqrt(delta);

    double t = (t1 < 0) ? t2 : Math.min(t1,t2);
    double tSymm = t1+t2-t;

    Point intersection2 = Geometry.translate(ray.getSource(),
        Geometry.realProduct(ray.getSupport(), tSymm)
    );
    Point intersection = Geometry.translate(ray.getSource(),
        Geometry.realProduct(ray.getSupport(), t)
    );

    if(intersection.isEqual(ray.getSource())){
        intersection = intersection2;
    }

    if(Geometry.scalarProduct(ray.getSupport(),
        new Vector(ray.getSource(), intersection)) < 0)
        throw new NoIntersectionException();

    return intersection;
```

```
}
```

Le plan Conformément au sujet, nous avons défini le constructeur de plan avec trois points en paramètre. A partir de ces points il fallait calculer, connaissant le rayon incident(si il y a intersection), le point d'intersection et la normale en ce point.
Concernant le plan, le calcul de la normale est très simple et ne nécessite pas de connaitre le point d'intersection. Il nous a suffit d'utiliser le produit vectoriel compris dans `Geometry`.

Listing 4 – Méthode getNormal

```
public Vector getNormal(Point intersection){
    Vector resultat = Geometry.vectorialProduct(
        new Vector(this.points[0],this.points[1]),
        new Vector(this.points[0],this.points[2])
    );

    return Geometry.normalize(resultat);
}
```

Le calcul du point d'intersection est plus ardu. Il a d'abord fallu résoudre le système permettant d'obtenir α, β et γ, pour cela nous avons utilisé les formules de Cramer, s'aidant de la fonction déterminant elle aussi définie dans `Geometry`.

Listing 5 – Méthode solveSystem

```
protected Vector solveSystem(Ray ray){
    double detSystem = Geometry.det(
                new Vector(points[0],points[1]),
                new Vector(points[0], points[2]),
                Geometry.realProduct(ray.getSupport(),-1)
    );

    if(Math.abs(detSystem) < Geometry.EPSILON)
        throw new NoIntersectionException();

    double alpha = Geometry.det(
                new Vector(this.points[0],ray.getSource()),
                new Vector(this.points[0], this.points[2]),
                Geometry.realProduct(ray.getSupport(),-1));

    double beta = Geometry.det(
                new Vector(this.points[0],this.points[1]),
                new Vector(this.points[0], ray.getSource()),
                Geometry.realProduct(ray.getSupport(),-1));

    double gamma = Geometry.det(
                new Vector(this.points[0],this.points[1]),
                new Vector(this.points[0], this.points[2]),
                new Vector(this.points[0],ray.getSource()));
```

```
    Vector resultat = Geometry.realProduct(
        new Vector(alpha,beta,gamma),
        1/detSystem
    );

    return resultat;
}
```

Des exceptions sont levées si les points définissant le plan sont alignés (determinant nul) ou si le rayon s'éloigne du plan plutôt que de s'en rapprocher (signe du produit scalaire du support du rayon avec le vecteur source du rayon). La formule donnée dans le sujet suffit ensuite à déterminer le point d'intersection à partir de γ. Nous avons également choisi par convention qu'un rayon partant du plan ne serait pas intersecté, peu importe sa direction. Les plans étant d'épaisseur nulle, la détermination du sens de propagation depuis le plan se révele ardue.

Listing 6 – getIntersection

```
public Point getIntersection(Ray ray){
    Vector v = this.solveSystem(ray);
    Point resultat = Geometry.translate(
        ray.getSource(),
        Geometry.realProduct(ray.getSupport(), v.getZ())
    );

    if(!resultat.isEqual(ray.getSource())){
        if(Geometry.scalarProduct(ray.getSupport(),
            new Vector(ray.getSource(), resultat)) < 0)
            throw new NoIntersectionException();
    }else{
        throw new NoIntersectionException();
    }

