PROJET L2 module INFO 3A 2020

Vous réaliserez, en monôme ou en binôme, un programme de lancer de rayons. Les enseignants seront plus exigeants pour les binômes que pour les monômes. Le langage de programmation est Python ; éventuellement, un autre langage fonctionnel peut être accepté (Caml, Ocaml) : demandez à votre encadrant ; cependant vous aurez moins de soutien de la part des enseignants.

Vous devrez ajouter quelque chose de spécifique à votre projet : visualiser des polyèdres, ou des fractales, ou des textures 3D, ou des procédures d'accélération, etc. Pour ces ajouts spécifiques, aucune correction ne sera fournie.

Le principe du lancer de rayons est le suivant. Un objet solide est :

- soit un objet simple, tel qu'une sphère, un ellipsoïde, un tore, un demi-plan, etc, représenté par une inéquation polynomiale P(x,y,z) <= 0 et une couleur. Par exemple, la sphère centrée à l'origine et de rayon 1 est décrite par : x^2 + y^2 + z^2 – 1 <= 0. La couleur est encodée par un entier c : b=(c % 256) est la composante bleue, v=((c // 256) % 256) est la composante verte, r=(c // 256 // 256) % 256 est la composante rouge. Donc c= 256*256*r + 256*v + b.

- soit la transformation (translation, homothétie ou affinité, rotation, symétrie, etc) d'un objet solide. Les transformations pourront être représentées par des matrices 4x4.

- soit la réunion, l'intersection ou la différence entre deux objets solides. Cette représentation arborescente est appelée CSG, Constructive Solid Geometry.

Pour chaque classe d'objet, il faudra programmer une méthode (ou une fonction) récursive calculant l'intersection entre un rayon (une demi-droite, représenté par son point origine o et un vecteur d donnant la direction du rayon : p(t) = (x(t), y(t), z(t))= o + t d est l'équation des points du rayon) et un objet de cette classe.

Il y a 3 phases dans le projet :

phase 1 : programmer l'intersection entre un rayon et un objet simple ; l'équation polynomiale de l'objet simple est décrite par un DAG (« directed acyclic graph »), représenté par un arbre avec partage éventuel de nœuds ; les feuilles de cet arbre sont soit des nombres ; soit des variables parmi t, x, y, z ; soit la somme de deux DAGs ; soit le produit de deux DAGs ; soit l'opposé d'un DAG. Dans ce DAG, vous substituerez x, y, z par leurs expressions en fonction de t : p(t) = (x(t), y(t), z(t))= o + t d avec o=(xo, yo, zo) un point 3D (xo, yo, zo sont des nombres), et d=(xd, yd, zd) est un vecteur 3D : xd, yd, zd sont des nombres. Cette substitution donne un DAG avec des nœuds +, *, opposé, et des feuilles qui sont soit des nombres, soit la variable t. Il faut transformer cet arbre en un polynôme uni-varié en la variable t. Pour cela, il faut associer à chaque type ou classe de nœuds une méthode ou une fonction qui convertit le nœud en polynôme uni-varié en t. En finalisation de la phase 1, générer un rayon pour chaque pixel (point) de l'image à calculer. Générer des images d'objets simples. Des équations d'objets simples (quadriques, tores, etc) sont fournies. Vous pourrez créer des animations.

phase 2 : programmer l'intersection entre un rayon et un objet de la classe transformation.

Phase 3 : programmer l'intersection entre un rayon et l'union, l'intersection, la différence entre 2 objets.

Phase 4 : programmer quelque chose de spécifique.

PHASE 1.

PHASE 1. Polynômes uni-variés en t. Fonctions : plus, mult, opp

Un polynôme uni-varié en la variable : t (qui est l'abscisse le long du rayon) est représenté par le tableau de ses coefficients (des nombres réels ou entiers relatifs en Python). Le coefficient du terme de degré i est stocké dans la i ème case du tableau. [1, 2, 3] représente le polynôme (1 + 2 t + 3 t^2).

