

IFT3355

INFOGRAPHIE

SECTION 12: SYSTÈMES DE RENDU

Derek Nowrouzezahrai et Pierre Poulin

Département d'informatique et de recherche opérationnelle

Université de Montréal

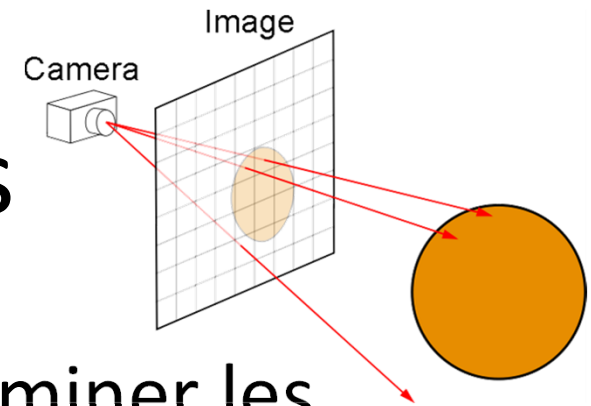


Systemes de rendu: ça sert à quoi?

- Les trois systemes que nous étudierons ont été toutes conçus pour résoudre le même problème:
la détermination des surfaces visibles par respect de l'œil (*visible surface determination*)
- Chacun a également évolué afin de supporter des effets beaucoup plus complexes (ex: GI)
 - En fait, les limitations structurels et/ou inconvenients présents dans chacun de ces architectures peuvent être typiquement attribués au fait que l'ombrage / l'illumination a seulement été considéré bien après la conception originale de chaque architecture



Famille 1: Lancer de rayons



- Originellement conçu pour déterminer les surfaces visibles de l'œil (caméra *pinhole*)

```
for( chaque pixel p )  
  Générer un rayon r de l'œil à p  
  
  float dist_min = infinité  
  Intersection h = null  
  
  for( chaque objet o )  
    if( trace( r, o ) et hit_dist <= dist_min )  
      dist_min = hit_dist  
      h.objet = o et h.dist = hit_dist  
  
  p.couleur = shade(h)
```



Famille 1: Lancer de rayons

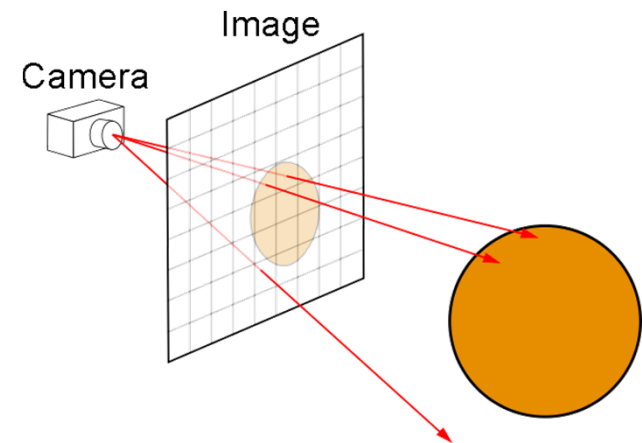
Visibilité:

1. Générer les rayons de l'oeil
2. Coder un fonction **trace** pour
 - des plans
 - des sphères
 - des triangles, etc.
3. Trouver l'intersection **la plus proche**

Illumination (locale, globale):

4. Qu'est-ce qu'on peut accomplir avec **shade**?

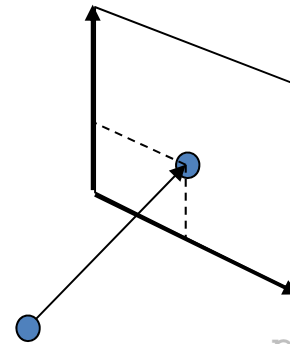
```
for( chaque pixel p )  
  Générer rayon r de l'oeil à p  
  float dist_min = infinité  
  intersection h = null  
  for( chaque objet o )  
    if( trace( r, o ) et  
        hit_dist <= dist_min )  
      dist_min = hit_dist  
      h.objet = o  
      h.dist = hit_dist  
p.couleur = shade(h)
```



La forme générale d'un rayon

```
for( chaque pixel p )  
    Générer rayon r de l'oeil à p  
    float dist_min = infinité  
    intersection h = null  
    for( chaque objet o )  
        if( trace( r, o ) et  
            hit_dist <= dist_min )  
            dist_min = hit_dist  
            h.objet = o  
            h.dist = hit_dist  
p.couleur = shade(h)
```

- Un rayon est défini par
 - une origine (x_0, y_0, z_0)
 - une direction $(\Delta x, \Delta y, \Delta z)$
- Pour les rayons de l'oeil:
 - défini par la position du caméra et un point sur la fenêtre (x_1, y_1, z_1)



- Sous forme paramétrique un rayon s'exprime comme

$$P(t) = P_0 + t(P_1 - P_0)$$

$$x = x_0 + t\Delta x \quad y = y_0 + t\Delta y \quad z = z_0 + t\Delta z$$

$$\Delta x = x_1 - x_0 \quad \Delta y = y_1 - y_0 \quad \Delta z = z_1 - z_0$$



Intersection avec un plan

- L'intersection d'un rayon avec un plan (A,B,C,D) consiste à exprimer *les points sur le plan* en fonction des points sur le rayon qui nous intéresse

$$Ax + By + Cz + D = 0$$

$$A(x_0 + t\Delta x) + B(y_0 + t\Delta y) + C(z_0 + t\Delta z) + D = 0$$

$$t = \frac{-(Ax_0 + By_0 + Cz_0 + D)}{A\Delta x + B\Delta y + C\Delta z}$$

- Pour l'intersection avec un polygone, il faudra déterminer si le point d'intersection est à l'intérieur du polygone



```
for( chaque pixel p )  
    Générer rayon r de l'oeil à p  
    float dist_min = infinité  
    intersection h = null  
    for( chaque objet o )  
        if( trace( r, o ) et  
            hit_dist <= dist_min )  
            dist_min = hit_dist  
            h.objet = o  
            h.dist = hit_dist  
p.couleur = shade(h)
```