    return     resultat;
}
```

La facette Les méthodes permettant la détermination du point d'intersection et de la normale sont très proches de plan. L'obtention de la normale ne diffère d'ailleurs pas. Pour l'intersection, on reprend le calcul du système linéaire mais cette fois-ci, les valeurs de α et β sont utilisées pour vérifier que le point appartient à la facette. Ceci est possible grâce à l'héritage de la méthode `solveSystem` de plan (visibilité protégée).

Le cube L'implémentation du cube est plus intéressante. Nous avons choisi de créer le cube à partir du moins de points possibles, c-a-d trois points(p_0, p_1 et p_2). Ces trois points décrivent la facette permettant de générer le reste du cube. Le premier point sert d'origine et permet de former deux vecteurs $\overrightarrow{(}p_0p_1)$ et $\overrightarrow{(}p_0p_2)$. Le cube est ensuite extrudé selon le sens direct déterminé par ces deux vecteurs.

Listing 7 – Détermination du vecteur profondeur

```
Vector x = new Vector(p_0,p_1);
Vector y = new Vector(p_0,p_2);
Vector z = Geometry.realProduct(
    Geometry.normalize(Geometry.vectorialProduct(x,y)),
    x.getNorm()
);
```

Dans un premier temps on introduira, à partir du calcul de vecteurs intermédiaires, les points extrémités du cube. Cela fait, on génère toutes les facettes permettant de construire le cube (deux facettes par face).

Listing 8 – Création des sommets

```
Point p_3 = Geometry.translate(p_1,y);
Point p_4 = Geometry.translate(p_0,z);
Point p_5 = Geometry.translate(p_1,z);
Point p_6 = Geometry.translate(p_2,z);
Point p_7 = Geometry.translate(p_3,z);

//Faces du fond
this.triangles[0] = new Triangle(new Point[]{p_0,p_2,p_1}, name, material);
this.triangles[1] = new Triangle(new Point[]{p_3,p_1,p_2}, name, material);

...
```

Pour le calcul de l'intersection, chaque facettes est analysée. On trouve alors deux ou aucun (si le rayon incident passe par une arête ou un coin, il est ignoré) point(s) d'intersection. Si des intersections sont trouvées, la plus proche du point source du rayon est retournée. Le calcul de la normale s'effectue de facon similaire.

Listing 9 – Méthode getIntersection

```
public Point getIntersection(Ray ray){
    Point closestIntersection = null;
    Point intersection;
    Point source = ray.getSource();
    double distance;
    double distMin = Double.POSITIVE_INFINITY;

    try{
        for(int i = 0; i < 12; i++){
            try{
                intersection = this.triangles[i].getIntersection(ray);
                distance = Geometry.distance(intersection, source);

                //Lorsqu'on est sur une arete.
                if(Math.abs(distance - distMin) < Geometry.EPSILON)
                    throw new RuntimeException("Arête");

                if(distance < distMin){
```

```
                    distMin = distance;
                    closestIntersection = intersection;
                }
            }catch(NoIntersectionException e){}
        }
    }catch(RuntimeException e){
            closestIntersection = null;
    }

    if(closestIntersection == null)
            throw new NoIntersectionException();

    return closestIntersection;
}
```

Nous avons réalisé lors de nos premiers tests complets que l'introduction d'un cube dans l'environnement ralentissait considérablement le temps de calcul de l'image. En effet, pour chaque cube, douze plans sont générés, et même si la plupart des rayons ne traversent pas le cube, le calcul d'intersection doit être effectué, entrainant ainsi des calculs souvent inutiles. Pour palier ce problème, nous avons inséré le cube dans une sphère fictive. S'il y a intersection entre le rayon et la sphère, on entame les calculs décrits précédement, sinon, on arrête les calculs, il n'y aura pas d'intersection. Nous avons ainsi constaté des temps de calculs jusqu'à deux fois inférieurs par rapport à des tests sans optimisation.

Listing 10 – Optimisation

