Production d'images scientifiques

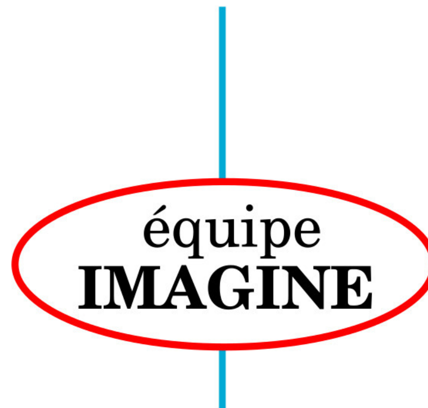
MMI - Lyon
29/01/2016

Damien Rohmer
[damien.rohmer@inria.fr]

Brève présentation

Damien Rohmer
[damien.rohmer@inria.fr]

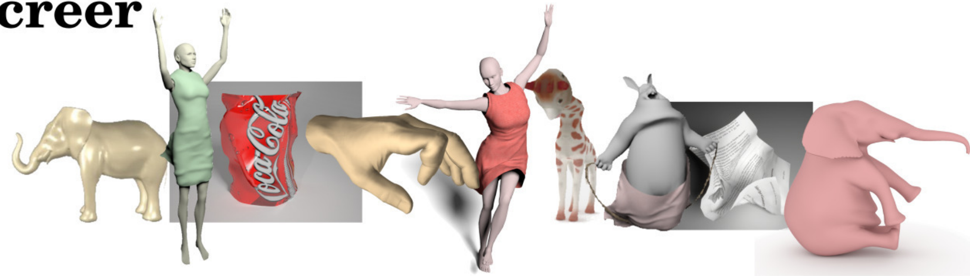
Enseigne à CPE Lyon (informatique/images)
Recherche en informatique graphique (~synthèse d'images)



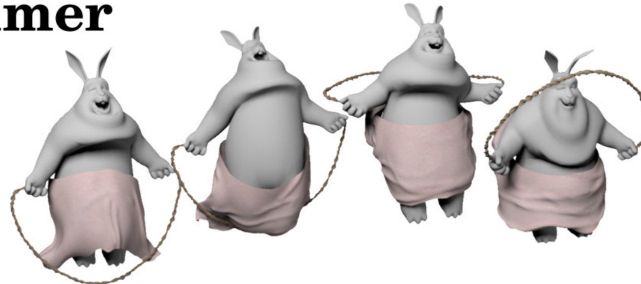
Thèmes de recherche

Modélisation virtuelle 3D

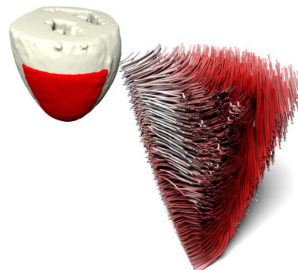
- Modéliser pour **créer**

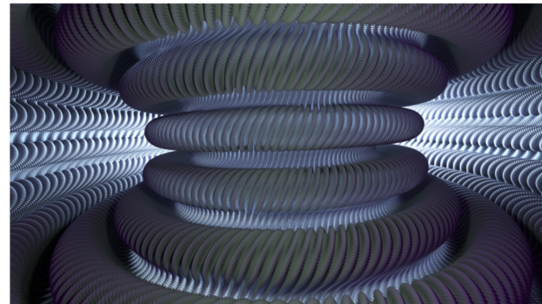


- Modéliser pour **animer**



- Modéliser pour **visualiser**

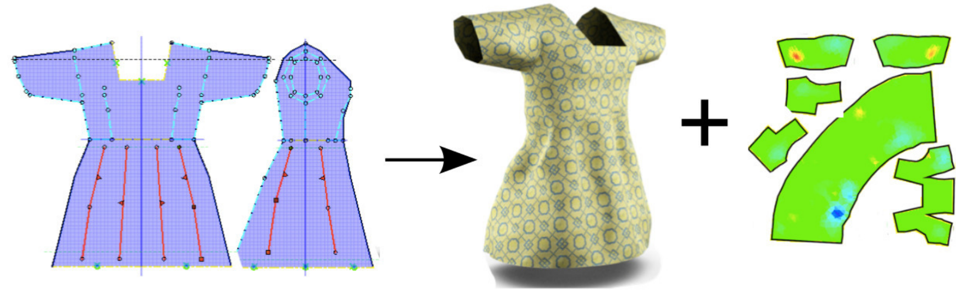


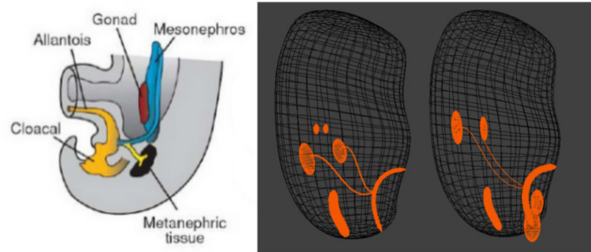


Modéliser pour créer

Design

Sketching surface développables



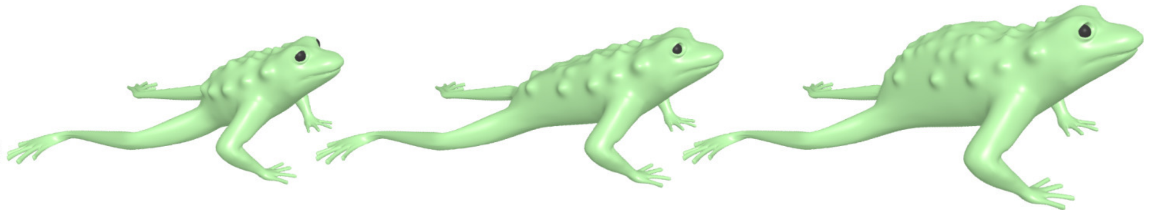


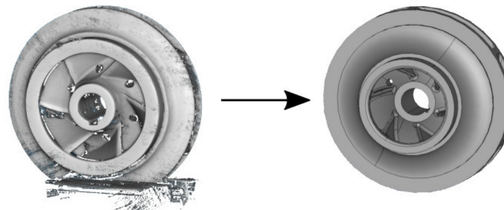
Médical

Modéliser croissance embryonnaire

Graphique

Déformation formes avec détails





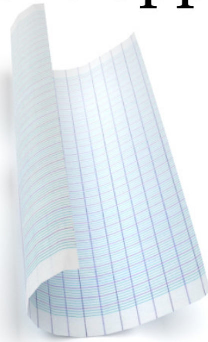
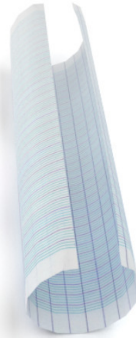
CAO

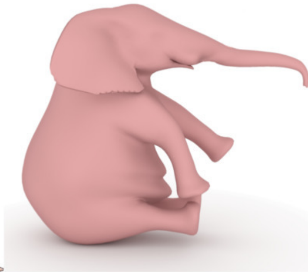
Générer modèles CAO fonctionnels

Modéliser pour animer

Surfaces développables

Papier froissé



Personnages

Amélioration du skinning

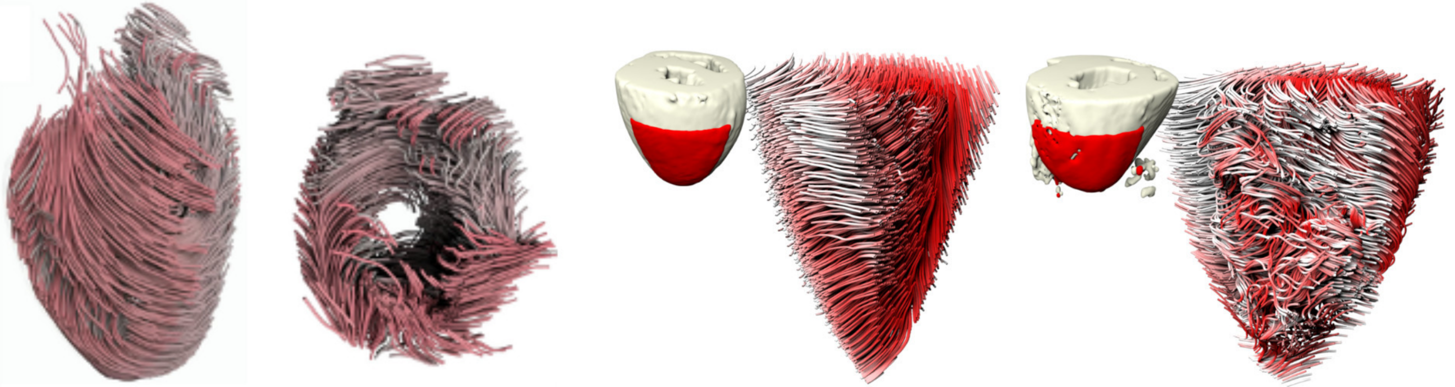
Vêtements

Ajout de plis géométriques

Modéliser pour visualiser

Médicale

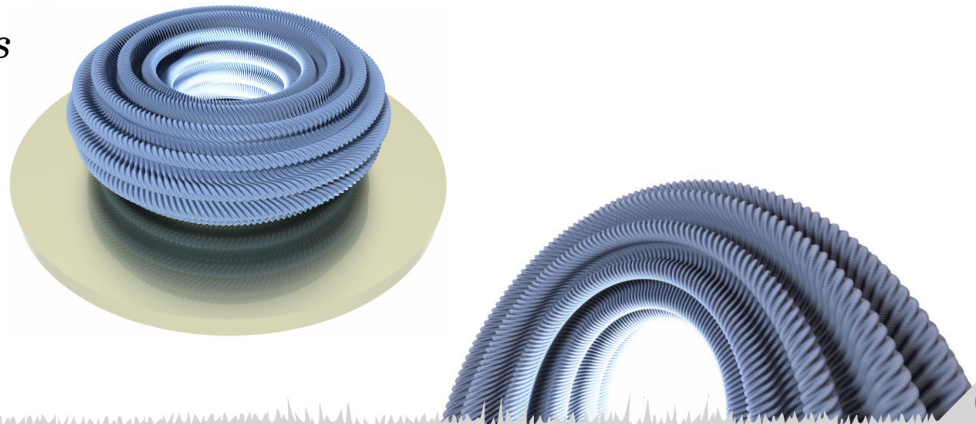
Fibres cardiaques



Mathématiques

Surfaces paradoxales

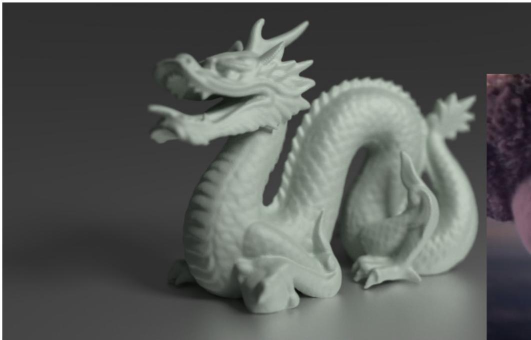
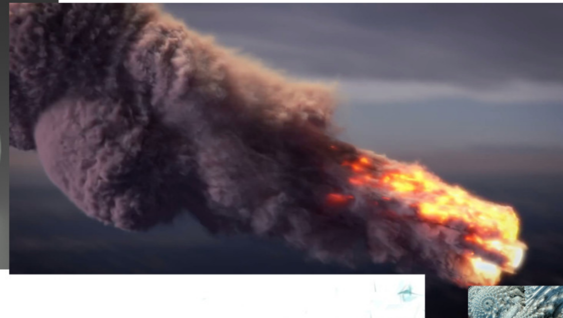
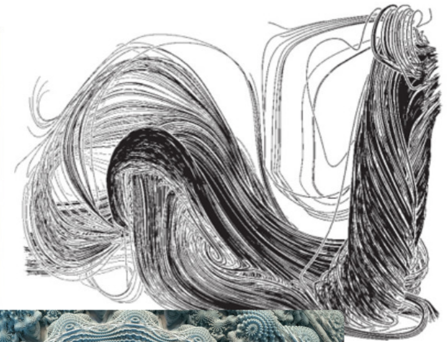
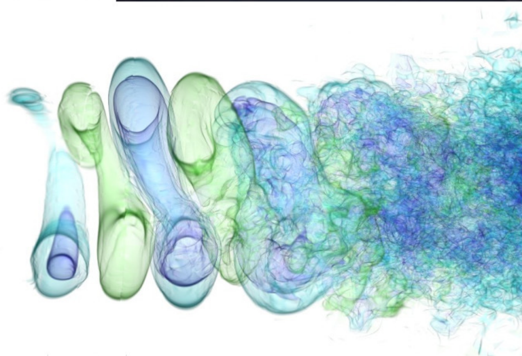
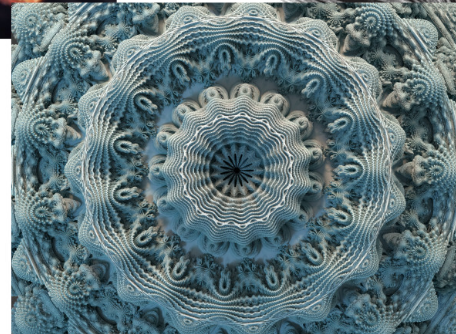
[Hévéa]



Que souhaite-on visualiser ?