Le but est d'écrire la somme (plus), la différence (moins) et le produit (mult) de 2 polynômes. Ne perdez pas votre temps à redéfinir ou surcharger les opérations +, -, *. Vous le ferez avec les DAGs représentant les expressions mathématiques des objets simples.

Les fonctions suivantes peuvent être utiles :

coeff(p, i) rend p[i] s'il existe, 0 sinon.

sigma(i1, i2, f) calcule la somme f(i1) + f(i1+1)+ ... f(i2).

subvec(v, i, j) rend le vecteur dans v contenant v[i]...v[j].

Exemples :
>>> plus([1, 2, 3], [4, 5, 6, 7]) # (1 + 2t + 3t^2) + (4 + 5t +6t^2 + 7t^3)
[5, 7, 9, 7]
>>> mult([1, 2, 3], [4, 5, 6, 7]) # (1 + 2t + 3t^2) * (4 + 5t +6t^2 + 7t^3)
[4, 13, 28, 34, 32, 21]

L'opposé d'un polynôme est trivial. De même pour le polynôme d'un nombre, ou de la variable t. Vous programmerez une fonction topolent pour convertir un DAG ne contenant que t comme variable en le polynôme uni-varié en t correspondant. Les DAG sont précisés ci-dessous.

Il faut ensuite calculer les racines de ce polynôme uni-varié en t (les valeurs de t pour lesquelles le polynôme s'annule). Pour cela, le polynôme est converti dans la base de Bernstein. Les précisions sont données ci-dessous.

PHASE 1. DAG et expressions mathématiques : eval, evalsymb, topolent, derivee

Partir de ces sources Python :

```
class M(object):
    def __init__(self):
        "   "
    def __add__( self, b):
        return Plus( self, b)
    def __mul__( self, b):
        return Mult( self, b)
    def __sub__( self, b):
        return Plus( self, Opp(b))
```

```
class Opp( M):
    def __init__(self, a):
        self.a=a

class Plus(M):
    def __init__(self, a, b):
        self.a=a
        self.b=b

class Mult(M):
    def __init__(self, a, b):
        self.a=a
        self.b=b

class Nb(M):
    def __init__(self, n):
        self.nb=n

class Var(M):
    def __init__(self, nom):
        self.nom=nom

x=Var(« x »)
y=Var(« y »)
z=Var(« z »)
cercle=x*x+y*y-Nb(1)
```

Ecrire une méthode eval pour chaque classe Var, Nb, Plus, Mult, Opp. Le paramètre est un dictionnaire qui donne les valeurs flottantes (ou entières) des variables : cercle.eval({ 'x' : 1, 'y': -1}) rend la valeur de l'expression cercle pour x= 1 et y= -1. Pour savoir si « x » est dans le dictionnaire dico, il faut dire : if « x » in dico.

Ecrire une méthode evalsymb pour ces classes. Cette fois, les variables sont remplacées par d'autres expressions (de type Var, Nb, Plus, Mult, Opp). Pour écrire un programme de lancer de rayon, il suffira de remplacer x par x=xo + xd * t, un polynôme uni-varié en t, où xo et xd sont des nombres. De même pour y et z.

En supposant que la seule variable est t , écrire une méthode topolent() qui convertit une expression en polynôme en t. Réutilisez les méthodes plus, mult, opp des polynômes.

Ecrire une méthode derivee(nom) qui calcule l'expression de la dérivée par rapport à la variable nom. Cela sera nécessaire pour calculer le vecteur normal à l'objet solide simple F(x,y,z)<=0 en un point d'intersection I=(x, y, z : 3 nombres) entre le rayon et cet objet. La normale est le vecteur (nx, ny, nz) où nx est la dérivée de F par rapport à la variable x en I, ny est la dérivée de F par rapport à la variable y en I, nz est la dérivée de F par rapport à la variable z en I.