```
double cote = Geometry.distance(this.vertexs[0],this.vertexs[1]);
Vector SC =  new Vector(ray.getSource(), this.center);
double dScalSC = Geometry.scalarProduct(ray.getSupport(),SC);

// optimisation de l'intersection, on teste d'abord l'intersection avec
// la sphere de diagonale la grande diagonale du cube
if( (Math.sqrt(3)*cote/2 -
    (Geometry.sum(
        Geometry.realProduct(ray.getSupport(),-dScalSC),
        SC
    ).getNorm()) < Geometry.EPSILON))

    throw new NoIntersectionException();

//Sinon calculs précédents.
```

2.2.2 Les sources de lumière

2.2.3 Les points de vue

Les écrans sont décrits comme des matrices de points dans l'espace. Or nous avons choisi de ne pas réellement garder en mémoire un tableau à deux dimensions de points. Au lieu de cela, il semblait plus judicieux de ne garder en attribut que les coins définissant l'écran, ainsi que deux vecteurs contenus dans le tableau `delta[]`. Ces vecteurs correspondent alors à la différence

entre deux points consécutifs de l'écran, en largeur et en hauteur. Lorsque l'on fait appel à la méthode `getPointAt`, les coordonnées du point demandé sont calculées par translations du point origine de l'écran.

Listing 11 – Méthode `getPointAt`

```
public Point getPointAt(int i, int j){
    if(i < 0 || j < 0 ||
       i >= this.resolution[0] ||
       j >= this.resolution[1])
        throw new IllegalArgumentException("Selection d'un point" +
                            " en dehors de l'ecran");

    Point requestedPoint = new Point(
                    this.corners[0].getX(),
                    this.corners[0].getY(),
                    this.corners[0].getZ()
                    );

    for(int k = 0; k < i; k++){
        requestedPoint = Geometry.translate(requestedPoint, this.delta[0]);
    }

    for(int l = 0; l < j; l++){
        requestedPoint = Geometry.translate(requestedPoint, this.delta[1]);
    }

    return requestedPoint;
}
```

Le constructeur « user-friendly » du `ViewPoint` quant à lui est fait pour être assez flexible aux entrées de l'utilisateur. Il n'est plus question ici de construire un `Screen` soi-même à partir de trois points qui forment deux vecteurs orthogonaux. L'important est de définir un repère orthogonal. Pour cela il faut fournir le point source du regard, un premier vecteur de direction du regard `look` et un deuxième vecteur de direction de la verticale `vertical`. Ce deuxième vecteur ne doit pas être colinéaire au premier. Ainsi il suffit d'orthonormaliser ces deux vecteurs et de calculer par un produit vectoriel un troisième vecteur pour compléter ce repère orthonormé. En prenant comme convention de placer le centre de l'écran au bout du vecteur `look`, le `ViewPoint` est complètement défini car l'angle de vue demandé couplé aux résolutions exigées suffisent à imposer une taille d'écran. Ce raisonnement est résumé sur la figure suivante.

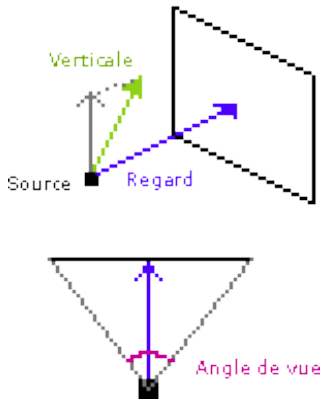

FIGURE 7 – La création d'un `ViewPoint`

2.3 Observation de la scène

Nous discutons ici de l'implémentation de la classe `RayTracer`. La méthode qui nous intéresse est `getAllComp`, celle qui calcule la couleur vue dans la direction d'un rayon en prenant en compte toutes les composantes du modèle de Blinn-Phong et de manière optimisée. Dans les méthodes de `RayTracer`, il est souvent nécessaire de manipuler des listes d'objets (lumières, éléments) ou bien de renvoyer des couples d'objet (un élément intersecté et son point d'intersection). Il a donc été choisi d'utiliser les `ArrayList` pour nos listes, et des `Hashtable` pour nos couples. Nous avons, pour simplifier le code, exporté et factoriser dans des méthodes privées certains calculs spécifiques :

- `getClosestInter` est une méthode qui renvoie le plus proche élément intersecté par un rayon, accompagné du point d'intersection obtenu. Cette méthode peut lever une exception `NoIntersectionException` qui pourra être gérée par les programmes appelant lorsque le rayon rencontre l'infini.
- `getVisibleSpots` renvoie la liste des lumières visibles depuis un point de la scène (utile pour le calcul de la composante diffuse et spéculaire).
- `ambiantComp`, `diffuseComp`, `specularComp` qui appliquent les formules données par le modèle de Blinn-Phong et calculent ainsi les couleurs vues pour chacune des composantes. Elles ne s'occupent pas des intersections de rayons et demandent donc de fournir en argument l'élément intersecté, le point d'intersection et éventuellement la liste des lumières visibles depuis ce point. Elles calculent essentiellement les angles entre rayons, et appliquent les coefficients du matériau considéré.

Listing 12 – Les méthodes privées de `RayTracer`