Intersection avec une sphère

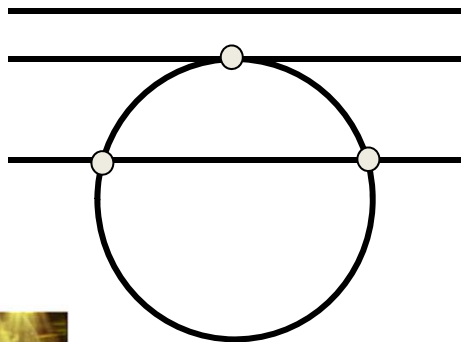
- Similairement pour l'intersection d'un rayon avec une sphère

$$x^2 + y^2 + z^2 = 1$$

$$(x_0 + t\Delta x)^2 + (y_0 + t\Delta y)^2 + (z_0 + t\Delta z)^2 = 1$$

$$(\Delta x^2 + \Delta y^2 + \Delta z^2)t^2 + 2(x_0\Delta x + y_0\Delta y + z_0\Delta z)t + (x_0^2 + y_0^2 + z_0^2 - 1) = 0$$

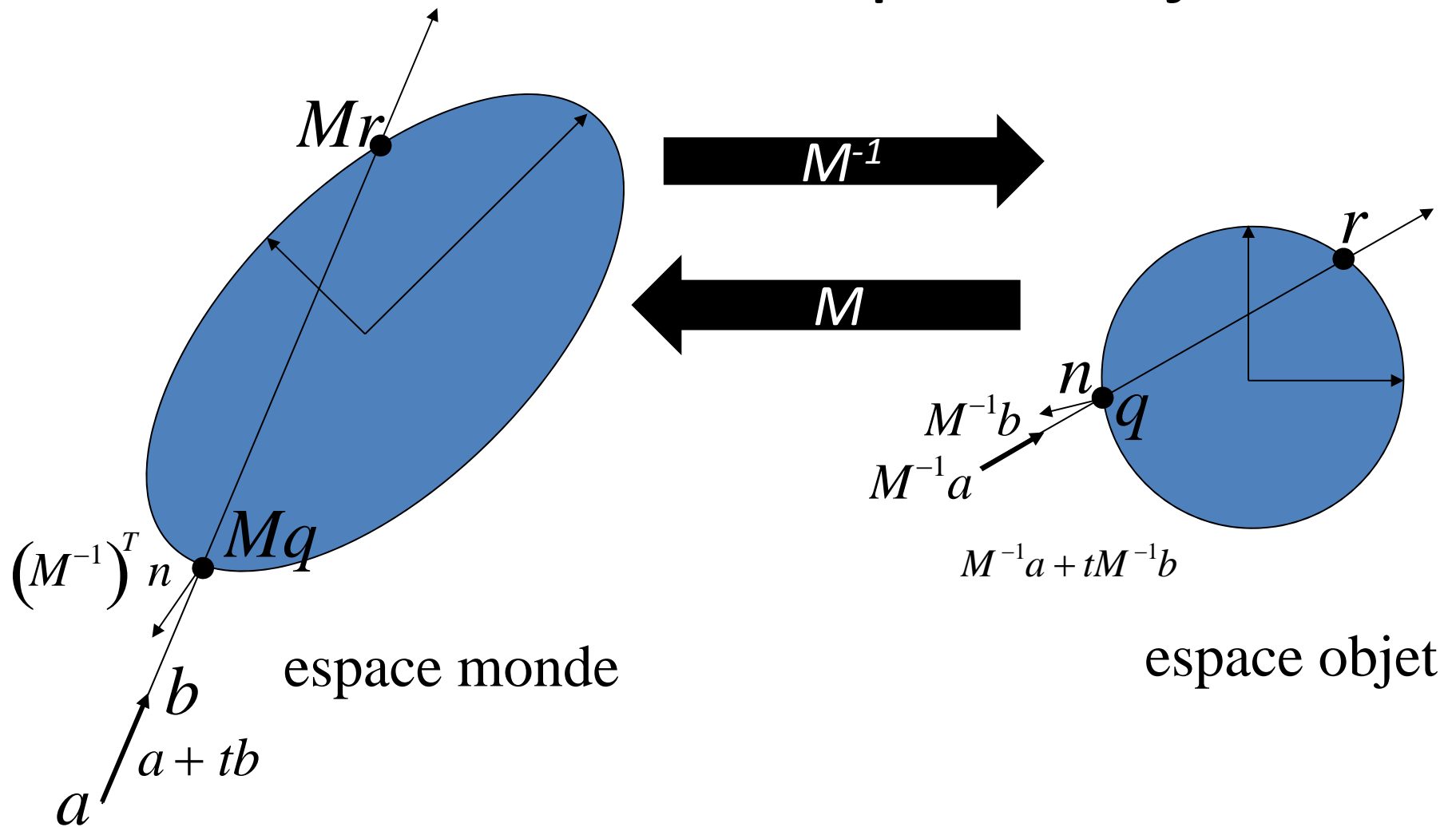
$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$