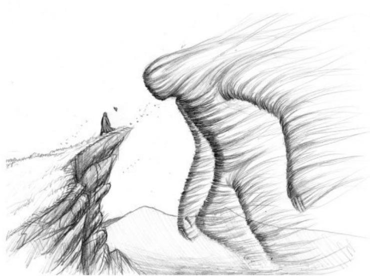
- Des surfaces ?
- Des volumes ?
- Des courbes ?
- Des fractales ?
- ...

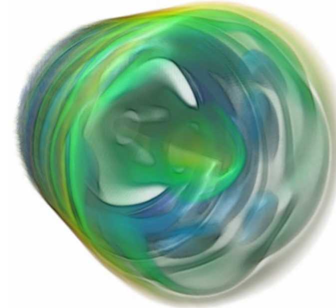
Techniques différentes

Quelles contraintes ajoute t-on ?

- Données de très grandes tailles (ne rentrent pas en RAM)
- Données bruitées
- Données non visualisable (tenseurs, abstraites, etc).
- ...





Cas le plus standard ?

On va considérer:

- Pas de contraintes de tailles (toutes les données en mémoire)
- Pas de bruit
- etc

Objectif: **Représentation d'objets réels**

~~Fractals, champs volumiques/vecteurs, etc ...~~

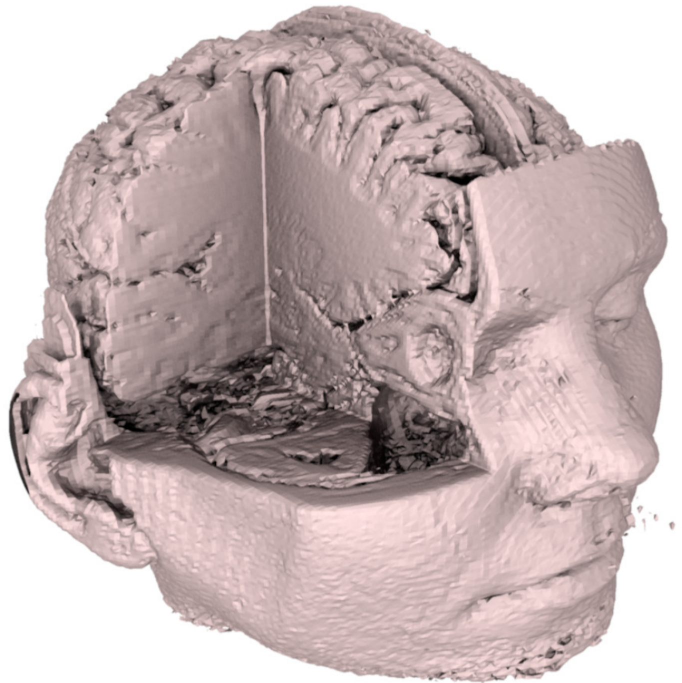
pas cette fois

L'informatique graphique en 5min

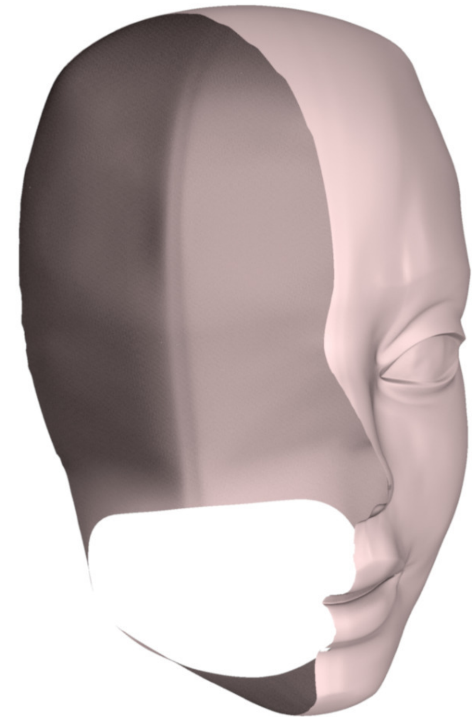
Que manipule t-on ?

Modélisation de surface

Volume



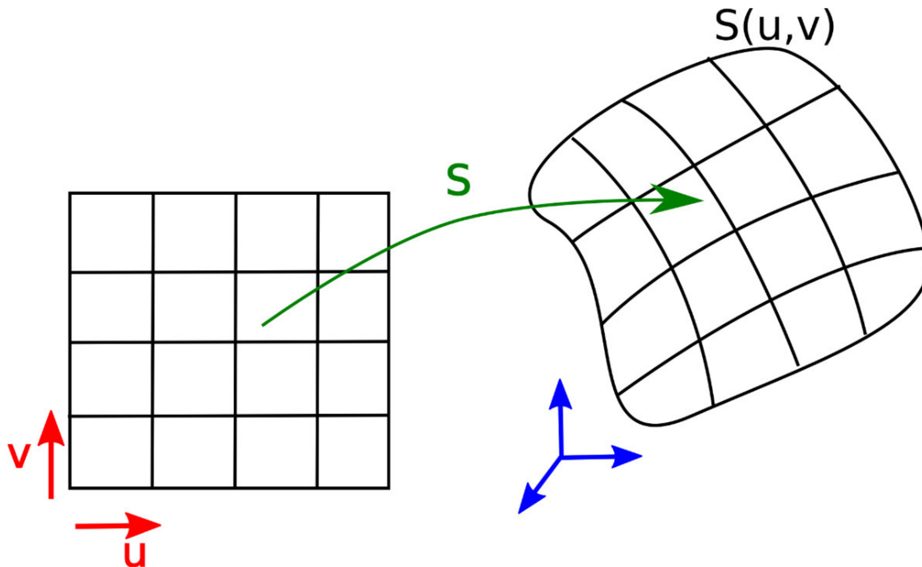
Surface



Encodage des surfaces

Représentation paramétrique ?

$$S : \begin{cases} \mathcal{D} \subset \mathbb{R}^2 & \rightarrow \mathbb{R}^3 \\ (u, v) & \mapsto S(u, v) = (S_x(u, v), S_y(u, v), S_z(u, v)) \end{cases}$$

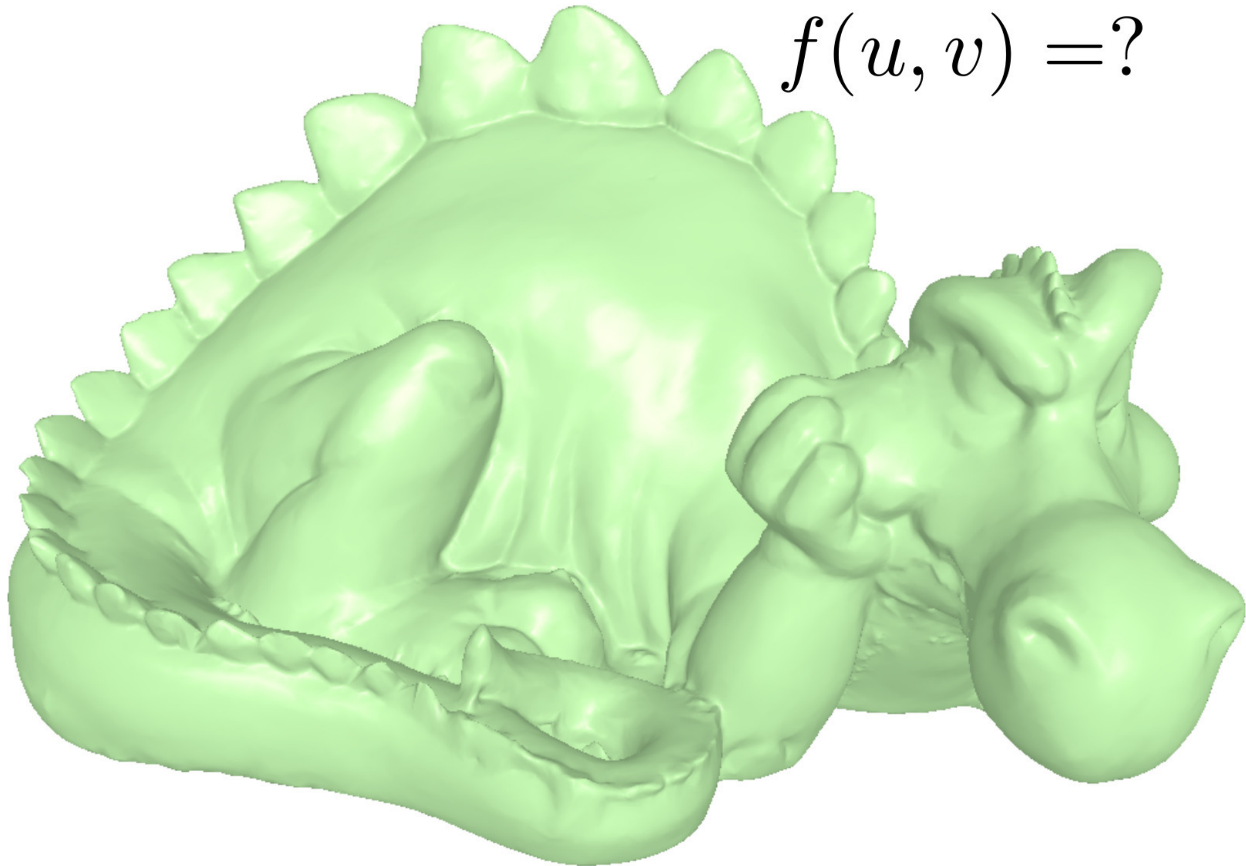


S: fonction/mapping (+information que la surface)

Encodage des surfaces

Quelle est la fonction ?

$$f(u, v) = ?$$



Encodage des surfaces

On définit la fonction par morceaux $S = \bigcup_i S_i$

On considère les morceaux les plus simples possibles

Encodage des surfaces

On définit la fonction par morceaux $S = \bigcup_i S_i$

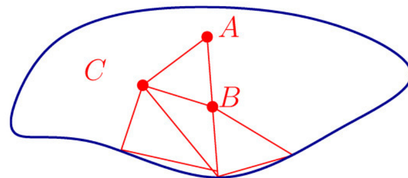
On considère les morceaux les plus simples possibles

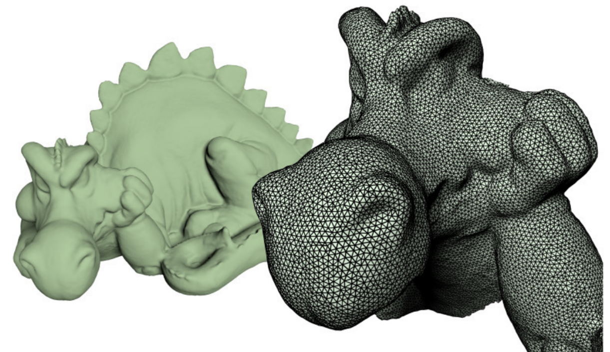
→ Fonction linéaire
=> Surface planes

Surface = Ensemble de triangles

$$S_i : \begin{cases} \mathcal{D} \subset \mathbb{R}^2 & \to \mathbb{R}^3 \\ (u, v) & \mapsto S_i(u, v) = u\,\overrightarrow{AB} + v\,\overrightarrow{AC} + \overrightarrow{OA} \end{cases}$$

