Oral du projet de Programmation Objet

Maxime Arthaud   Martin Carton   Korantin Auguste

13 mai 2013

Introduction

Nous fournissons 3 choses :

- ▶ Un format de fichier pour pour pouvoir écrire des scènes à la main, dans une interface graphique ou automatiquement.
- ▶ Deux programmes :
    - ▶ Une interface graphique pour éditer un fichier représentant une scène.
    - ▶ Un programme en ligne de commande pour générer une image à partir d'un fichier de scène.
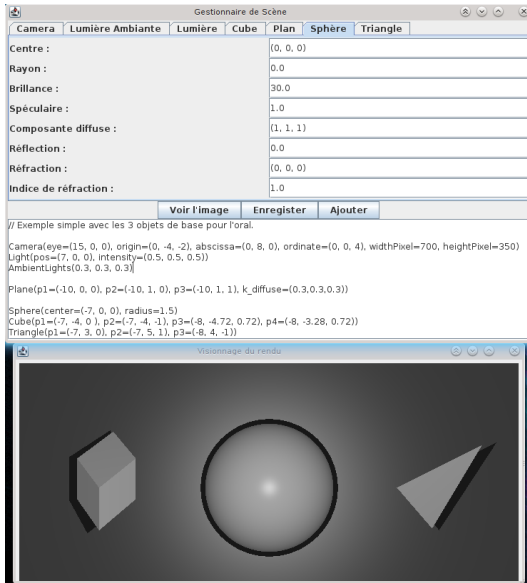
Organisation

Nous avons découpé le travail :

- Maxime a principalement travaillé sur l'interface graphique.
- Martin a principalement travaillé sur le format de fichier, l'écriture d'images PPM et les classes représentant les objets.
- Korantin a principalement travaillé sur le lancer de rayon et les propriétés optiques.

Exemple de fichier

```
// exemple de scene :
Camera(eye=(0, 0, 0), origin=(-0.5, 0, 1.5),
       abscissa=(1, 0, 0), ordinate=(0, 1, 0),
       widthpixel=200, heightpixel=200)
AmbientLights(0.2, 0.2, 0.2)
Light(pos=(5, 0, 0), intensity=(0.5, 0.5, 0.5))
Sphere(center=(0, 5, 20), radius=3.14,
       k_diffuse=(0.6,0.3,0.6))
Plane(p1=(-1,0,30), p2=(1, 0, 30), p3=(0, 1, 30))
```
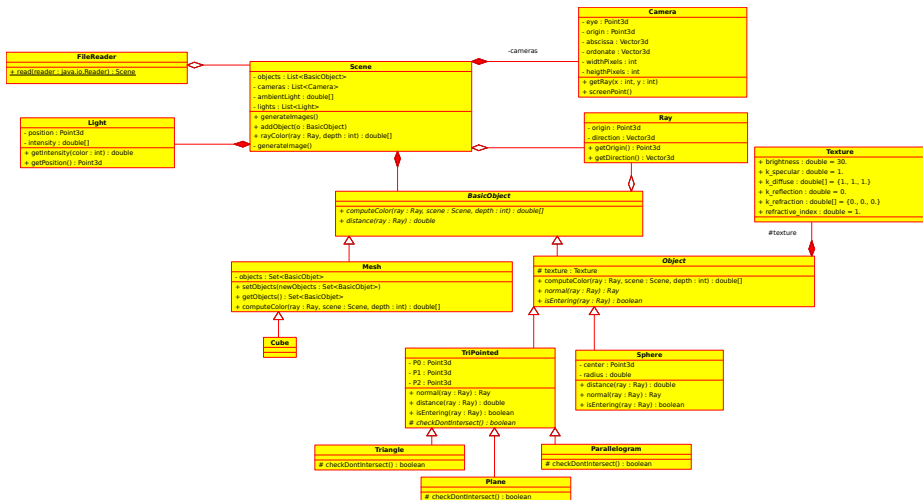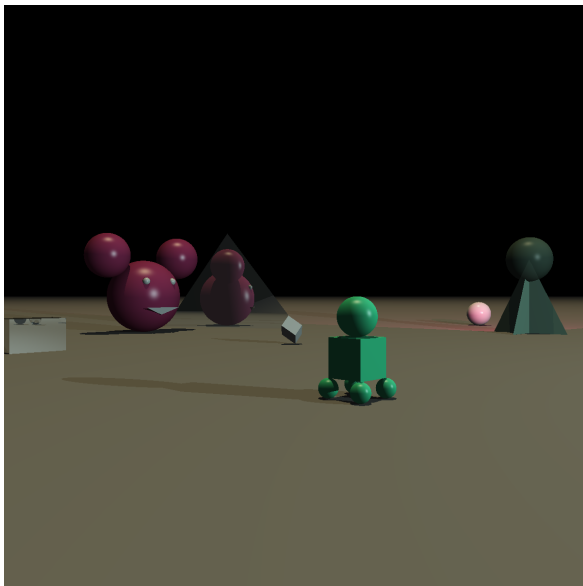
Interface utilisateur