PHASE 1. Recherche de racines entre 0 et 1 d'un polynôme uni-varié dans la base de Bernstein

Ecrire l'algorithme de Casteljau : à partir d'un vecteur v0 représentant un polynôme uni-varié en t, cette méthode calcule le vecteur des milieux v1[i]= (v0(i]+v0[i+1])/2., puis v2, etc, jusqu'au vecteur vide. Par exemple (format Python):
v0= [0, 4, 16]
v1= [2, 10] est le vecteur des milieux (ou des moyennes) de v0
v2= [6] est le vecteur des milieux (ou des moyennes) de v1
Il faut retourner la paire ([0, 2, 6], [6, 10, 16]).

Ecrire mintab(v) et maxtab(v) qui calculent le plus petit et le plus grand élément dans un tableau.

Ecrire la fonction solve(epsilon, tab, t1, t2, solutions) ; epsilon est un petit nombre (il peut même être nul) comme 1e-8; tab est le vecteur des coefficients dans la base de Bernstein. t1 et t2 sont les bornes de l'intervalle considéré : au début t1=0.0 et t2=1.0 ; solutions est la liste des solutions déjà connues (None initialement). Si 0 est hors de [mintab(tab), maxtab(tab)], rendre solutions. Sinon : si t2 – t1 < epsilon, alors rendre cons((t1+t2)/2., solutions). Sinon : calculer (tab1, tab2)= casteljau(tab), et chercher récursivement les racines de tab1 dans [t1, (t1+t2)/2.], et de tab2 dans [(t1+t2)/2., t2]. Tester avec tab=[1., -2., -2., 1.] : vous devez trouver deux racines (leur somme vaut 1).

PHASE 1. Conversion de la base canonique (1, t, t^2...) à la base de Bernstein (B0(t), B1(t), B2(t)...)

Ecrire l'algorithme de conversion de la base canonique (1, t, t^2, .. t^d) vers la base de Bernstein de degré d : (B0(t), … Bd(t)).

t^k est la somme pour i de 0 à d de : C(k, i) / C(k,d) * Bi(t), où :

C(k, i) est le nombre de façons de choisir k éléments parmi i. C(k, i) est appelé « coefficient binomial » (pour trouver la définition sur internet). Pour tester votre programme :

```
>>> tobernstein( [10., 0., 0.])          # 10 + 0 t + 0 t^2
[10.0, 10.0, 10.0]
>>> tobernstein( [0., 10., 0.])          # 0 + 10 t + 0 t^2
[0.0, 5.0, 10.0]
>>> tobernstein( [0., 0., 10.])          # 0 + 0 t + 10 t^2
[0.0, 0.0, 10.0]
```

Ecrire une fonction pour calculer les racines comprises entre 0 et 1 d'un polynôme donné dans la base canonique (la fonction solve calcule les racines pour un polynôme donné dans la base de Bernstein). Il faut donc convertir le polynôme de la base canonique à la base de Bernstein.

Chercher les racines plus grandes que (ou égales à) 1 d'un polynôme donné dans la base canonique. Exemple : pour trouver les racines >= 1 de p(t)= 1 + 2 t + 3 t^2 + 4 t^3 = 0, poser T=1/t, donc t=1/T, (si t >= 1, alors T est dans (0, 1]). D'où :

1 + 2 / T + 3 / (T^2) + 4 / (T^3) = 0

Multiplier par T^3 donne :
q(T) = T^3 p(1/T) = T^3 + 2 T^2 + 3 T + 4 = 0

Les coefficients de ce polynôme q(T) sont les mêmes que ceux du polynôme p(t), mais en ordre inverse. Calculer les racines entre 0 et 1 de q(T), et en déduire les racines t=1/T du polynôme p(t). Attention : il faut inverser l'ordre des racines.

```
>>> racines( [15., -8., 1. ])
(3.000000011175871, (5.0000000186264515, None))

>>> polca=[15., -8., 1. ]
>>> n= len( polca)
>>> poly= [polca[n-i-1] for i in range( 0, n, 1)]
>>> poly
[1.0, -8.0, 15.0]
>>> p= tobernstein( poly)
>>> p
[1.0, -3.0, 8.0]
>>> roots=solve( 1e-8, p, 0., 1., None)
>>> roots
(0.19999999925494194, (0.3333333320915699, None))
>>> mymap( (lambda x: 1./x), roots)
(5.0000000186264515, (3.000000011175871, None))
```

PHASE 1. Finalisation.