```
/** Obtenir l'intersection d'un rayon avec un élément la plus
* proche. Lève une exception si aucun objet n'est rencontré.
* @param elements La liste d'elements a considerer.
* @param ray Le rayon a lancer.
* @return Une Hashtable contenant l'association (point
* d'intersection / element intersecté le plus proche)
* @throws NoIntersectionException Lorsqu'aucun n'objet n'est
* intersecté
*/
private static Hashtable<Point, Element> getClosestInter(
    ArrayList<Element> elements,
```

```
    Ray ray
);


/** Obtenir la liste des lumières visibles depuis un point
* d'intersection à partir d'une liste d'éléments
* @param elements La liste d'elements a considerer.
* @param spotLights La liste des lumières de la scene
* @param incRay Le rayon incident
* @param elInter L'element intersecté
* @param intersection le point d'intersection
* @return La liste des lumières visible
* @throws NoIntersectionException Lorsqu'aucun n'objet n'est
* intersecté
*/
private static ArrayList<SpotLight> getVisibleSpots(
    ArrayList<Element> elements,
    ArrayList<SpotLight> spotLights,
    Ray incRay,
    Element elInter,
    Point intersection
);

/** Obtenir la composante ambiante a partir d'un point
* d'intersection, d'un élément intersecté et d'une lumière
* ambiante.
* @param elInter L'élément intersecté
* @param intersection Le point d'intersection
* @param ambientLight Source de lumiere ambiante.
* @return La couleur de la composante ambiante
*/
private static Color ambientComp(
    Element elInter,
    Point intersection,
    Color ambientLight
);

/** Obtenir la composante diffuse a partir d'un point
* d'intersection, d'un élément intersecté et d'une liste de
* sources de lumière visibles.
* @param elements Liste d'elements consideres
* @param spotLights Liste des sources de lumières visibles depuis
* le point d'intersection
* @param elInter L'élément intersecté
* @param intersection Le point d'intersection
* @return La couleur de la composante diffuse
*/
private static Color diffuseComp(
    ArrayList<Element> elements,
    ArrayList<SpotLight> visibleSpots,
```

```
    Element elInter,
    Point intersection
);