0, 1 ou 2 intersections réelles



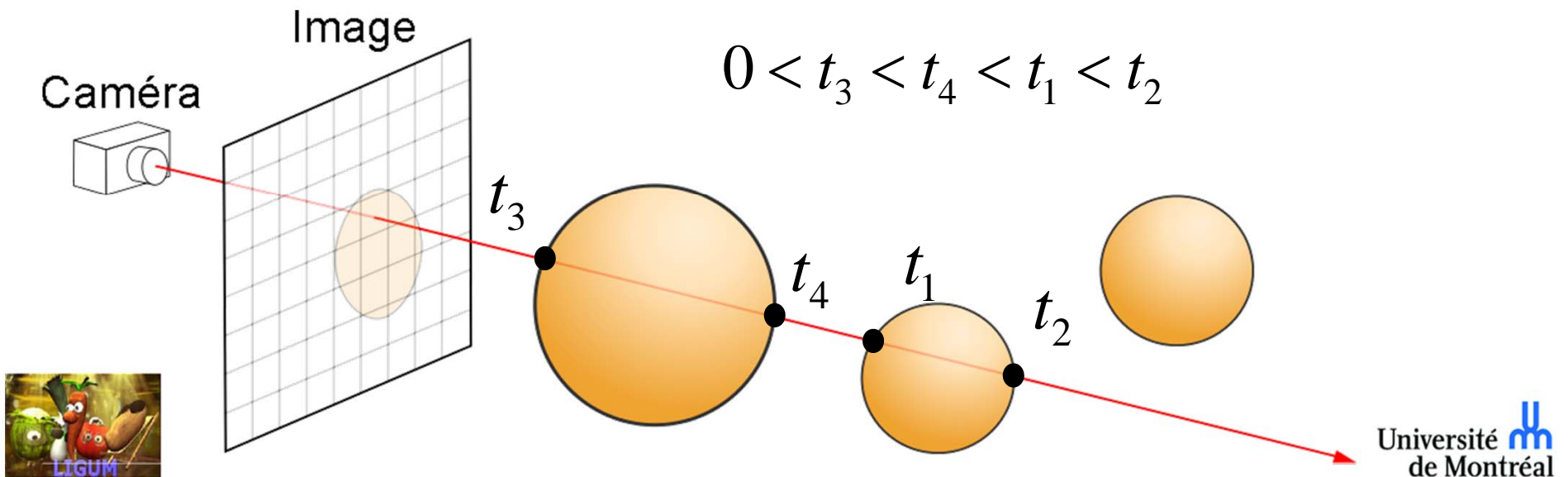
Intersection en espace objet



Famille 1: Lancer de rayons

- Pour identifier la surface visible du pixel/caméra, il suffit d'intersecter *tous* les objets et de ne conserver que l'intersection ayant la plus petite valeur positive de t

```
for( chaque pixel p )  
  Générer rayon r de l'oeil à p  
  float dist_min = infinité  
  intersection h = null  
  for( chaque objet o )  
    if( trace( r, o ) et  
        hit_dist <= dist_min )  
      dist_min = hit_dist  
      h.objet = o  
      h.dist = hit_dist  
  p.couleur = shade(h)
```



Famille 1: Lancer de rayons

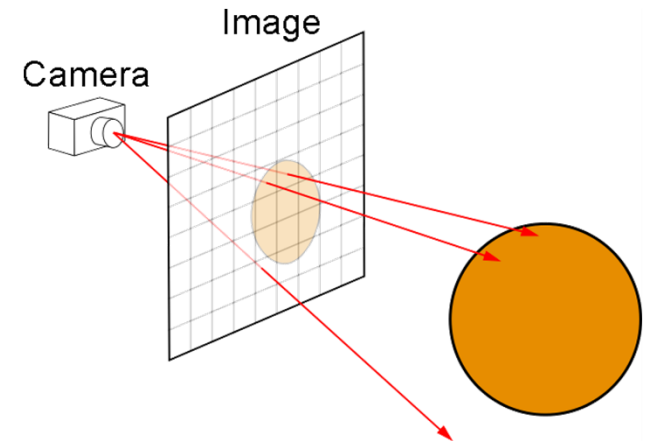
- Compléxité

- *brute-force* en temps: $O(p n)$

- avec un structure d'accélération: $O(p \log n)$

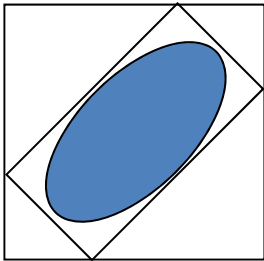
- Noter qu'un tel structure aura besoin de $O(n \log n)$ de temps pour sa construction!

- Noter aussi que chaque représentation supplémentaire de ta scène encourt un coût en mémoire!

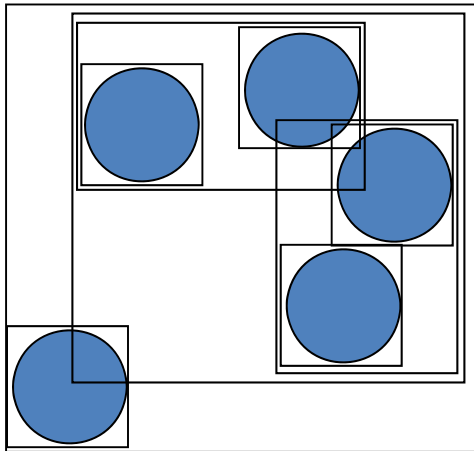


Volumes englobants

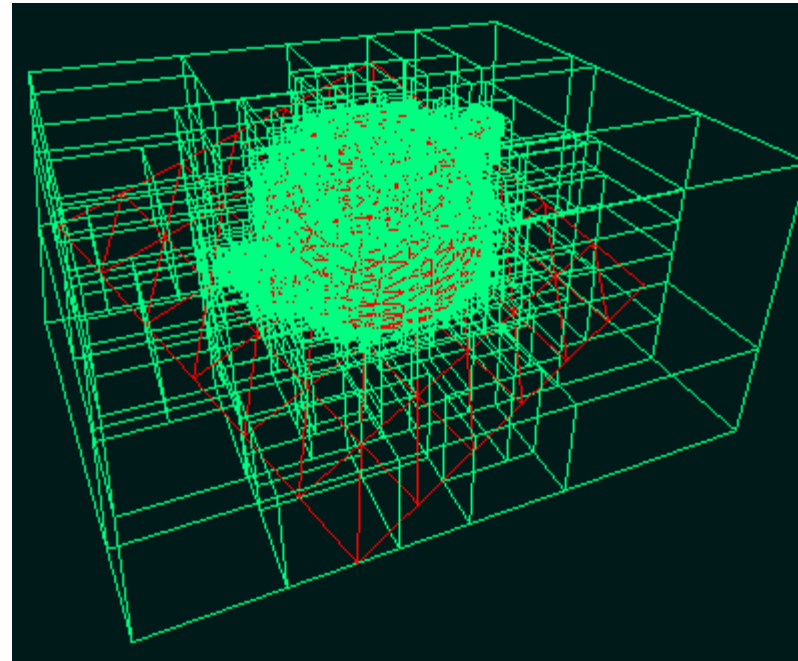
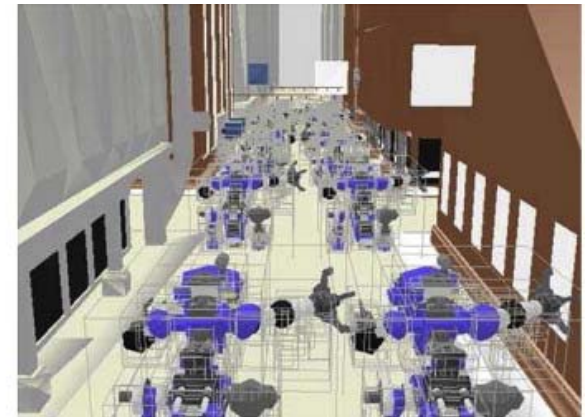
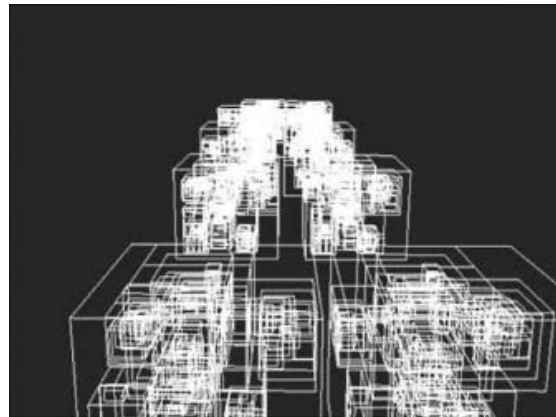
boîte orientée