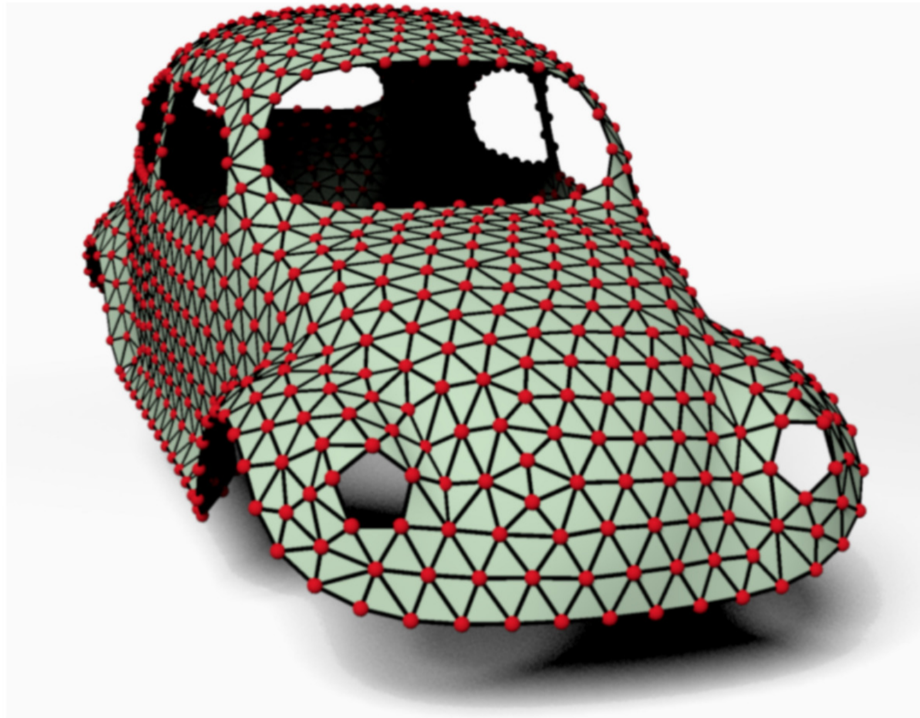
$$\mathcal{D} : (u, v) \in [0, 1]^2, 0 \leq u + v \leq 1$$





Maillage

Ensemble de triangles / quadrangles



- sommet *(vertex)*

$$\mathcal{V} = (v_1, \ldots, v_N)$$

- arête *(edge)*

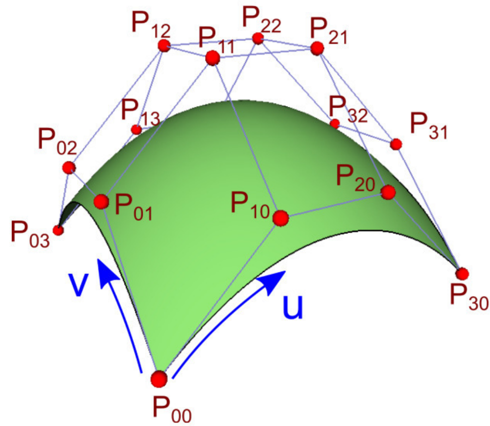
$$\mathcal{E} = (e_1, \ldots, e_{N_e}) \in (\mathcal{V}^2)^{N_e}$$

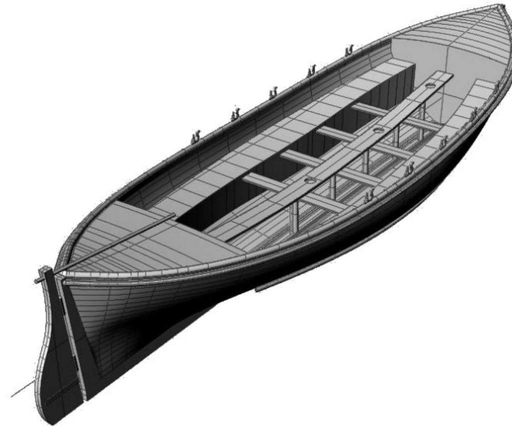
- face

$$\mathcal{F} = (f_1, \ldots, f_{N_f}) \in (\mathcal{V}^3)^{N_f}$$

Surfaces paramétriques

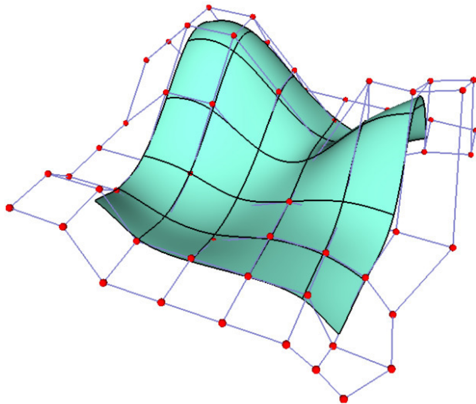
Ensemble de patchs Spline/NURBS

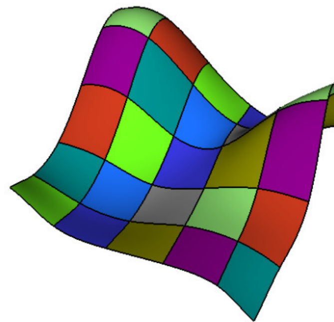


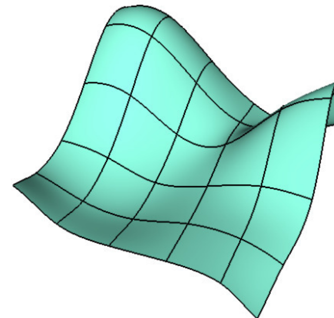


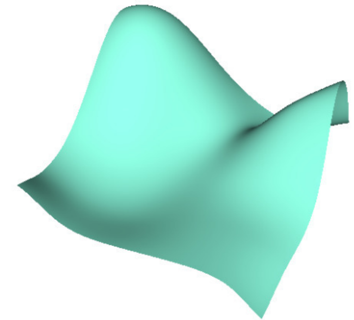
+ Lisses

- Libertés



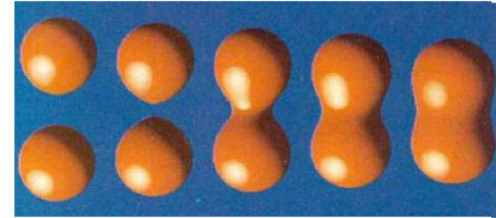






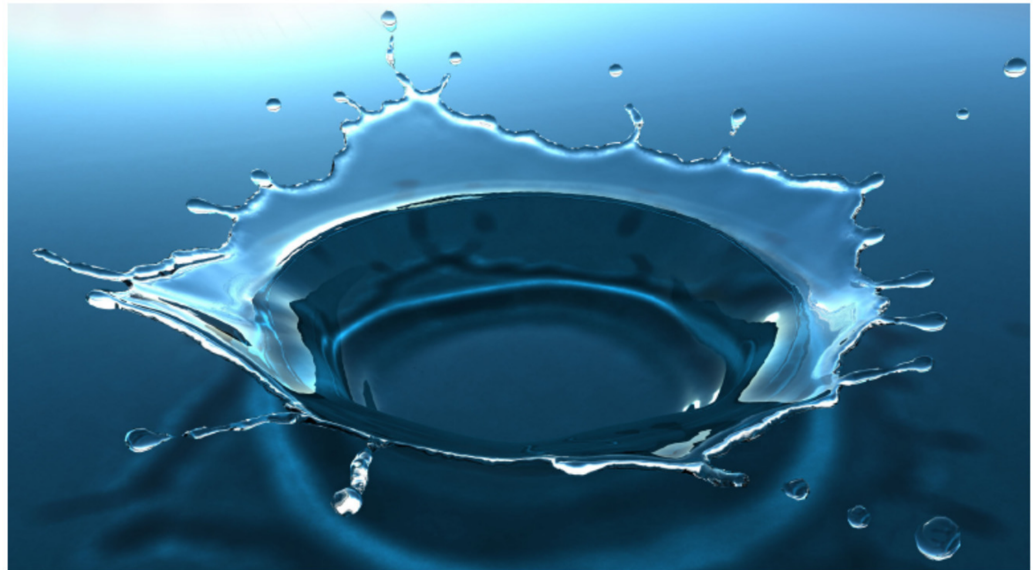
Surfaces implicites

$$S = \{\mathbf{p} \in \mathbb{R}^3 \mid \psi(\mathbf{p}) = 0\}$$
$$S = \psi^{-1}(0)$$



[Wyvill, McPheeters, Wyvill, The Visual Computer, 1986]

+ Changement de topologie
- Visualisation
- Manipulation
- Voisinage



[Thuereu, Wojtan, Gross, Turk, SIGGRAPH 2010]

Les méthodes de rendus (en 5min)

Comment représente t-on une surface 3D sur un écran 2D

Les approches classiques

1- Rendu projectif

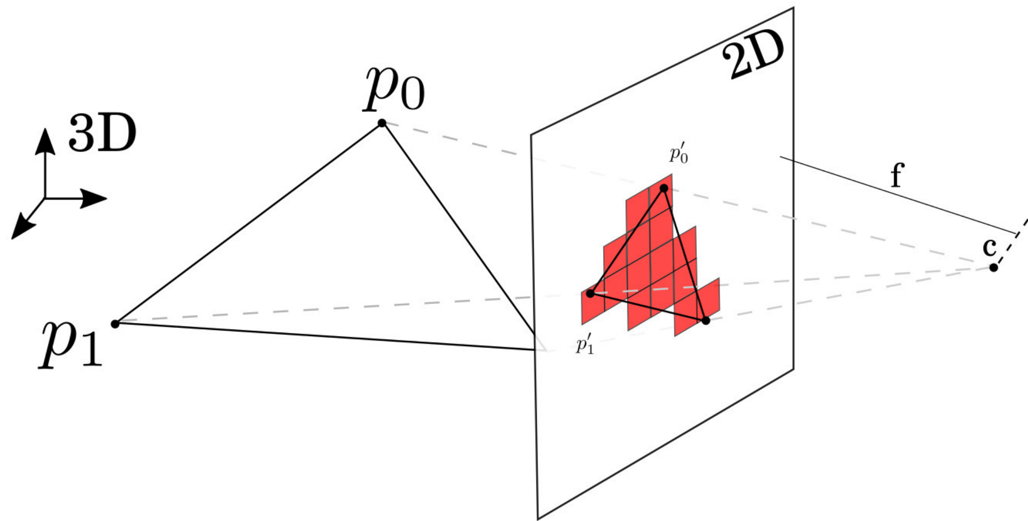
2- Lancé de rayons (ray tracing)

3D -> 2D

+ Calcul d'illumination

(couleur final, matériaux, aspect, etc)

Rendu projectif



On projette les triangles sur les plans 2D

(projection de 3 sommets 3D vers 2D, puis "remplissage" des triangles 2D)

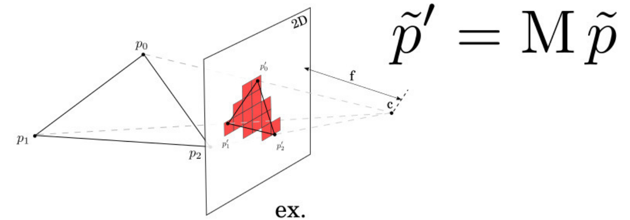
(application d'une matrice de projection 4x4)

=> On fait cela pour tous les triangles

On affiche les pixels qui sont devant les autres (ZBuffer)

Rendu projectif

Méthode des cartes graphiques
jeux vidéos, etc



Avantage

- Très rapide
Implémentée sur les cartes

~1-10 millions de triangles
affichés en temps réel aisé.