/** Obtenir la composante spéculaire a partir d'un point
* d'intersection, d'un élément intersecte, d'une liste de
* sources de lumière visibles, et d'un rayon incident.
* @param elements Liste d'elements consideres
* @param visibleSpots Liste des sources de lumières visibles
* depuis le point d'intersection
* @param elInter L'élément intersecté
* @param intersection Le point d'intersection
* @param ray Le rayon incident
* @return La couleur de la composante spéculaire
*/
private static Color specularComp(
    ArrayList<Element> elements,
    ArrayList<SpotLight> visibleSpots,
    Element elInter,
    Point intersection,
    Ray ray
);
```

C'est la méthode `getAllComp` qui gère les intersections du rayon avec les éléments de la scène, les calculs des rayons réfléchis/réfractés et la recherche des lumières visibles depuis un point d'un élément. Elle fait donc essentiellement appelle aux fonctions ci-dessus et récupère leurs résultats pour obtenir facilement les trois premières composantes du modèle. En guise d'optimisation, elle teste si le matériau de l'élément rencontré est affecté par chacune des composantes (en regardant les coefficients associés) et ne calcule alors que celles dont les coefficients sont non nuls. Faisant appel à la méthode `getClosestInter`, cette méthode peut par extension lever une `NoIntersectionException`, auquel cas la `Scene` qui l'aura appelé saura que le résultat attendu est la couleur de l'infini.

La subtilité vient lorsqu'il s'agit de calculer les composantes réfléchies et réfractées. D'une part, la méthode `getAllComp` est récursive. Elle prend en paramètre un nombre d'itérations `nbIteration` qui correspond au nombre d'appels récursifs effectués. Ainsi après avoir calculé les trois premières composantes du modèle, elle calcule les rayons réfléchis/réfractés et se rappelle sur ces rayons avec `nbIteration` décrémenté. Elle doit donc cette fois-ci gérer elle-même les `NoIntersectionException` levées récursivements, et doit donc recevoir en argument la couleur de l'infini qu'elle renverra dans ces cas-là. D'autre part, les étapes de cet algorithme récursifs nous offre une vision réduite du milieu traversé par le rayon. On connaît en effet que le milieu dans lequel nous entrons ou duquel nous sortons (élément intersecté) mais pas forcément celui dans lequel nous allons. Il faut donc, par le moyen de piles (`Stack<Double>`), se souvenir des indices de réfractions des milieux traversés auparavant. Tant que nous entrons dans de nouveaux objets (cette information est donnée par le sens de la normale au point d'intersection), nous empilons les valeurs des indices de réfractions rencontrés. Et dès que nous sortons d'un objet, il suffit de dépiler l'indice de sommet de pile pour retrouver l'indice précédent. Cette solution fonctionne pour la plupart des cas, mais il est par exemple impossible de lancer notre premier rayon depuis l'intérieur d'une sphère, auquel cas se retrouverai avec une

pile vide.

Listing 13 – `getAllComp`

```
/** Obtenir la somme logarithmique des quatres composantes du
* modele de Blinn-Phong a partir d'une liste d'elements, d'une
* liste de sources de lumiere ponctuelles, d'une source de
* lumiere ambiante, d'un rayon, d'un indice de réfraction et d'un
* nombre d'itérations
* @param elements Liste d'elements consideres
* @param spotLights Liste de sources de lumieres considerees
* @param ambientLight Source de lumiere ambiante.
* @param ray Rayon a lancer.
* @param iRefs Pile d'indice de refraction des milieux rencontrés
* @param infinityColor Couleur de l'infini.
* @param nbIterations Limite le nombre de deviations considerees.
* @return La couleur de la composante réfléchie et réfractée
*/
public static Color getAllComp(
    ArrayList<Element> elements,
    ArrayList<SpotLight> spotLights,
    Color ambientLight, Ray ray,
    Stack<Double> iRefs,
    Color infinityColor,
    int nbIterations
){
    Color colorResult = new Color(0, 0, 0); // Pas la couleur de l'infini

    if(nbIterations != 0) {
        //Duplication des piles pour ne pas que la refraction perturbe
        //la reflexion.
        @SuppressWarnings("unchecked")
        Stack<Double> iRefsT = (Stack<Double>)iRefs.clone();
        @SuppressWarnings("unchecked")
        Stack<Double> iRefsR = (Stack<Double>)iRefs.clone();

        // Recherche de l'intersection.
        Hashtable<Point, Element> intersectedElement =
            getClosestInter(elements, ray);
        Point intersection = intersectedElement.keys().nextElement();
        Element elInter = intersectedElement.get(intersection);