boîte alignée
sur les axes

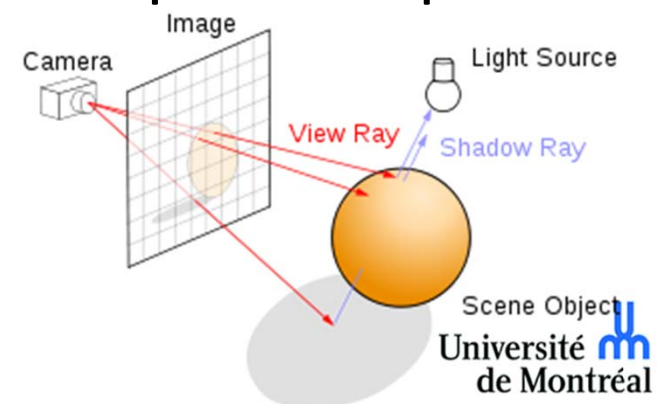


hiérarchie de
boîtes englobantes
(*BVH*)



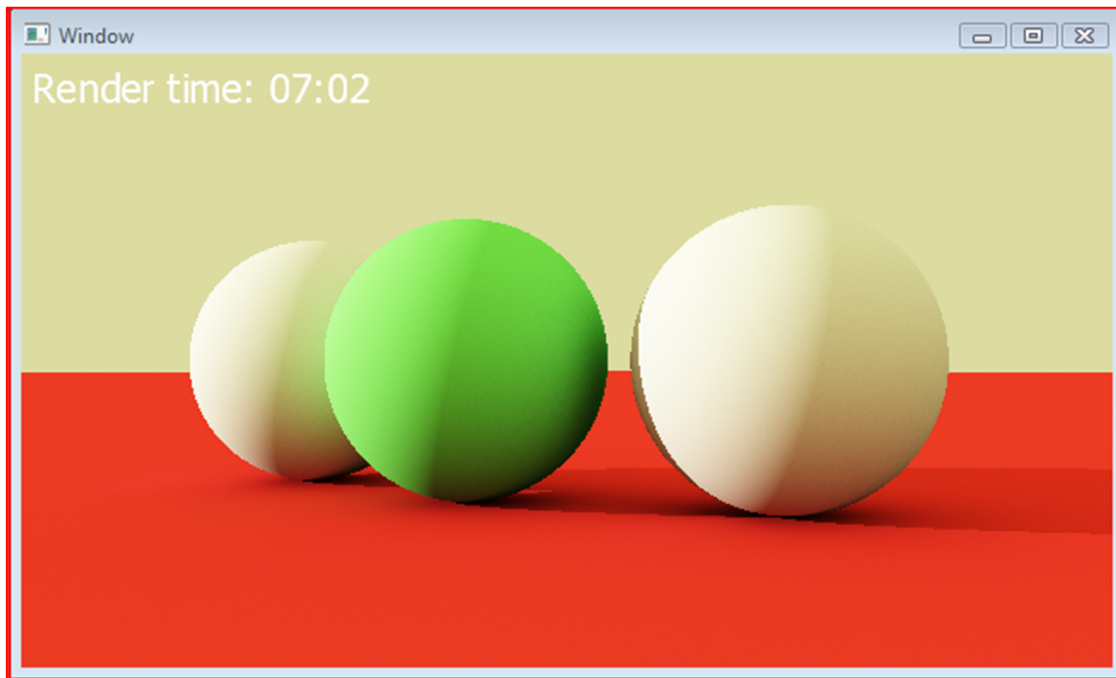
Famille 1: Lancer de rayons

- Les lanceurs de rayons ont graduellement été étendus afin de permettre la simulation des effets de plus en plus complexes comme:
 - Les ombres dur et les réflexions miroir parfaits
 - Le flou de mouvement (*motion blur*)
 - Profondeur de champ (*depth of field*)
 - Simulation de modèle d'illumination plus complexes
 - L'illumination globale



Famille 1: Lancer de rayons

- Presque toutes ces extensions se réalisent complètement en dehors de l'algorithme de base



```
for( chaque pixel p )
    Générer rayon r de l'oeil à p
    float dist_min = infinité
    intersection h = null
    for( chaque objet o )
        if( trace( r, o ) et
            hit_dist <= dist_min )
            dist_min = hit_dist
            h.objet = o
            h.dist = hit_dist
p.couleur = shade(h)
```



Famille 1: Lancer de rayons

- Implémentation HW
 - Deux problèmes avec les lanceurs de rayons qui rends les implémentations HW difficiles:
 - Cohérence des rayons secondaires, et
 - Accès globale de la scène
- Construire et traverser une structure d'accélération sur ex: un GPU d'un manière efficace est très difficile



High Performance Graphics (2008)
M. Duggan, S. Laine, and W. Hart (Editors)

Architecture Considerations for Tracing Incoherent Rays

Timo Aila
Timo Karras
NVIDIA Research

Abstract

This paper proposes a massively parallel hardware architecture for efficient tracing of incoherent rays, e.g. for global illumination. The general approach is centered around hierarchical treelet subdivision of the acceleration structure and repeated spawning/spawning of rays to reduce cache pressure. We describe a heuristic algorithm for determining the treelet subdivision, and show that our architecture can reduce the total memory bandwidth requirements by up to 50% in difficult scenes. Furthermore the architecture allows subdividing rays in an arbitrary order with practically no performance penalty. We also conclude that scheduling algorithms can have an important effect on results, and that using fixed-size queues is not an appealing design choice. Increased auxiliary register, including incoherent stacks, is identified as the foremost remaining challenge of this architecture.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture—Graphics Processors I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

1. Introduction

We study how GPUs and other massively parallel computing architectures could evolve to more efficiently trace incoherent rays in massive scenes. In particular our focus is on how architectural as well as algorithmic decisions affect the memory bandwidth requirements in cases where memory traffic is the primary factor limiting performance. In these cases, the concurrently executing rays tend to access different portions of the scene data, and their collective working set is too large to fit into caches. To alleviate this effect, the execution needs to be modified in some non-trivial ways.