Inconvénients

- Pas d'ombres portées

- Gestion devant/derrière à gérer

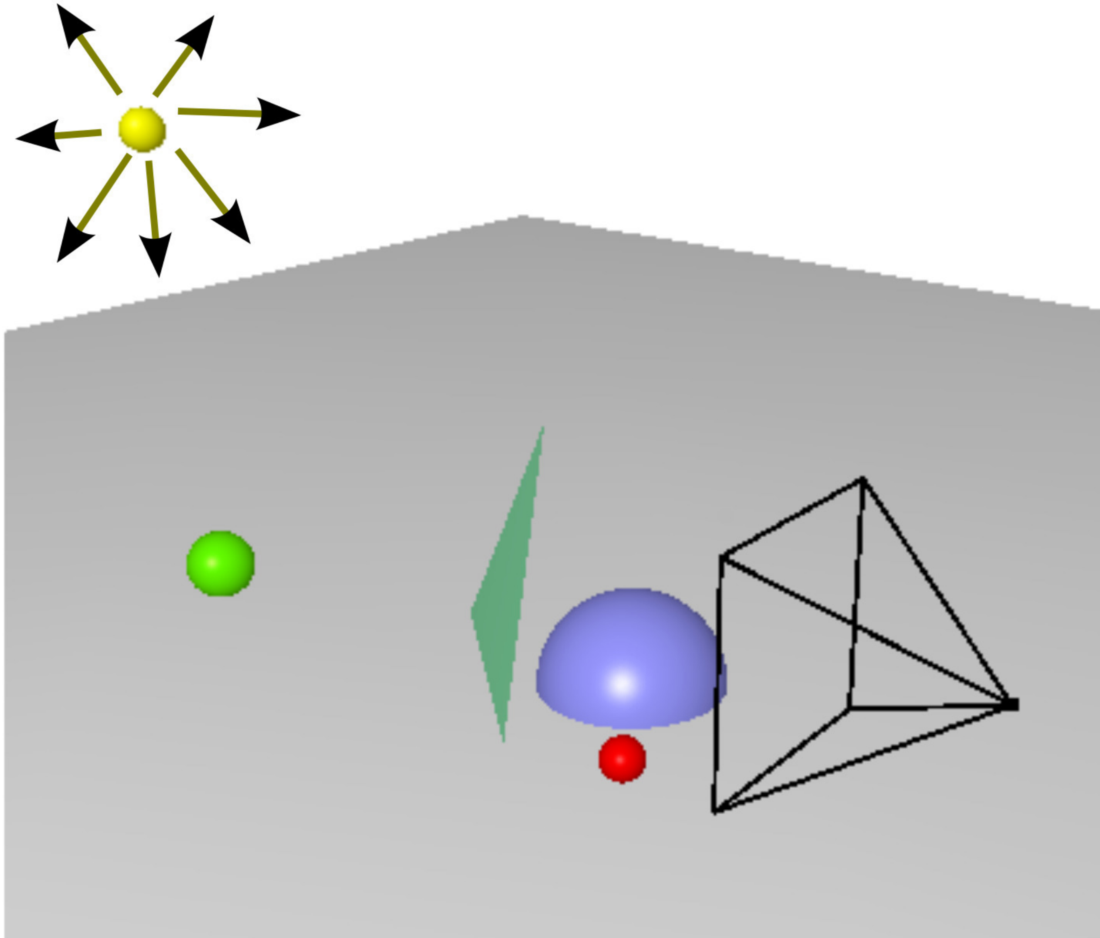
- Que des triangles

raison pour laquelle
on dois tout trianguler

- Méthode non physique
(tout effet doit être mimé par
triche/hack)