En finalisation de la phase 1, vous visualiserez des objets simples. Partez du fichier fourni pour programmer un lancer de rayons sur des surfaces implicites, ie des objets définis par une inéquation algébrique f(x,y,z) <= 0.

$ cat rayon.py

```
# coding=utf−8
# :set expandtab
# :set tabstop=4
import math
import random
from expr import *
import os
import sys
from PIL import Image, ImageDraw, ImageFont

def interpole( x1, y1, x2, y2, x) :
    # x=x1 -> y=y1
    # x=x2 -> y=y2
    x1, y1, x2, y2, x= float(x1), float(y1), float( x2), float(y2), float(x)
    return (x-x2)/(x1-x2)*y1 + (x-x1)/(x2-x1)*y2

def normalize3( (a,b,c)):
        (a,b,c)=(float(a),float(b),float(c))
        n=math.sqrt(a*a+b*b+c*c)
```

```
        if 0.==n:
                return (0.,0.,0.)
        else:
                return (a/n, b/n, c/n)

class Obj(object):
   def __init__( self):
      " "


class Rayon( object):
        def __init__( self, source, dir):
                self.source=source
                self.dir=dir

class Camera( object):
        def __init__( self, o, ox, oy, oz, hsizeworld, hsizewin, soleil):
                self.o=o
                self.ox= ox #vers la droite du spectateur
                self.oy= oy #regard du spectateur
                self.oz= oz #vertical du spectateur
                self.hsizeworld=hsizeworld
                self.hsizewin=hsizewin
                self.soleil = normalize3( soleil)
                self.background=(100,100, 255)
                self.nom= "img.png"
        def generate_ray( self, x, z):
                (x0, y0, z0)= self.o
                kx = interpole( 0., 0., self.hsizewin, self.hsizeworld, float(x))
                kz = interpole( 0., 0., self.hsizewin, self.hsizeworld, float(z))
                return Rayon( (x0 + kx*self.ox[0] + kz*self.oz[0],
                     y0 + kx*self.ox[1] + kz*self.oz[1],
                     z0 + kx*self.ox[2] + kz*self.oz[2]),
                     self.oy)

def topolent( e):
        return e.topolent()

class Prim( Obj):
        def __init__( self, fonc_xyz, color):
                self.fonc=fonc_xyz
                self.color=color
        def intersection( self, rayon):
                dico = { "x": Nb(rayon.source[0]) + Nb(rayon.dir[0])*Var("t"),
                  "y": Nb(rayon.source[1]) + Nb(rayon.dir[1])*Var("t"),
                  "z": Nb(rayon.source[2]) + Nb(rayon.dir[2])*Var("t")}
                expression_en_t=self.fonc.evalsymb( dico)
                pol_t = topolent( expression_en_t)
                return racines( pol_t)
        def normale( self, (x,y,z)):
```

```
                fx=self.fonc.derivee("x")
                fy=self.fonc.derivee("y")
                fz=self.fonc.derivee("z")
                dico={"x":x, "y":y, "z":z}
                (a,b,c)= ( fx.eval( dico), fy.eval( dico), fz.eval( dico))
                return normalize3( (a, b, c) )

def pscal3( (x1,y1,z1), (x2,y2,z2)):
        return x1*x2 + y1*y2 + z1*z2

def clamp( mi, ma, v):
        return min( ma, max( mi, v))

def raycasting( cam, objet):
        img=Image.new("RGB", (2*cam.hsizewin+1, 2*cam.hsizewin+1), (255,255,255))
        for xpix in range( -cam.hsizewin, cam.hsizewin+1, 1):
                for zpix in range( -cam.hsizewin, cam.hsizewin+1, 1):
                        rayon= cam.generate_ray( xpix, zpix)