It is somewhat poorly understood when exactly the memory bandwidth is the primary performance bottleneck of ray tracing. It was widely believed that that was already the case on GPUs but Aila and Laine [AL09] recently concluded that simple scenes that were not the case, even without caches. Yet, those scenes were very simple. We believe that applications will generally follow the trend seen in rendered movies, for example Avatar already has billions of geometric primitives in most complex shots. If we furthermore target incoherent rays arising from global illumination computations, it seems unlikely that currently available caches would be large enough to absorb a significant portion of the traffic.

Also, the processing cores of current massively parallel

computing systems have not been tailored for ray tracing computations, and significant performance improvements could be obtained by using e.g. fixed-function hardware for traversal and intersection [SW02, SW04, WS05]. However, for such custom units to be truly useful, we must be able to feed them enough data, and eventually we will need to sustain immense data rates when the rays are highly incoherent, perhaps arising from global illumination computation. In this paper we seek architectural solutions for situations where memory traffic is the primary bottleneck, while not engaging into further discussion about when and where these conditions might arise.

We study this problem on a hypothetical parallel computing architecture, which is sized according to NVIDIA Fermi [NV10]. We observe that with current approaches the memory bandwidth requirements can be well over hundred times higher than the theoretical lower bound, and that traversal cache traffic is a significant issue. We build on the ray scheduling work of Purrer et al. [PKGH07] and Naratal et al. [NLM07], and extend their concept of queue-based scheduling to massively parallel architectures. We analyze the primary architectural requirements as well as the bottlenecks in terms of memory traffic.

Our results indicate that scheduling algorithms can have an important effect on results, and that using fixed-size

Understanding the Efficiency of Ray Traversal on GPUs

Timo Aila*
NVIDIA Research

Samuli Laine*
NVIDIA Research

Abstract

We discuss the mapping of elementary ray tracing operations—acceleration structure traversal and primitive intersection—to wide SIMD/SMT machines. Our focus is on NVIDIA GPUs, but very little is usually mentioned about their performance. Nobody knows whether the methods are any where near the theoretically obtainable limits, and if not, what might be causing the discrepancy. We study this question by comparing the measurements against a simulator that sets the upper bound of performance for a given kernel. We observe that previously known methods are a factor of 1.5–2.5X off from theoretical optimum, and most of the gap is not explained by memory bandwidth, but rather by previously unobserved inefficiencies in hardware work distribution. We then propose a simple solution that significantly narrows the gap between simulation and measurement. This results in the fastest GPU ray tracer to date. We provide results for primary, ambient occlusion and diffuse interreflection rays.

CR Categories: I.3.1 [Computer Graphics]: Picture/Image Generation—(I.3.7) Computer Graphics—Three-Dimensional Graphics and Realism

Keywords: Ray tracing, SIMD, SMT

1 Introduction

This paper analyzes what currently limits the performance of acceleration structure traversal and primitive intersection on GPUs. We refer to these two operations collectively as *traverse*. A major question in optimizing *traverse* on any platform is whether the performance is primarily limited by computation, memory bandwidth, or perhaps something else. While the answer may depend on various aspects, including the scene, acceleration structure, viewport, and ray load characteristics, we argue the situation is poorly understood on GPUs in almost all cases. We approach the problem by implementing optimized variants of some of the fastest GPU *traverse* kernels in CUDA (NVIDIA 2008) and compare the measured performance against a custom simulator that sets the upper bound of performance for that kernel on a particular NVIDIA GPU. The simulator will be discussed in Section 2.1. It turns out that the current kernels are a factor of 1.5–2.5 below the theoretical optimum, and that the primary culprit is hardware work distribution. We propose a simple solution for significantly narrowing the gap. We will then introduce the concept of speculative traversal, which is applicable beyond NVIDIA's architecture. Finally we will discuss approaches that do not pay off today, but could improve performance on future architectures.

*Email: {aila,laine}@nvidia.com

Scope This document focuses exclusively on the efficiency of *traverse* on GPUs. In particular we will not discuss shading, which may or may not be a major cost depending on the application. We will also not discuss the important task of building and maintaining acceleration structures.

Hierarchical traversal In a coherent setting, truly impressive performance improvements have been reported on GPUs using hierarchical traversal methods (e.g. [Rechen et al. 2005, Wald et al. 2007]). These methods are particularly useful with coherent rays, e.g. primary, shadow, specular reflection, or equally localized rays such as short ambient occlusion rays. It is not yet clear how beneficial these techniques can be on wide SIMD machines, or with incoherent rays such as diffuse or mildly glossy interreflection. We acknowledge that one should use hierarchical traversal methods whenever applicable, but this article will not cover that topic. Apart from special cases, even the hierarchical methods will revert to tracing individual rays near the leaf nodes. The awkward fact is that when using hierarchical traversal methods, only the most coherent part of the operations near the root gets accelerated, often leaving seriously incoherent per-ray workloads to be dealt with. This appears to be one reason why very few, if any, meaningful performance gains have been reported from hierarchical tracing on GPUs.

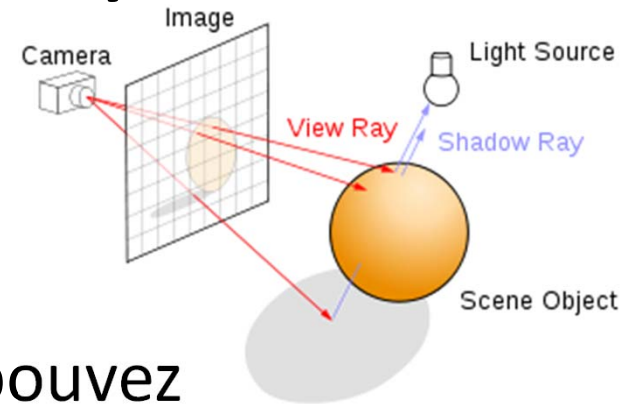
SMT/SIMD SMT is a superset of SIMD with execution divergence handling built into hardware. SMT typically completes all control decisions, memory addresses, etc. separately on every (SMT) lane [Lindholm et al. 2008]. The execution of *traverse* consists of an unpredictable sequence of node traversal and primitive intersection operations. This unpredictability can cause some penalties on CPUs too, but on wide SIMD/SMT machines it is a major cause of inefficiency. For example, if a warp executes node traversal, all threads in that warp that want to perform primitive intersection are idle, and vice versa. We refer to the percentage of threads that perform computation as *SMT efficiency*.