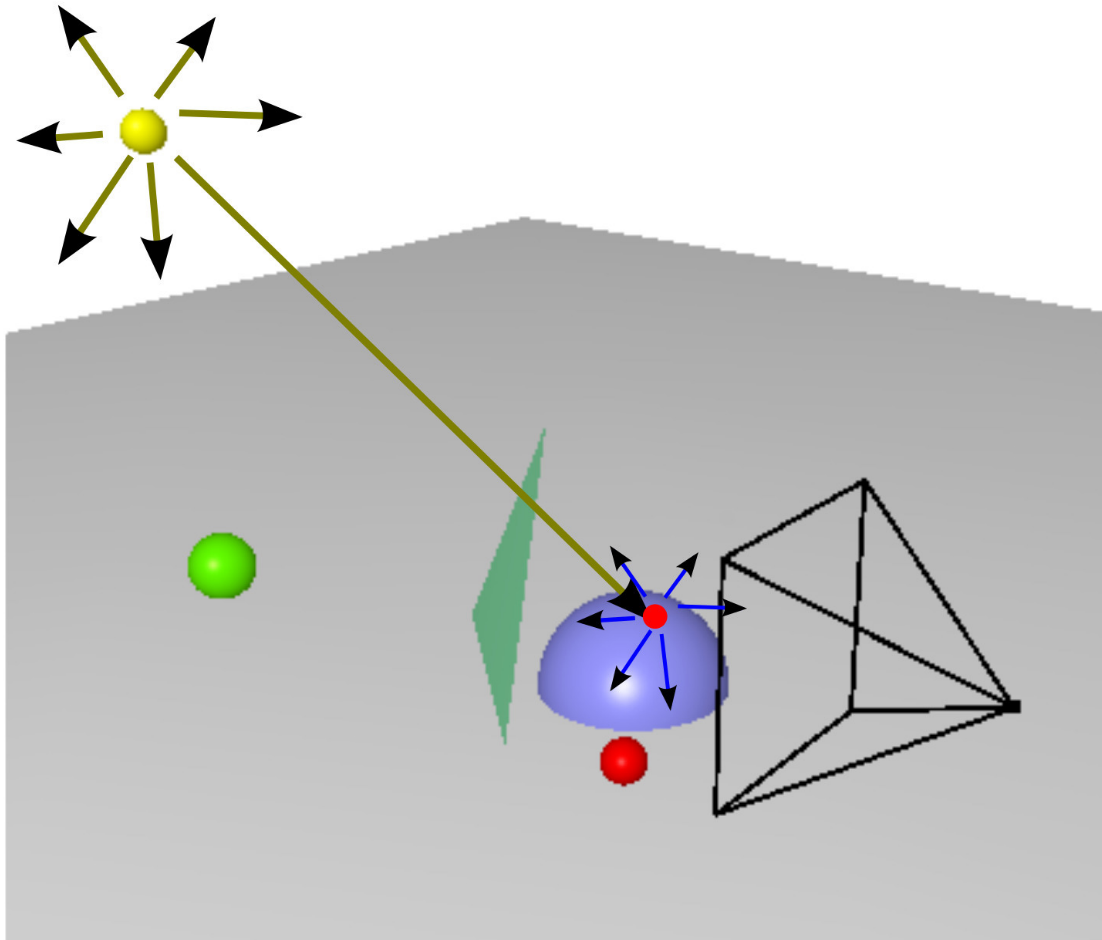
Rendu par lancé de rayons

Observons ce qui se passe dans la réalité



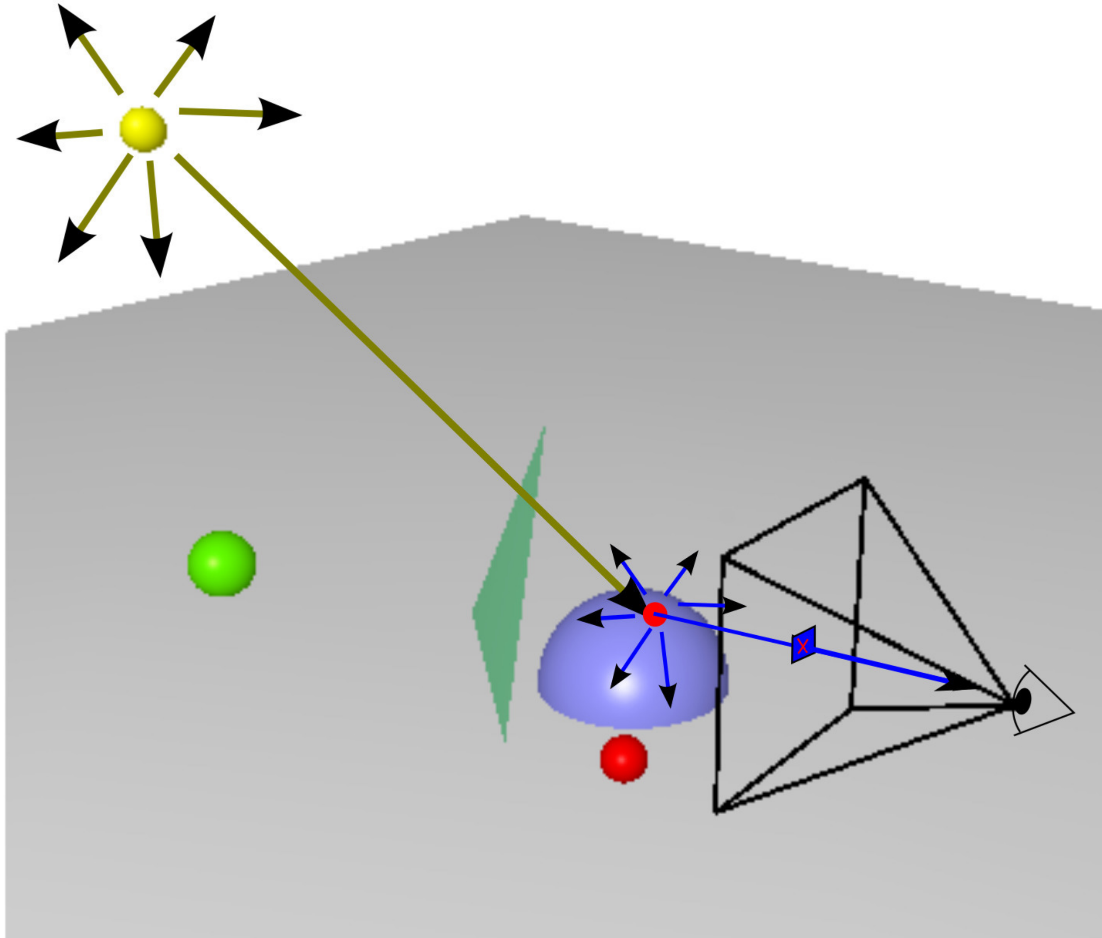
Rendu par lancé de rayons

Observons ce qui se passe dans la réalité



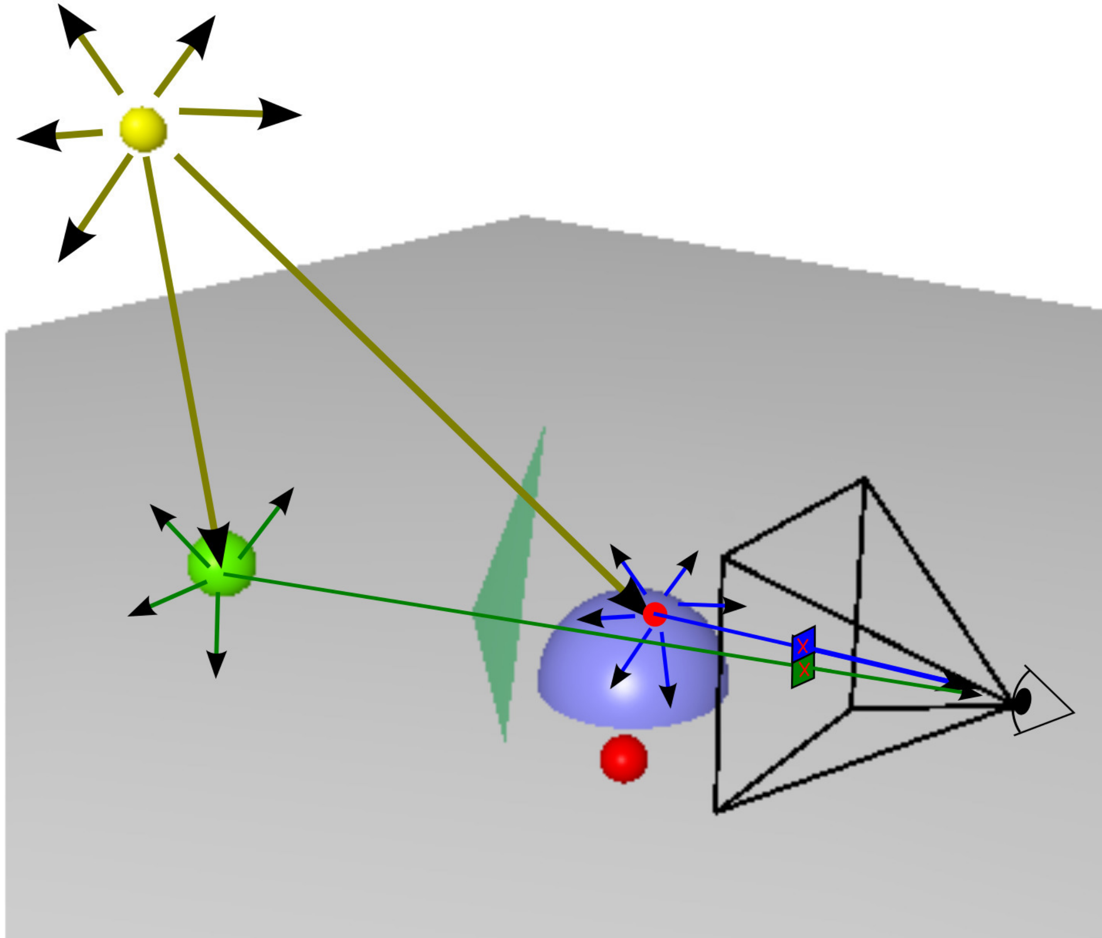
Rendu par lancé de rayons

Observons ce qui se passe dans la réalité



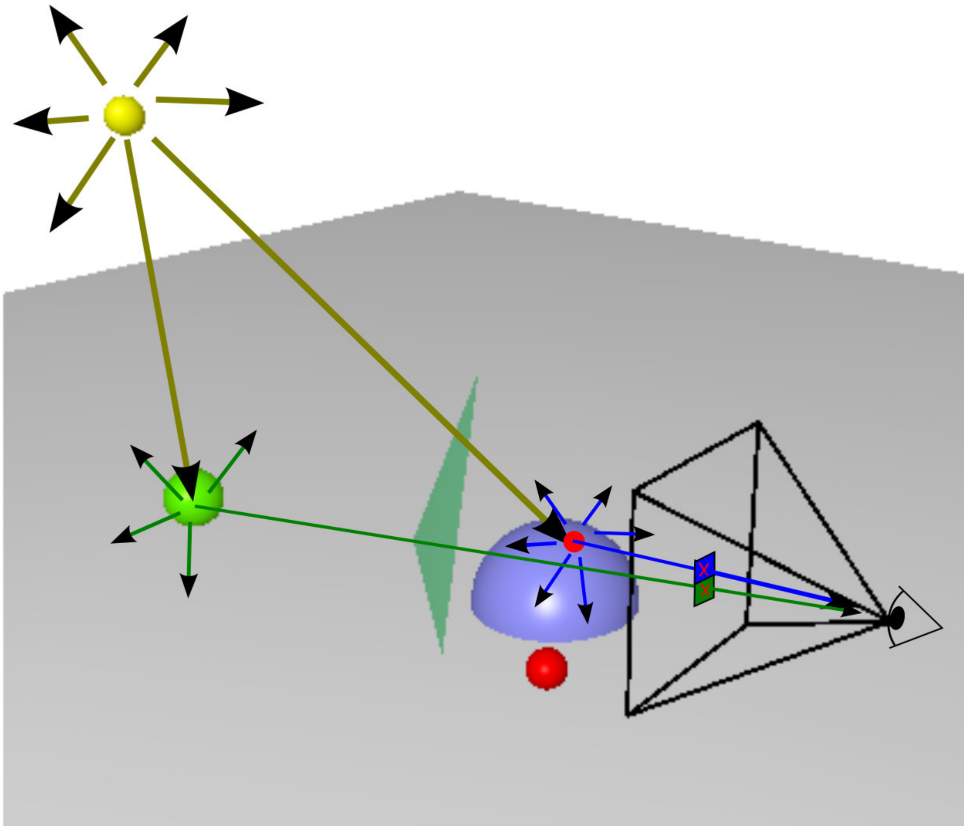
Rendu par lancé de rayons

Observons ce qui se passe dans la réalité



Rendu par lancé de rayons

Observons ce qui se passe dans la réalité

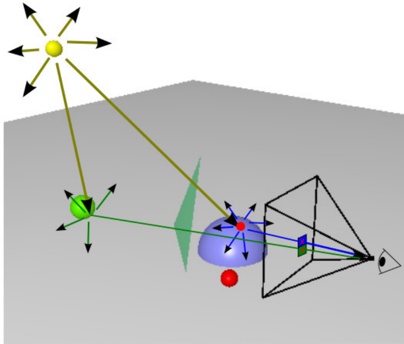


First algorithm

```
For all light sources
   For all direction d1
      Throw_ray(d1)


Throw_ray(d)
|  If d intersect any object
|     Save Color
|     For all direction d2
|         Throw_ray(d2)
|
|  If d intersect screnn
|     Set current Color to pixel
```

Rendu par lancé de rayons

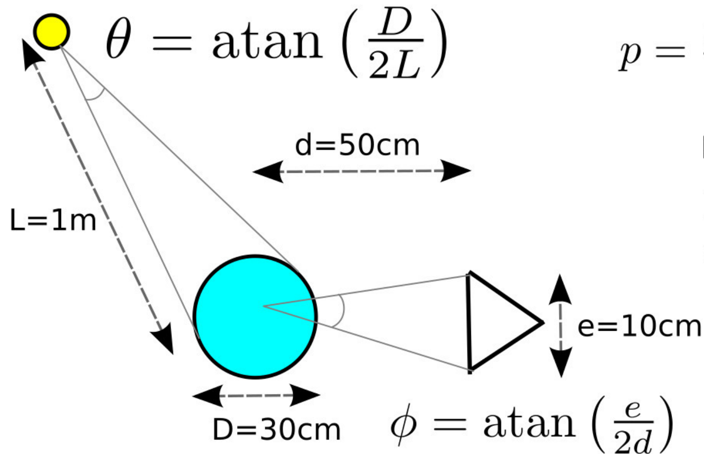


+ Simple
+ Précis physiquement

- Complexité algorithmique

→ Recursion infinie

ex.



$$p = \frac{2\pi(1-\cos(\theta))}{4\pi} \times \frac{2\pi(1-\cos(\phi))}{4\pi}$$

p touche l'objet=0.0015%

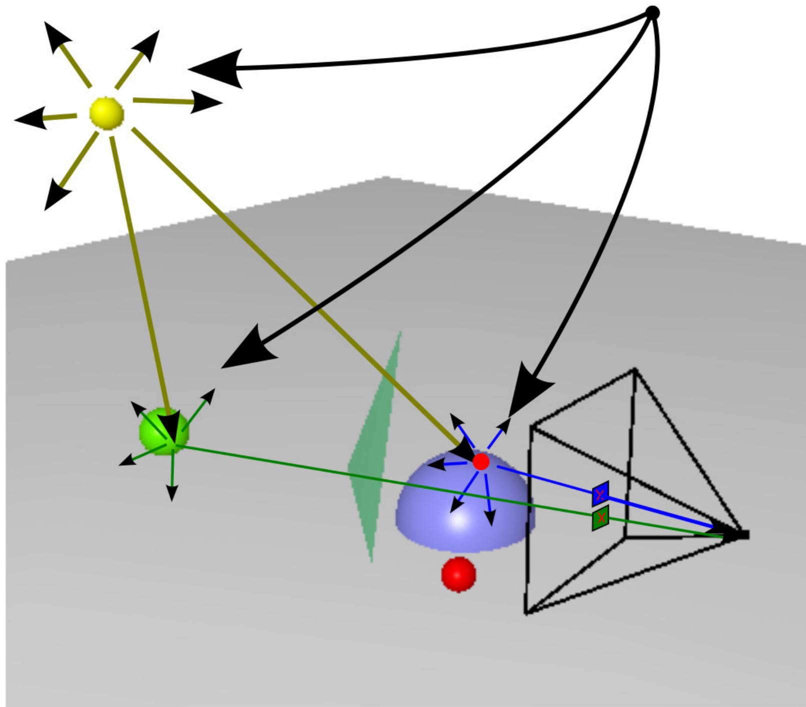
p touche le pixel=0.0000000007%

envoie 130 000 000 000 rays

Exemple:
0.1 ms/rayons :
150 jours/image

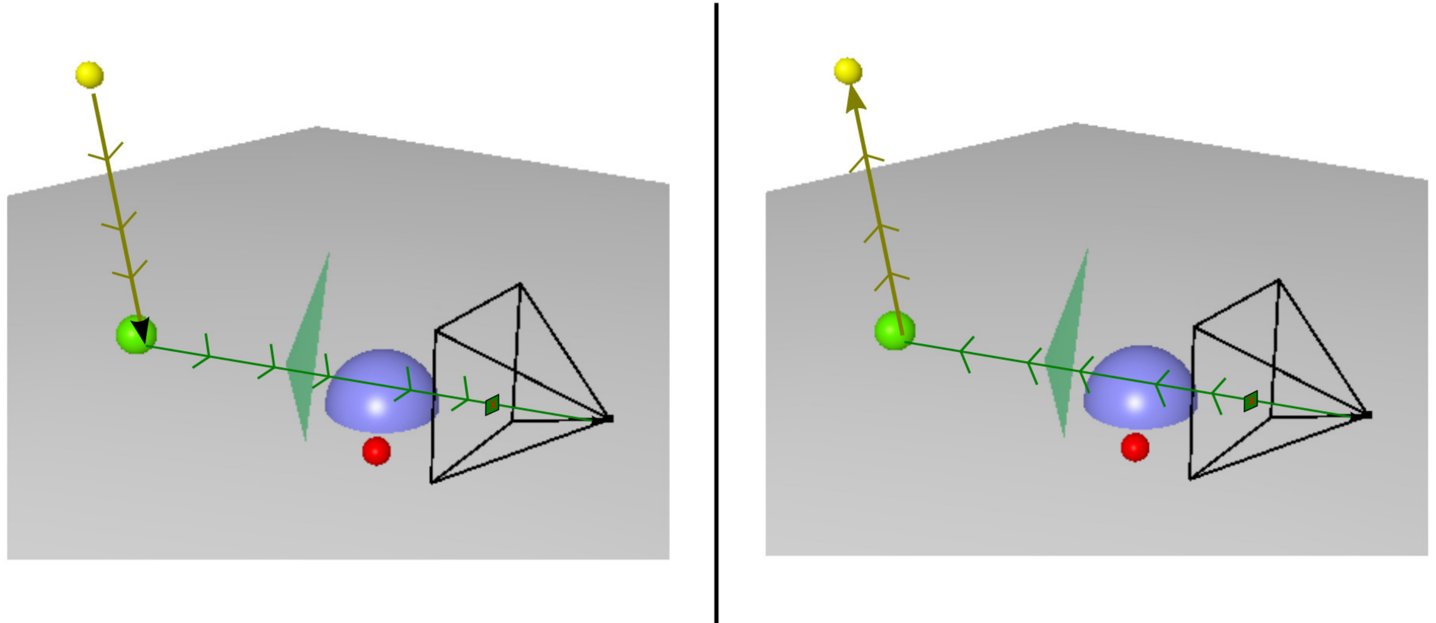
Acceleration du rendu

Probleme : plein de rayons inutiles

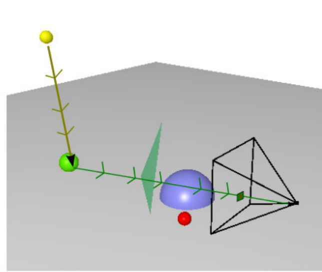


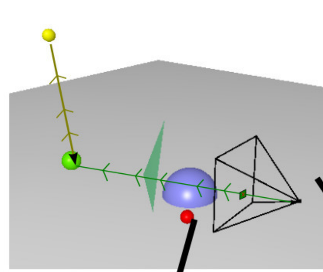
Acceleration du rendu

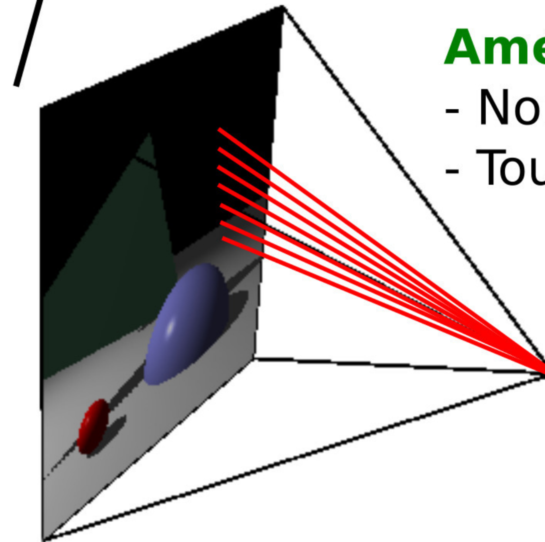
Principe de Fermat:
Même chemin lumineux dans les deux sens