        // Recherche des lumières visibles
        ArrayList<SpotLight> visibleSpots;
        visibleSpots = getVisibleSpots(
            elements, spotLights, ray, elInter, intersection
        );
        // Ajouter les trois premières composantes
        if (!(elInter.getMaterial().getCAmbient().isEqual(
            new Color(0, 0, 0)))) {
            colorResult.logAdd(ambientComp(
```

```
                elInter, intersection, ambientLight)
            );
        }
        if (!(elInter.getMaterial().getCDiffuse().isEqual(
            new Color(0, 0, 0)))) {
            colorResult.logAdd(diffuseComp(
                elements, visibleSpots, elInter, intersection)
            );
        }
        if (!(elInter.getMaterial().getKSpecular() < Geometry.EPSILON)) {
            colorResult.logAdd(specularComp(
                elements, visibleSpots, elInter, intersection, ray)
            );
        }

        // S'occuper des reflexion refractions.
        if (!(elInter.getMaterial().getCReflection().isEqual(
            new Color(0, 0, 0)))) {
            // Calculer le rayon réflechi
            Ray rayReflec = elInter.getReflected(ray, intersection);
            // Obtenir la composante réfléchie dans la direction
            // du rayon réfléchi, avec un nbIteration décrémenté
            // et un nouvel iRef pour la réfraction
            Color cReflec = infinityColor;
            try{
                cReflec = (getAllComp(elements, spotLights, ambientLight,
                rayReflec, iRefs, infinityColor,
                (nbIterations - 1)));
            } catch(NoIntersectionException e){}
            // Sommer logarithmiquement les composantes et appliquer les
            // bons coefficients
            cReflec = cReflec.multiply(
                elInter.getMaterial().getCReflection()
            );
            colorResult.logAdd(cReflec);
        }

        if (!(elInter.getMaterial().getCTransmission().isEqual(
            new Color(0, 0, 0)))) {
            // Calculer le rayon refracté
            Ray rayRefrac = elInter.getRefracted(
                ray, intersection, iRefsT
            );
            // Obtenir la composante réfléchie dans la direction
            // du rayon refracté, avec un nbIteration décrémenté
            // et un nouvel iRef pour la réfraction
            Color cRefrac = infinityColor;
            try{
                cRefrac = (getAllComp(elements, spotLights, ambientLight,
                    rayRefrac, iRefsT, infinityColor,(nbIterations - 1))
```