2 Test setup

All of the tests in this paper use bounding volume hierarchy (BVH) and Woop's unit triangle intersection test [Woop 2004]. The ray-box and ray-triangle tests were optimized in CUDA 2.1 by examining the native assembly produced. The BVH was built using the greedy surface-area heuristic with maximum leaf node size of 4 for all scenes. To improve tree quality, large triangles were temporarily split during the construction [Hart and Green 2007]. The two BVH child nodes were stored in 64 bytes, and always fetched and tested together. The traversal always proceeds to the closer child. Woop's intersection test requires 48 bytes per triangle. Further information about the scenes is given in Table 1. All measurements were done using an NVIDIA GTX285. Register content of the kernels ranged from 21 to 25 (this variation did not affect performance). We used a thread block size of 102, but most other choices were equally good. Rays were assigned to warps following the Morton order [Laine 2008]. Rays were assigned to warps following the Morton order [Laine 2008]. Rays were assigned to warps following the Morton order [Laine 2008].

*Woop is NVIDIA's name for the design of their ray-trace intersection, usually as a SIMD unit on NVIDIA GPUs.

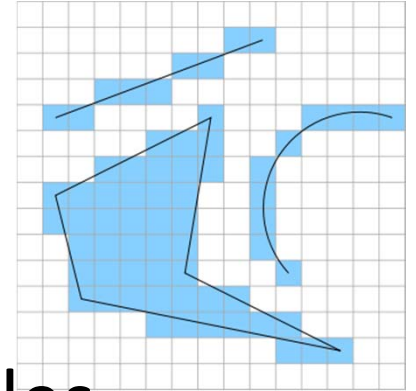
Famille 1: Lancer de rayons



- Avantages:
 - L’algorithme le plus flexible: vous pouvez implémenter n’importe quel algorithme de rendu *précisément* dans un lanceur de rayons
- Désavantages:
 - Typiquement le plus lent des algorithmes
 - Les structures d’accélération prennent typiquement plus d’espace que la scène originale! Difficile (impossible?) à coder comme moteur de *streaming*



Famille 2: Rastérisation



- Similairement conçu pour déterminer les surfaces visibles de l'œil (caméra *pinhole*)
 - Supposition additionnel: que chaque objet est beaucoup plus grand qu'un pixel

```
for( chaque objet o )  
  Projeter o sur le plan d'image  
  
  for( chaque pixel p )  
    if( isinside(p, o) )  
      // effectuer un test de z-buffer  
      if( o.z < zbuffer[p] )  
        zbuffer[p] = o.z  
        p.couleur = shade(o)
```



Famille 2: Rastérisation

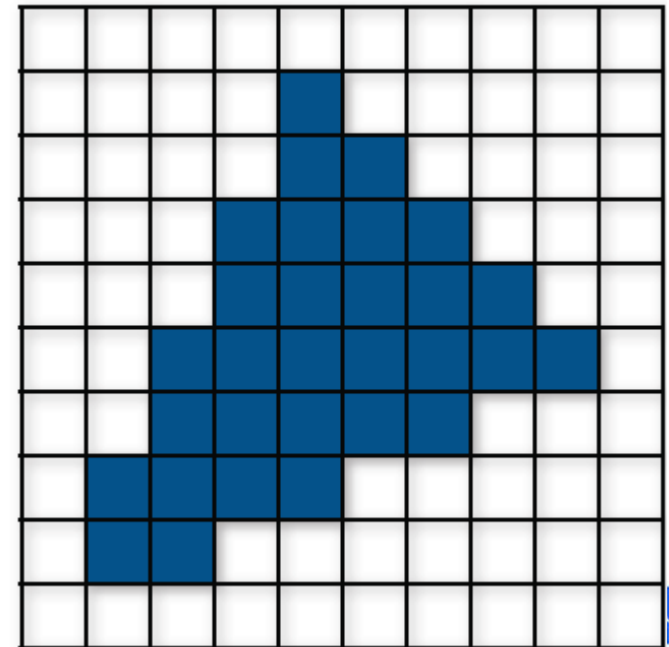
Visibilité:

1. Transformer et projeter chaque objet (triangle)
2. Coder un fonction **isinside** pour
 - des triangles
 - des polygones, etc.
3. Trouver l'objet **la plus proche** avec un *z-buffer*

Illumination (locale, globale):

4. Calculer le couleur du pixel avec **shade**

```
for( chaque objet o )  
    Projeter o sur l'image  
for( chaque pixel p )  
    if( p est dedans o )  
        // test de z-buffer  
        if( o.z < zbuffer[p] )  
            zbuffer[p] = o.z  
            p.couleur = shade(o)
```



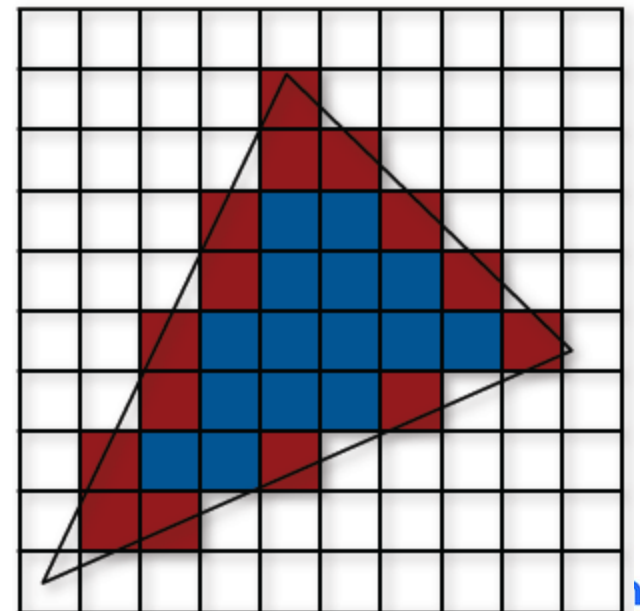
Rastérisation – remplissage: *scan conversion* (séquentielle)

- Après les transformations et projections (selon les matrices standards), la remplissage des triangles est l'opération la plus importante d'un moteur de rastérisation

```
for( chaque objet o )  
  Projeter o sur l'image  
  for( chaque pixel p )  
    if( p est dedans o )  
      // test de z-buffer  
      if( o.z < zbuffer[p] )  
        zbuffer[p] = o.z  
        p.couleur = shade(o)
```