Acceleration du rendu

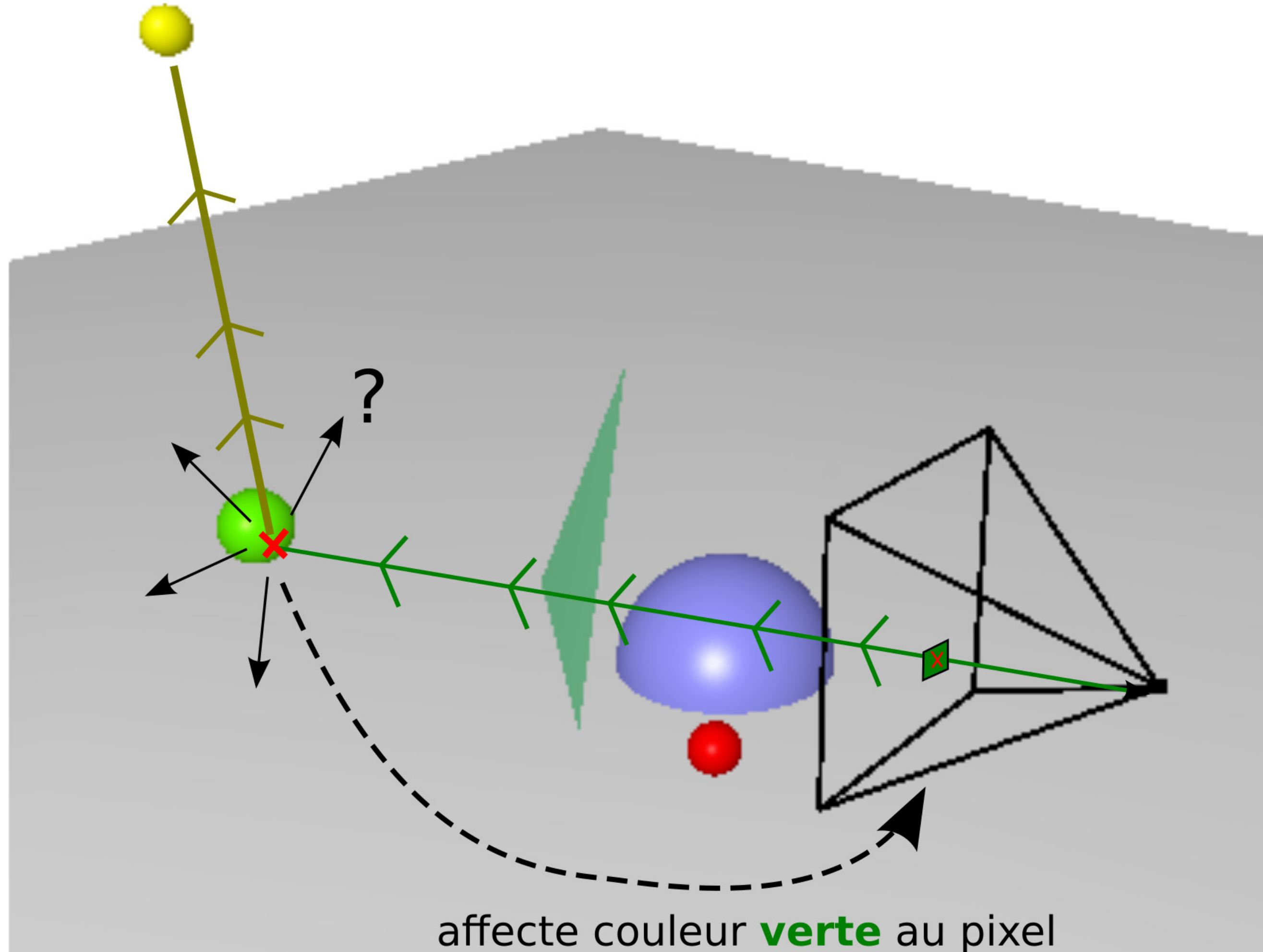




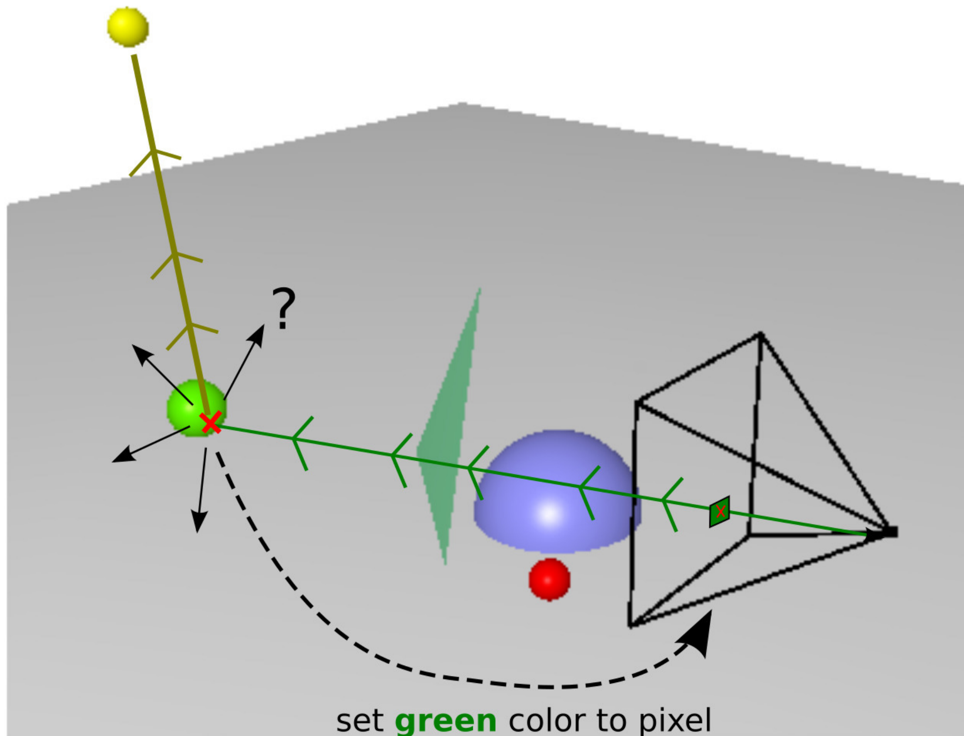


Amélioration:
- Nombre fini de rayons
- Tous utiles

Acceleration du rendu



Acceleration du rendu



1/ Principe de Fermat

+ Physique OK

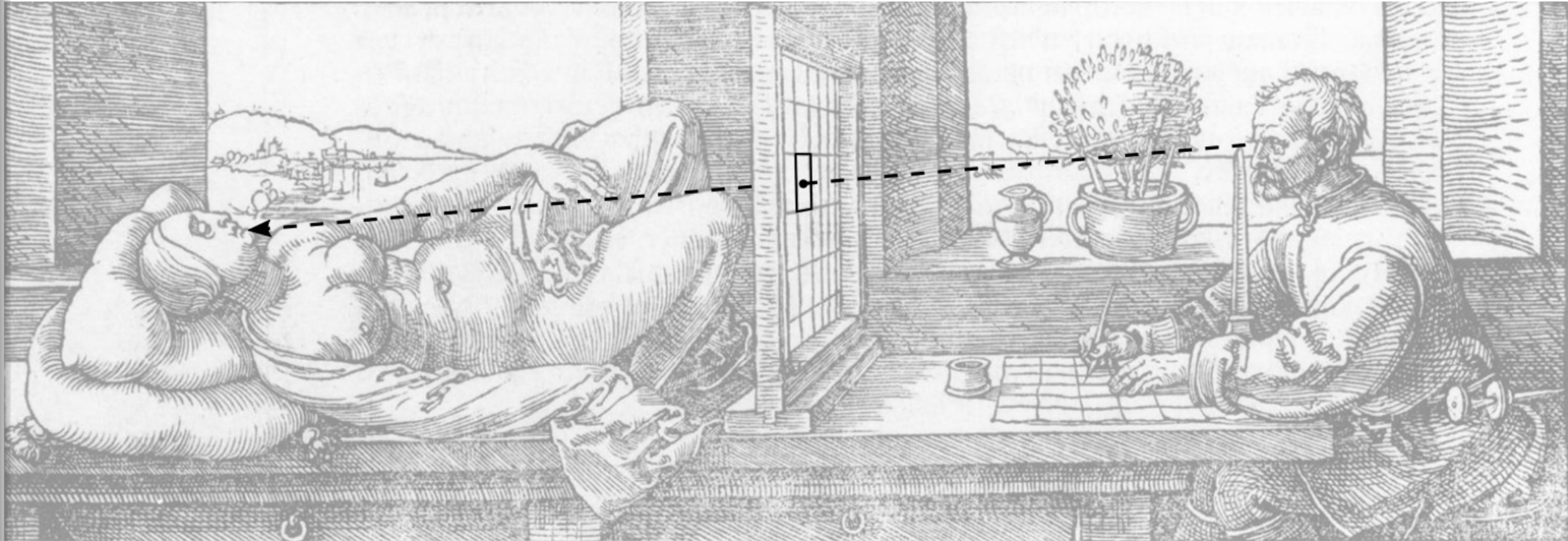
2/ Pas de diffusion secondaire

- perte de physique

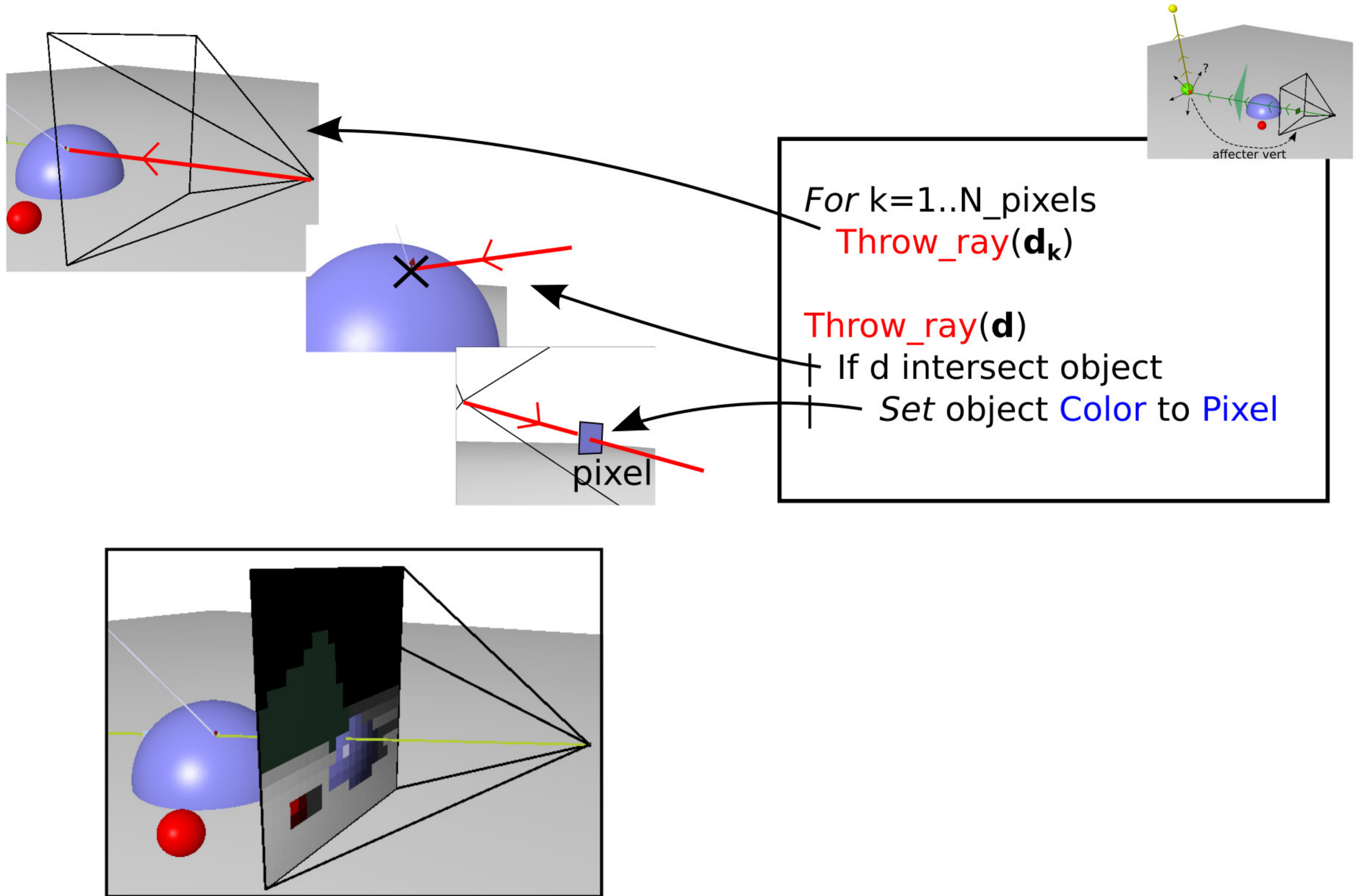
quantification:
~2 000 000 rays

0.1 ms/rays :
3 min/pic
VS 150 days/pic

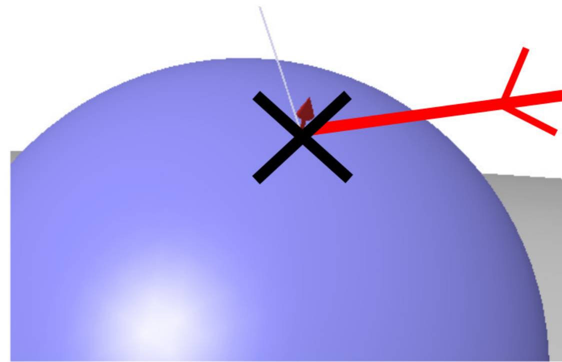
Draughtsman

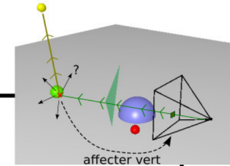


Acceleration du rendu



Acceleration du rendu





For k=1..N_pixels
  Throw_ray($\mathbf{d_k}$)