```
                );
            } catch(NoIntersectionException e){}
            // Sommer logarithmiquement les composantes et appliquer les
            // bons coefficients
            cRefrac = cRefrac.multiply(
                elInter.getMaterial().getCTransmission()
            );
            colorResult.logAdd(cRefrac);
        }
    }
    return colorResult;
}
```

3 Implémentation de l'interface graphique

L'interface homme-machine (IHM) de notre application repose sur l'interface Swing de la bibliothèque `javax`. Le but de cette IHM est de permettre aux utilisateurs de disposer des objets au sein d'une scène préalablement créée. L'interface doit donc être la plus simple possible. Ainsi, nous avons opté pour une fenêtre répartie en trois sections : une barre de menu, une liste d'objets placés dans la scène, et un aperçu des différentes vues.

Nous voulions ensuite que la vue présente certaines fonctionnalités essentielles et d'autres plus annexes. Par manque de temps, nous n'avons pu implémenter toutes les fonctionnalités annexes voulues ; l'interface reste cependant fonctionnelle. Parmi ces fonctionnalités, nous retrouvons l'ajout d'éléments, de points de vue et de sources de lumière, mais également la suppression et la modification de ces différents objets positionnables. De plus, il est également possible d'exporter une vue au format `.ppm`. N'ayant pu réaliser un processus d'enregistrement/ouverture de "projet", l'exportation s'effectue dans le dossier d'exécution.

3.1 La barre de menu

Dans cette section, nous traiterons des informations relatives à la barre de menu qui compose notre fenêtre principale. Celle-ci hérite directement de l'objet `JMenuBar` proposé par Swing. La seule différence notable étant le stockage des différents éléments du menu dans une table de hachage. Ceci nous permet de récupérer simplement un `JMenuItem` (agrégat de notre barre) par son nom. De fait, nous pouvons exporter la déclaration des contrôleurs dans une classe externe (ici `RayTracingVC`). Notre barre de menu possède également des méthodes privées permettant de créer de manière semi-automatique les éléments du menu ainsi que la table de hachage associée.

La barre de menu se présente donc en 3 sections : projet, scène, et aide. La section projet rends possible la création d'un nouveau projet, c-a-d une nouvelle scène. Elle permet aussi de quitter le programme. Au départ, cette section devait présenter également un sous-menu enregistrer et ouvrir. Ceci aurait permis à l'utilisateur de pouvoir sauvegarder une scène créée pour revenir travailler dessus plus tard. Le temps ayant manqué, ces fonctionnalités n'ont pu être intégré et ont donc été retirées.

La section scène permet quant à elle des actions relatives à la création/modification et suppression de positionnables. Cette section du menu permet donc l'affichage de fenêtres de saisie

correspondant aux différents positionnables.

Les fenêtres de saisie sont construites sur le même modèle. Elles héritent donc de la classe `JDialog` disponible dans l'interface Swing. Ceci nous permet d'afficher une fenêtre moins lourde que la fenêtre principale. Il aurait été intéressant, au vu de la ressemblance de toutes nos fenêtres de saisie, de mettre au point une classe intermédiaire entre la `JDialog` et les différentes fenêtres afin de factoriser certaines parties du code.

Ces fenêtres se révelent toutefois très pratiques ; Elles se chargent d'instancier les différents objets et de les ajouter à la scène. Elles provoquent donc de manière implicite un rafraichissement des vues (la scène notifie la fenêtre principale de chacun de ses changements). Leur force réside cependant dans leur aptitude à intercepter les différentes exceptions levées par les constructeurs des objets. Ainsi, si un paramètre saisi est invalide, la construction échoue, l'objet n'est pas ajouté à la scène, et la fenêtre de saisie transmet à l'utilisateur l'erreur correspondante.

Enfin, la section scène du menu présente aussi un élément permettant l'exportation d'une vue en fichier `.ppm` ASCII. L'exportation est en fait réalisée par le modèle (i.e. la scène). Le contrôleur fourni à la scène le point de vue concerné par l'opération et cette dernière se charge de convertir l'`imagePPM` stockée en une fichier `.ppm` valide.

La dernière section du menu est une section consacrée à l'aide et au fonctionnement de l'application. L'aide est accessible dès le démarrage du programme et doit permettre à un utilisateur externe d'utiliser ce dernier sans autre explication. Nous avons donc opté pour une navigation simple par liste déroulante regroupant les thèmes susceptibles de poser problème.

3.2 Le corps de l'application

Sous la barre de menu sont disposées deux parties distinctes, séparées par une barre verticale. Nous traitons ici de la partie gauche concernant la liste des positionnables ajoutés. Cette liste est mise à jour chaque fois qu'un positionnable est ajouté/modifié/supprimé. C'est de plus grâce à celle-ci qu'il est possible de sélectionner des éléments.