Conception de l'interface utilisateur
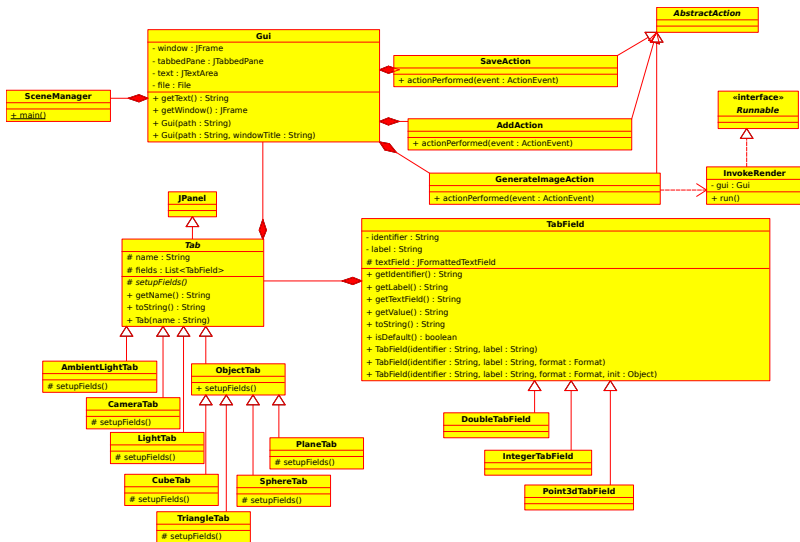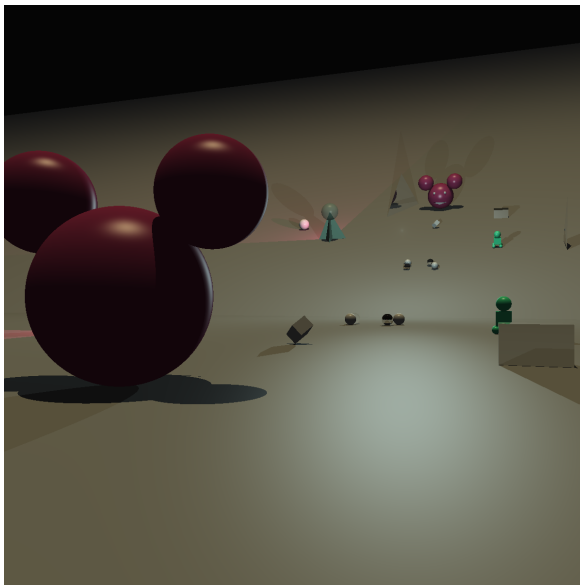
Conception du raytracer

FileReader
+ read(reader : java.io.Reader) : Scene

Light
- position : Point3d
- intensity : double[]
+ getIntensity(color : int) : double
+ getPosition() : Point3d

Scene
- objects : List<BasicObject>
- cameras : List<Camera>
- ambientLight : double[]
- lights : List<Light>
+ generateImages()
+ addObject(o : BasicObject)
+ rayColor(ray : Ray, depth : int) : double[]
- generateImage()

Camera
- eye : Point3d
- origin : Point3d
- abscissa : Vector3d
- ordonate : Vector3d
- widthPixels : int
- heigthPixels : int
+ getRay(x : int, y : int)
+ screenPoint()

-cameras

Ray
- origin : Point3d
- direction : Vector3d
+ getOrigin() : Point3d
+ getDirection() : Vector3d

Texture
+ brightness : double = 30.
+ k_specular : double = 1.
+ k_diffuse : double[] = {1., 1., 1.}
+ k_reflection : double = 0.
+ k_refraction : double[] = {0., 0., 0.}
+ refractive_index : double = 1.

#texture

BasicObject
+ computeColor(ray : Ray, scene : Scene, depth : int) : double[]
+ distance(ray : Ray) : double

Mesh
- objects : Set<BasicObjet>
+ setObjects(newObjects : Set<BasicObjet>)
+ getObjects() : Set<BasicObjet>
+ computeColor(ray : Ray, scene : Scene, depth : int) : double[]

Object
texture : Texture
+ computeColor(ray : Ray, scene : Scene, depth : int) : double[]
+ normal(ray : Ray) : Ray
+ isEntering(ray : Ray) : boolean

Cube

TriPointed
- P0 : Point3d
- P1 : Point3d
- P2 : Point3d
+ normal(ray : Ray) : Ray
+ distance(ray : Ray) : double
+ isEntering(ray : Ray) : boolean
# checkDontIntersect() : boolean

Sphere
- center : Point3d
- radius : double
+ distance(ray : Ray) : double
+ normal(ray : Ray) : Ray
+ isEntering(ray : Ray) : boolean

Triangle
checkDontIntersect() : boolean

Parallelogram
checkDontIntersect() : boolean

Plane
checkDontIntersect() : boolean

Exemple de rendu plus complexe

Exemple de rendu plus complexe, autre point de vue

Problèmes rencontrés

- Un objet peut s'intersecter avec lui-même.
- La réfraction, notamment dans une sphère, n'est pas représentative.
- Complexité en $O(x \times y \times n^2 \times m)$ avec n le nombre d'objets et m le nombre de sources lumineuses.

Améliorations

- Support de plusieurs formats de fichier.
- Possibilité de mettre une image sur un Parallélogramme.
- Création de vidéos.