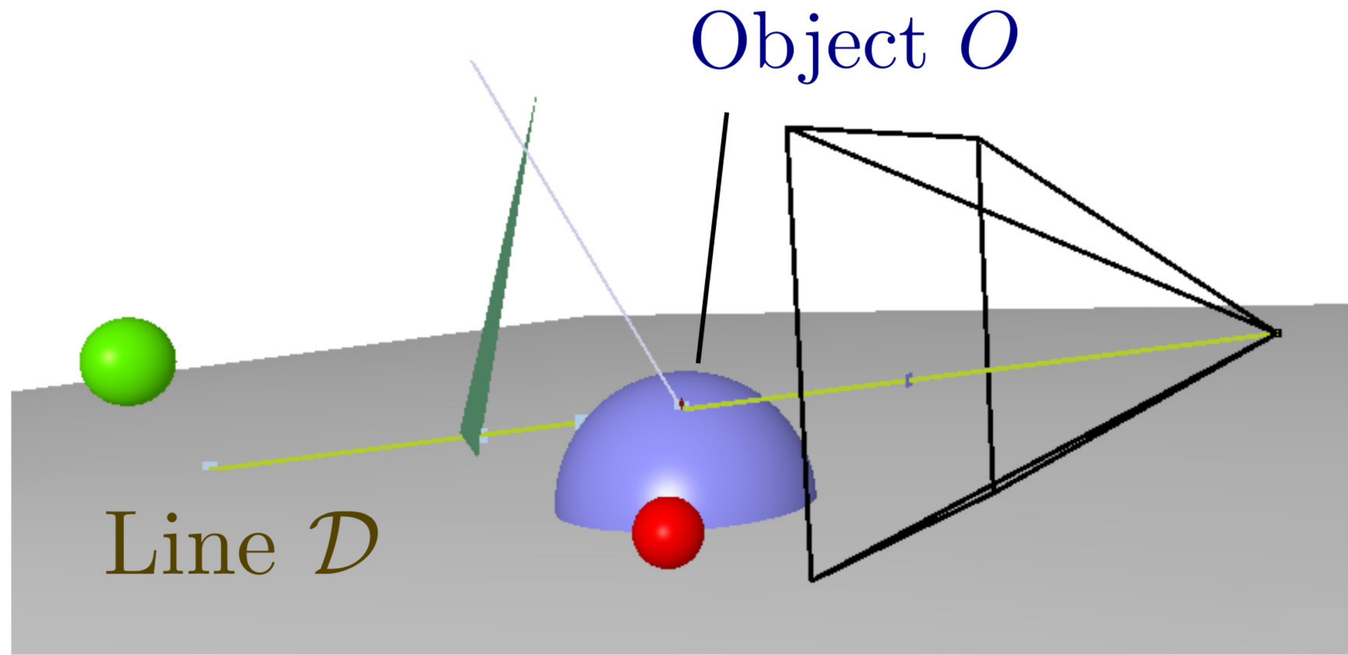
Throw_ray(**d**)
| If d intersect object
| *Set* object Color to Pixel

Complexité
du calcul

Formalisation

For all lines $\mathcal{D}$

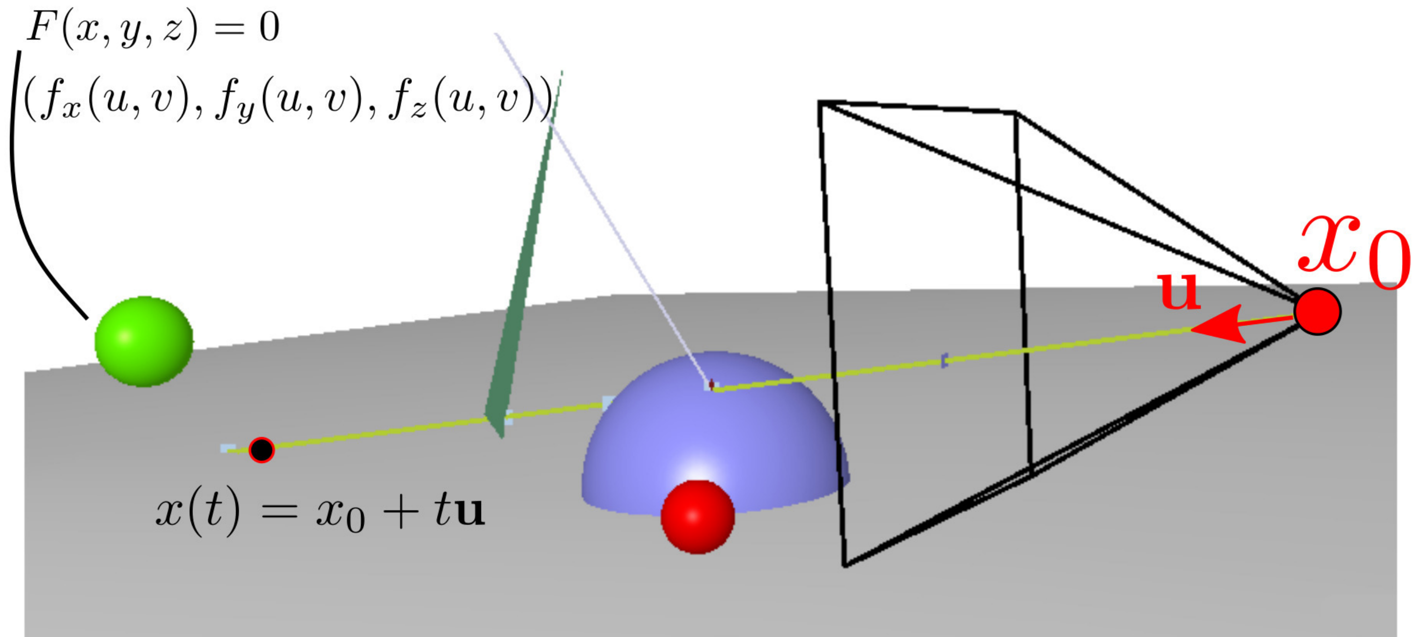
  Compute $D \cap O$



Formalisation

For all lines $\mathcal{D}$

Compute $D \cap O$



Formalisation

1: We search

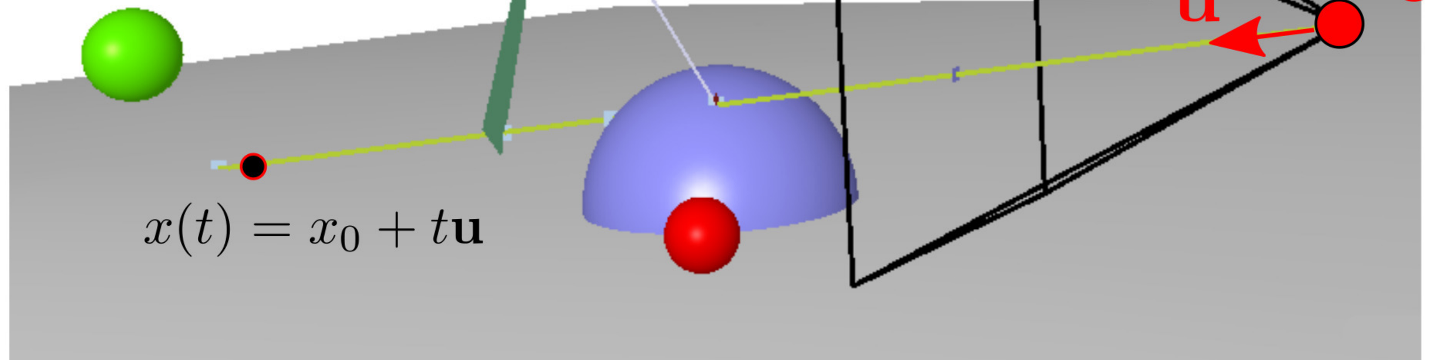
$$\begin{cases} F(x,y,z)=0 \\ x(t)=x_0+t\mathbf{u} \end{cases}$$

2: Solve for t

Deduce:
x(t) et **n**(t)

Keep only t>0

Shading
(local properties)



Lancé de rayons

Avantage

- Ombres portées "natives"

- Gestion profondeur "native"

- Toutes formes peut être représenté exactement (intersection segment/forme)
(plans, triangles, sphère, quadriques, fonctions implicites, CSG, etc.)

- Beaucoup d'effets "naturels" (reflection, refraction, atténuation, etc)

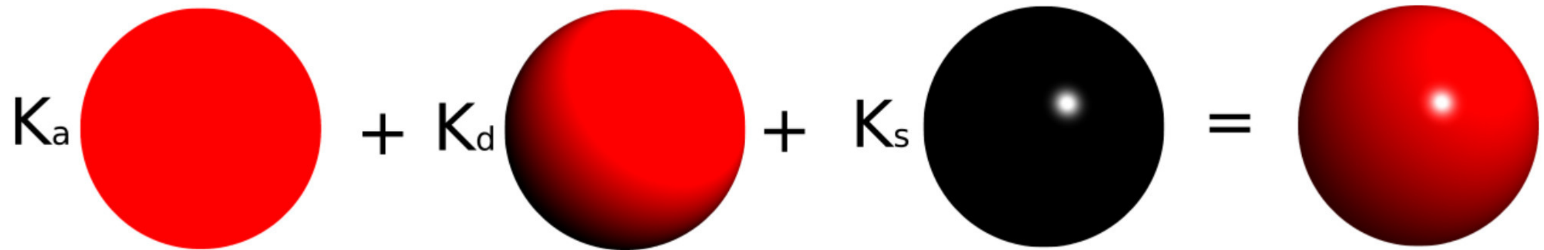
Inconvénients

- Plus lent (plus de calculs)

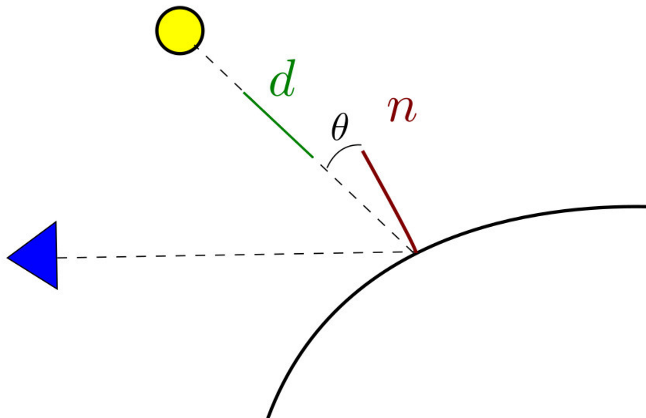
Illumination

Sans illumination: pas d'effet de profondeurs

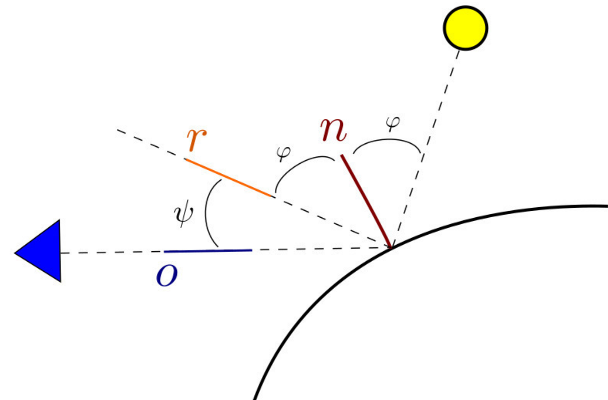
Approche de Gouraud :



Diffuse coefficient $c_d = \cos(\theta)$

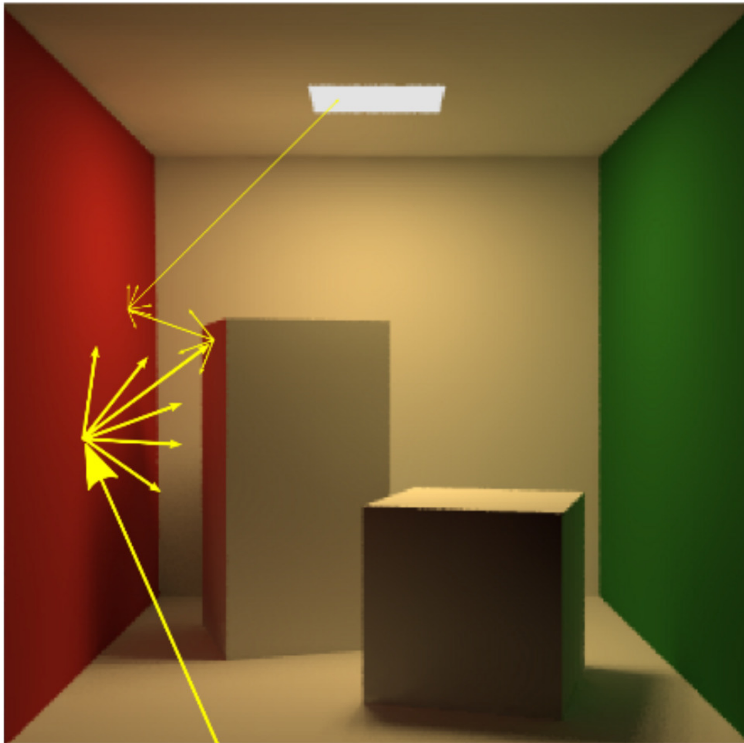


Specular coefficient $c_s = \cos(\psi)^n$



Rendu basé physique

Aujourd'hui: méthodes de calculs prenant en compte les diffusions secondaires.