La partie droite de la fenêtre principale permet donc d'afficher les différentes vues. Il est possible de choisir la vue à afficher grâce à un système d'onglets (un onglet par point de vue). L'affichage se produit dès lors qu'un point de vue est créé. Chaque vue est ensuite rafraichie après ajout/suppression ou modification d'un des positionnables de la liste. Comme exigé, l'affichage s'effectue au fur et à mesure du calcul, c-a-d ligne par ligne.

4 Tests

4.1 Tests

La réalisation des tests a été effectuée selon deux aspects différents : les tests unitaires pour toutes les classes où cela était envisageable et des tests globaux plus visuels pour certaines sections tels que l'interface graphique ou le traceur de rayon.

Tests unitaires Nous avons tout au long du projet réalisé des tests unitaires de chaque classe élémentaire. La plupart des classes et classes de tests associées ont été codés par deux personnes différentes. Bien que ceci est parfois rendu la réalisation difficile, cela nous a permis de plus facilement desceller certaines erreurs. En outre, tous les tests unitaires ont été réalisés à l'aide de JUnit. L'écriture des tests a donc été quasi-automatiques et realtivement simpmle.

La réalisation des tests s'est faite au fur et à mesure du développement. La plupart des tests ont donc été réalisés en parralèle du développement des classes. Ainsi, les tests ont d'abord porté sur les classes les plus élémentaires. Ceci permet de s'assurer du bon fonctionnement de la base de l'application avant d'édifier une construction plus complexe.

Pour certaines classes , par exemple l'interface graphique, réaliser des tests unitaires n'était pas envisageable. Pour palier cette difficulté nous avons testé ces classes de manière visuelle. Nous avons donc procédé à une analyse du fonctionnement dans une utilisation normale, avant de tenter des opérations plus à risque et d'évaluer le comportement de nos classes. Par exemple, l'ajout d'un élément à partir de coefficient et valeurs valides et invalides. Nous avons donc tenter d'évaluer le comportement de notre application face à n'importe quelles situations.

Bien entendu, de nombreux tests ont été réalisés autour du traceur au fur et à mesure du développement. La séparation du calcul des composantes a permis notament de tester chaque composante une à une pour s'assurer de son fonctionnement.

Tests globaux Des tests globaux ont donc été réalisés pour contrôler le fonctionnement du `RayTracer`. Ainsi, ces tests ont été tout d'abord très simple. Nous avons vérifier l'orientation de l'image et le positionnement des objets dans l'espace. Puis, le rendu des couleurs a été testés pour chacune des composantes. Nous avons également évalué les rendus à partir de différents indices de réfraction et en modifiant le nombre de déviations maximales. Une présentation de ces tests les plus pertinents sera proposée.

Conclusion

Le projet s'est donc déroulé en deux phases. Une première phase d'analyse, durant laquelle nous avons à tour de rôle exposé nos idées et retenu les plus unanimes. C'est lors de cette phase qu'un premier diagramme UML a vu le jour. Nous y avons ajouté toutes la documentation nécessaire qui nous permettrait de se répartir le travail plus facilement. Une fois cette première phase terminée, nous avons entamé une phase de développement (qu'il s'agisse de classes ou de tests). Lors de cette phase, notre principale objectif était de se concentrer sur un seul objet : la sphère. La priorité était donné au calcul des quatre composantes. Nous avons donc entamé le développement des plans, facettes et cubes qu'une fois la majeure partie du travail effectué. De la même façon, l'interface homme-machine a été placée au second plan. Les tests sur le traceur ont été effectué depuis des fichiers .java écris à la main. Notre objectif était d'obtenir avant-tout un coeur d'application fonctionnel. Le développement de l'interface n'a cependant pas été négligé. Le développement du projet n'aura donc pas connu de temps mort. De fait, chaque début de semaine, un point été fait sur les travaux restants, et le travail effectué. Des tâches étaient assignées à chaque membre du groupe et nous avons travaillé autant que possible ensemble. Les différents conflits de développement et/ou incompréhensions n'ont donc jamais été laissé en suspens.

Nos objectifs principaux concernant le projet ont été remplis : la scène accepte les trois types d'objets exigés, et calcule pour chacun d'eux les composantes ambiante, diffuse, spéculaire, réfléchie et réfractée. Toutefois, nous n'avons pu apporter toutes les optimisations voulues à l'application. La plupart d'entre elles auraient nécessitées la réécriture partielle voir complète de certaines classes. Le traceur permet cependant d'obtenir des images assez complexes dans un temps de calcul raisonnable au vu de ce qui existe déjà dans le domaine.