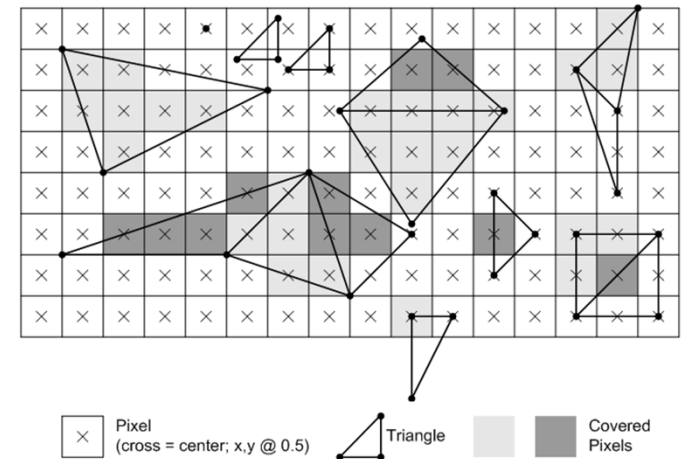
- Une méthode de remplissage qui procède d'une manière séquentielle est le *scan conversion*:

1. Commence en traversant chaque rangée de pixel (un *scan line*)
2. Garder trace des arrêts/bords
3. Remplit les trames de pixels

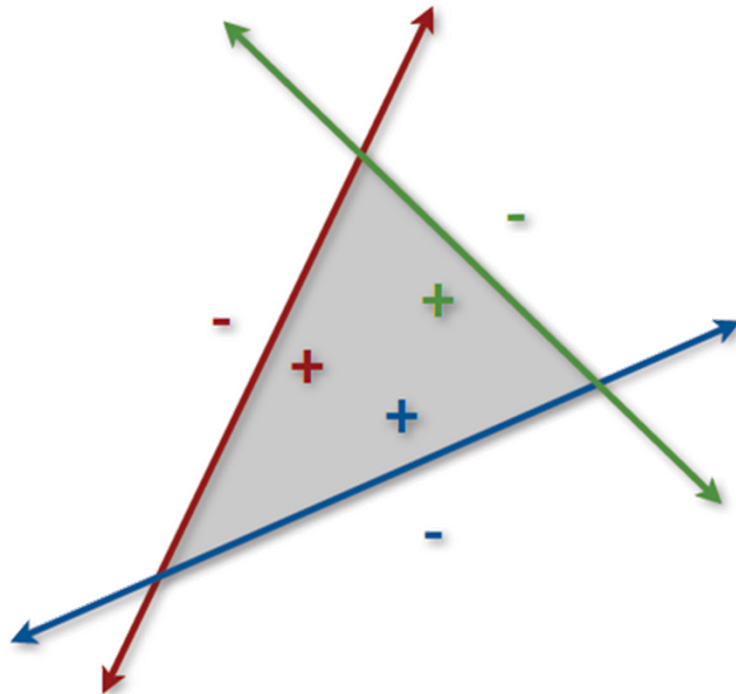


Rastérisation – remplissage: y-a-t'il un *isinside* test parallèle?

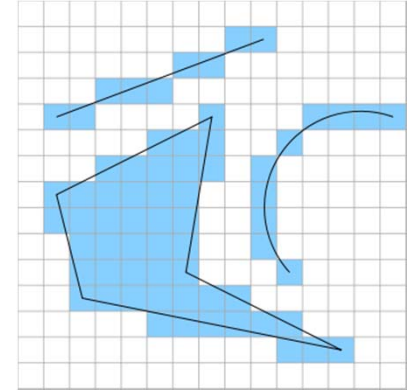
- Il y a plusieurs manières de rapidement déterminer si un pixel est dedans un triangle (en parallèle)
 - Test de demi-espace (*half-space tests*)
 - Utilisation des coordonnées Barycentrique
- Fait attention autour des arrêts!



```
for( chaque objet o )  
  Projeter o sur l'image  
  for( chaque pixel p )  
    if( p est dedans o )  
      // test de z-buffer  
      if( o.z < zbuffer[p] )  
        zbuffer[p] = o.z  
        p.couleur = shade(o)
```



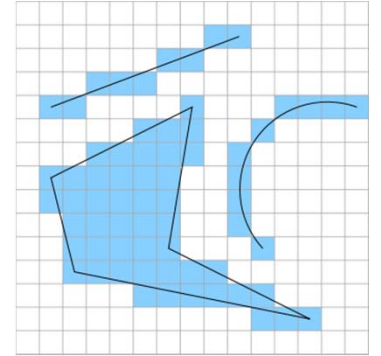
Famille 2: Rastérisation



- Complexité
 - *Brute-force* en temps: $O(n p)$
 - Nous pouvons encore utiliser des structures d'accélération pour améliorer la complexité à $O(p \log n)$
 - *Z-buffer* hiérarchique
 - Boîtes englobantes (projetées): pas nécessaire de tester tous les pixels pour chaque objet pendant le remplissage
 - Pour les scènes statiques, beaucoup d'optimisation possible (ex: *PVS*)
 - La rastérisation traite la géométrie *projetée*, où la détermination de visibilité a lieu en espace image
 - Ces opérations sont beaucoup plus facile à paralléliser



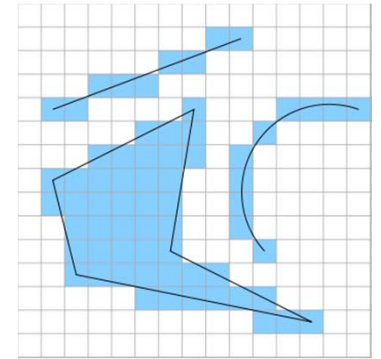
Famille 2: Rastérisation



- Comme avec lancer de rayons, le modèle de rastérisation est toujours en évolution
- Des centaines d'extensions existe pour permettre la simulation des effets plus complexe dans le cadre d'un système de rastérisation:
 - Les ombres dur et étendues
 - L'illumination globale
- Un développement récent s'appelle la rastérisation stochastique qui cible la simulation des effets de distributions



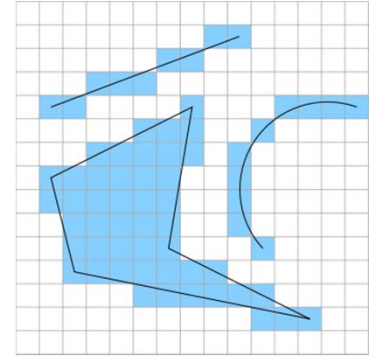
Famille 2: Rastérisation



- Le structure de rastérisation (*streaming-friendly*, opérations en espace image, transformations appliqués indépendamment à chaque objet, etc.) rend son implémentation en HW très facile
- OpenGL et DirectX sont des API modelés autour d'un concept de rastérisation extensible et réalisable sur des GPUs spécialisés