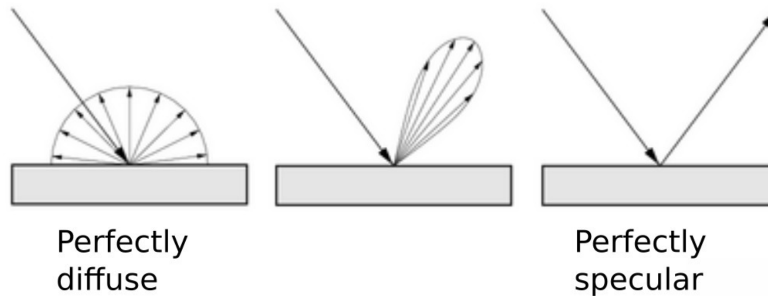
Echantillonage des rayons secondaires

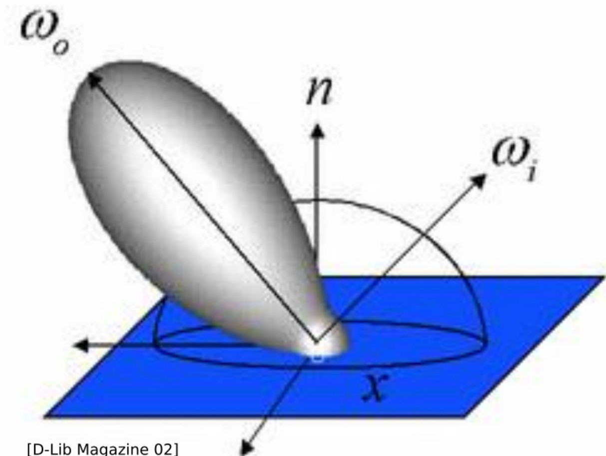
(converge vers résolution équation de la lumière)

Force brute: Path tracing
Metropolis light transport

Amélioration de l'illumination

Bidirectional **R**eflectance **D**istribution **F**unction





[D-Lib Magazine 02]

Les outils de rendus

Enumération

Logiciels libres

Lancé rayon (anciens)

- PovRay
- Yafray

Renderman

- Aqsis
- Pixie

Lancé rayons + GI

- Yaf(a)ray
- Sunflow

Lancé rayons non biaisé

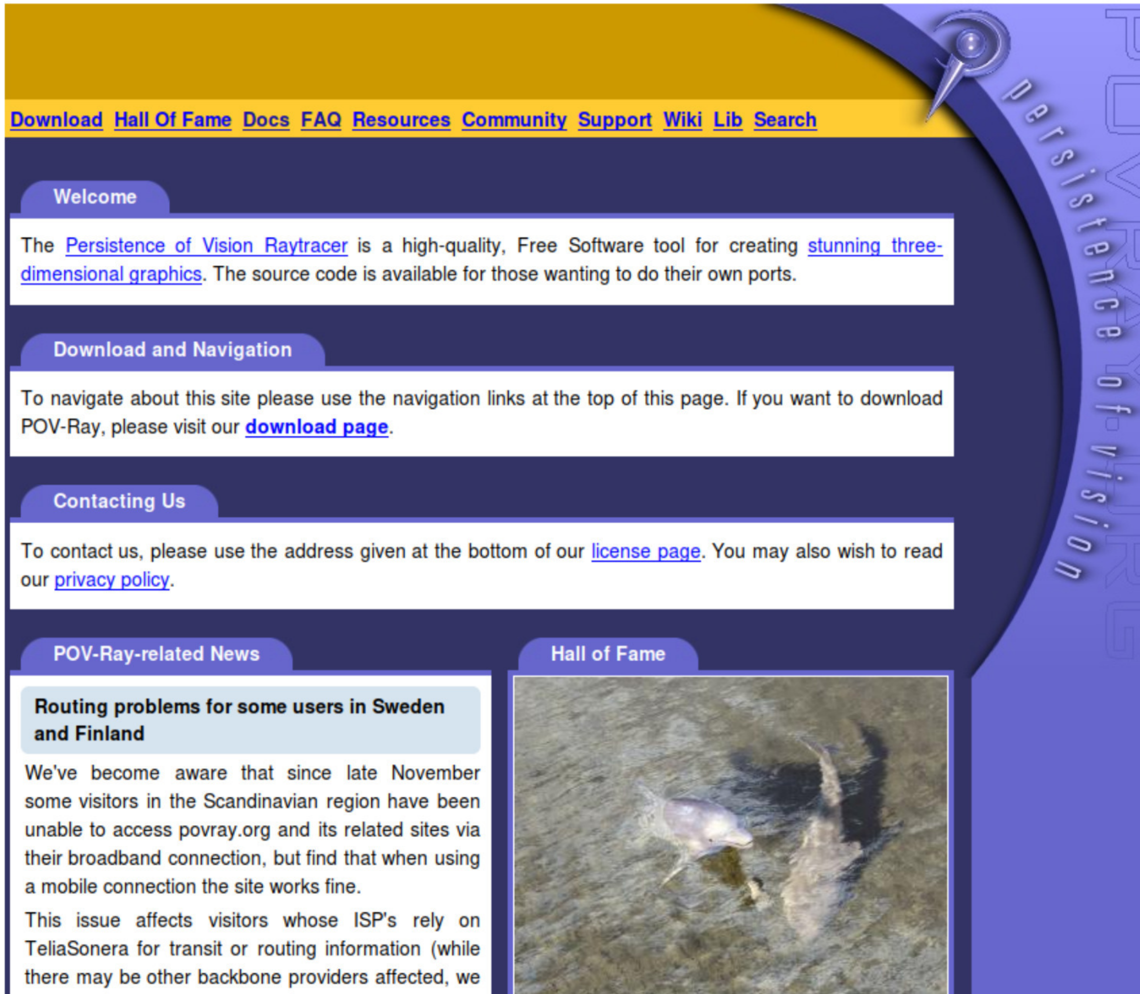
- LuxRender
- Mitsuba
- Blender Cycles

Logiciels commercials

- Indigo
- Maxwell
- Renderman
- Mental Ray
- VRay
- Brazil

Pov-Ray - *L'ancien*

The Persistence of Vision Raytracer



Pov-Ray - *L'ancien*

The Persistence of Vision Raytracer

Description d'une scène 3D
dans un langage de script

(proche d'un langage informatique: boucles, defines, etc)

=> On lance PovRay qui analyse ce script
=> Viens générer le rendu

```
//EXAMPLE OF A "CUBE BUILDING"

#include "colors.inc"
#include "woods.inc"
#include "stones.inc"
#include "metals.inc"
#include "golds.inc"
#include "glass.inc"

//Place the camera
camera {
  sky <0,0,1>          //Don't change this
  direction <-1,0,0>   //Don't change this
  location  <10,10,5>  //Change this to move the camera to a different point
  look_at   <0,0,0>    //Change this to aim the camera at a different point
  right <-4/3,0,0>     //Don't change this
  angle 30
}

//Place a light
light_source {
  <10,-10,10>  //Change this if you want to put the light at a different point
  color White*3
}

//Set a background color
background { color White*2 }

//Create a "floor"
plane {
  <0,0,-1>,0
  texture { T_Silver_1A }
}

//Create a box that extends between the 2 specified points
#declare mycube = box {
  <0,0,0>  // one corner position <X1 Y1 Z1>
  <1,1,1>  // other corner position <X2 Y2 Z2>
}

//Change cubes and their locations below this point.

object { mycube texture {T_Stone32} }
object { mycube translate <1,0,0> texture {T_Stone32} }
object { mycube translate <0,1,0> texture {T_Stone32} }
object { mycube translate <0,2,0> texture {T_Stone32} }
object { mycube translate <0,1,1> texture {T_Stone32} }
```

Pov-Ray - *L'ancien*

Avantages

- Ancien, testé, pas/peu de bugs
- Très bien documenté
- Beaucoup de ressources, wiki, tutoriaux, etc
- - valeur sure
- Langage de script puissant
- Diverses représentations (mesh, isosurfaces, volume, fractales)

Inconvénients

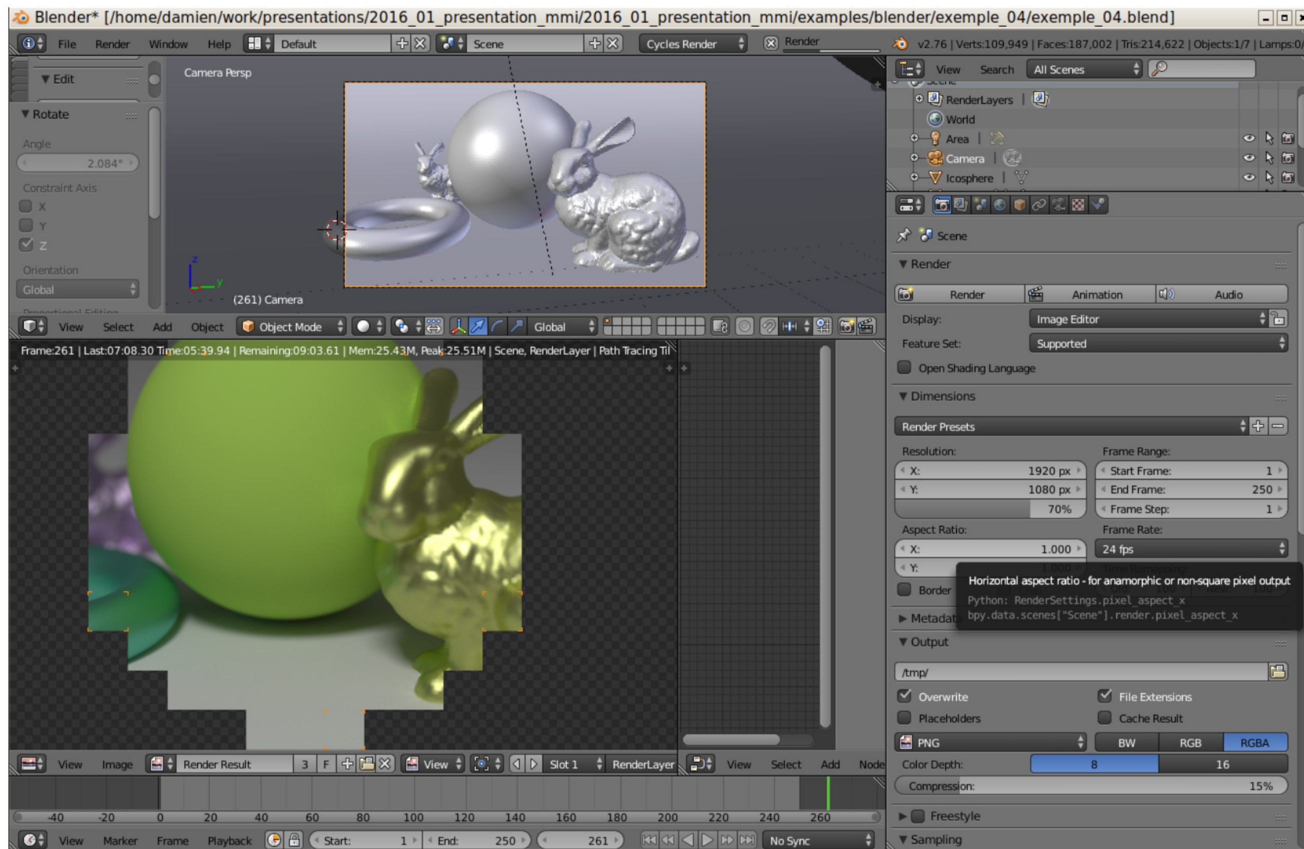
- Rendu des maillages (shadow line artifact)
- Lent pour les ombres douces et effets avancés
- Illumination globale lente
- Langage de script incompatible avec logiciel de modélisation

=> Valeur sure, simple d'accès et puissant.
=> Idéal pour paramétrique, isosurfaces et fractales.
=> Pas idéal pour les maillages
=> Plus lent que les nouveaux moteurs

Blender Cycles

Moteur de rendu physique intégré à Blender

logiciel de modélisation 3D



Blender Cycles

Principe:
- On créé une scène 3D dans Blender
- On paramètre les matériaux et effets dans Blender (GUI)
- On lance le rendu avec Cycles à partir de la scène Blender

rendu itératif (path tracing, etc)

→ Tout se fait depuis Blender

Merci de votre attention