                        roots=objet.intersection( rayon)
                        if None==roots:
                                (r,v,b)= cam.background
                        else:
                                t= hd(roots) # roots[0] #c'est le 1er element (un t) de la pire (tete, queue)
                                pt=(xo,yo,zo)= (rayon.source[0]+ t*rayon.dir[0],
                               rayon.source[1]+ t*rayon.dir[1],
                               rayon.source[2]+ t*rayon.dir[2])
                                (a,b,c)=normalize3( objet.normale(pt))
                                (rr,vv,bb)=objet.color
                                (rr,vv,bb)= (float(rr), float(vv), float(bb))
                                ps=pscal3( (a,b,c), cam.soleil)
                                if ps < -1. or 1 < ps:
                                        print("PS="+str(ps))
                                        ps = clamp( -1., 1., ps)
                                coef= interpole( -1., 0.5, 1., 1., ps)
                                r=coef*rr
                                v=coef*vv
                                b=coef*bb
                                (r,v,b) = (int(r), int(v), int(b))
                        img.putpixel(      (xpix+cam.hsizewin,      2*cam.hsizewin-(zpix+cam.hsizewin)),
(r,v,b))
        img.show()
        img.save( cam.nom)

oeil=(0.001,-4.,0.003)
droite=  (1.,0.,0.)
regard=  (0.,1.,0.)
vertical=(0.,0.,1.)
#le repere local est tel que regard=oy, vertical=oz, droite=ox, o=oeil
#ox, oy,oz orthogonaux et normés
```

```
camera=Camera( oeil, droite, regard, vertical, 1.5, 100, normalize3((0., -1., 2.)))

def boule( (cx,cy,cz), r):
        x=Var("x")
    y=Var("y")
    z=Var("z")
        return (x-Nb(cx))*(x-Nb(cx)) + (y-Nb(cy))*(y-Nb(cy)) + (z-Nb(cz))*(z-Nb(cz)) - Nb(r*r)

def tore( r, R):
        x=Var("x")
        y=Var("y")
        z=Var("z")
        tmp=x*x+y*y+z*z+Nb(R*R-r*r)
        return tmp*tmp- Nb(4.*R*R)*(x*x+z*z)

def steiner2():
        x=Var("x")
    y=Var("y")
    z=Var("z")
        return (x * x * y * y - x * x * z * z + y * y * z * z - x * y * z)

def steiner4():
        x=Var("x")
    y=Var("y")
    z=Var("z")
        return y * y - Nb( 2.) * x * y * y - x * z * z + x * x * y * y + x * x * z * z - z * z * z * z

camera.nom="boule.png"
raycasting( camera, Prim( boule( (0., 2., -0.5), 1.), ((255,255,255))))
camera.nom="tore.png"
raycasting( camera, Prim( tore(0.45, 1.), (255,200, 255)))

def hyperboloide_2nappes():
        x=Var("x")
        y=Var("y")
        z=Var("z")
        return Nb(0.) - (z * z - (x * x + y * y + Nb(0.1)))

def hyperboloide_1nappe():
        x=Var("x")
    y=Var("y")
    z=Var("z")
    return Nb(0.)-(z * z - (x * x + y * y - Nb(0.1)))

def roman():
        x=Var("x")
    y=Var("y")
    z=Var("z")
        return ( x * x * y * y + x * x * z * z + y * y * z * z - Nb(2.) *  x * y * z)
```

```
camera.hsizeworld=10.
raycasting( camera, Prim( steiner2(), (255,200, 255)))

camera.hsizeworld=1.5
camera.nom="roman.png"
raycasting( camera, Prim( roman(), (255,200, 255)))
camera.nom="hyper1.png"
raycasting( camera, Prim( hyperboloide_1nappe(), (255,200, 255)))
camera.nom="hyper2.png"
raycasting( camera, Prim( hyperboloide_2nappes(), (255,200, 255)))
camera.nom="steiner2.png"
camera.nom="steiner4.png"
raycasting( camera, Prim( steiner4(), (255,200, 255)))
```