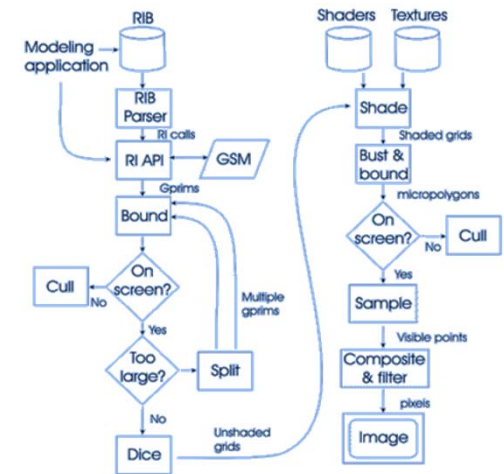
Famille 2: Rastérisation



- Avantages:
 - *Stream-friendly*
 - Facile à paralléliser
- Désavantages
 - Aucun accès direct à des informations *globale*
 - Même l'implémentation des effets simples peut nécessiter des solutions "créatives"
 - Très difficile à simuler des effets d'illumination avancés (où la cohérence en espace rayons est réduite)



Famille 3: Micropolygon



- Quand les objets (projetés) sont plus petit que la taille d'un pixel, on tombe sur un cas particulier:
 - Premièrement, beaucoup des problèmes de visibilité (de l'œil) deviennent plus facile à gérer
 - Deuxièmement, la rasterisation ne marche plus
 - L'interpolation barycentrique entre espace objet est espace image n'est plus nécessaire
- Il existe une troisième famille de système conçu avec cette scénario comme cas de base



Famille 3: Micropolygon

- Contrairement à lancer de rayons et rasterisation, le système micropolygon de **REYES** a été conçu non seulement pour calculer la visibilité de l'œil, mais aussi pour:
 - Gérer des scènes très complexes et avec beaucoup de détails géométriques
 - Supporter quelques effets de shading avancés qui ont été identifiés comme important pour le réalisme: flou de mouvement et *displacement*
 - Première système de shaders programmable



Famille 3: Micropolygon

- L'algorithme de REYES (une exemple de rasterisation de micropolygon) est malheureusement plus complexes qu'un système de lancer de rayons ou de rasterisation
 - Mais il est heureusement similaire à la rasterisation

```
for( chaque objet o )  
    // Convertir objet en micropolygon  
    o' = BoundSplit(o);  
    mp = Dice(o');  
  
    Shade(mp);  
    SampleHide(mp);
```

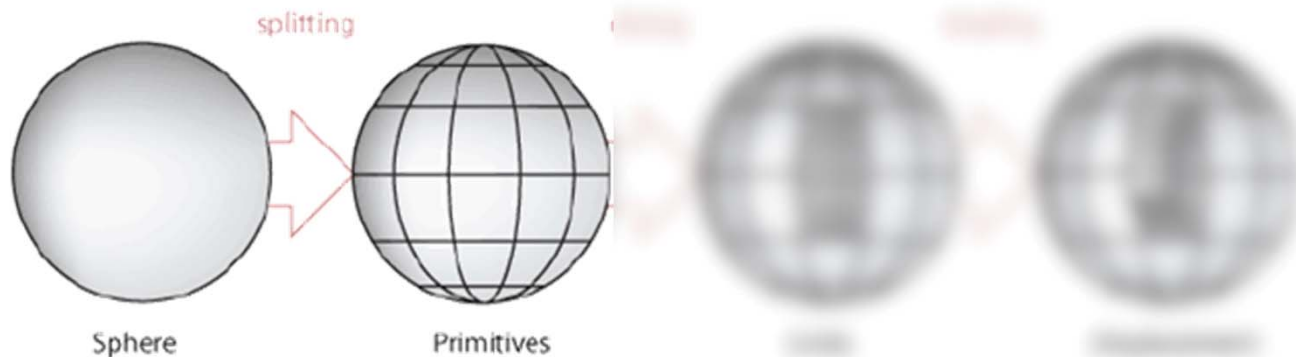
- Les premières étapes ont comme bût de convertir chaque objet dans la scène en représentation commun, des micropolygons, avant le calcul de shading et de visibilité



Famille 3: Micropolygon

- L'étape **Bound** détermine le volume englobant de l'objet et divise l'objet récursivement jusqu'il atteins une taille (projeté) spécifiée en espace image
 - Plusieurs méthodes. Ex: *coarse dicing*, projection des sous-boites englobantes, ...
 - Doit prendre en compte les *displacements* potentiels!

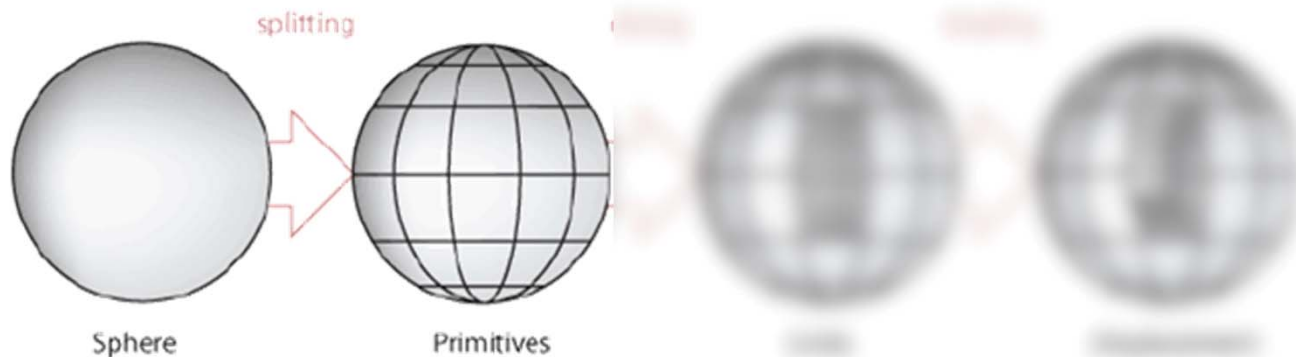
```
for( chaque objet o )  
    o' = BoundSplit(o);  
    mp = Dice(o');  
  
    Shade(mp);  
    SampleHide(mp);
```



Famille 3: Micropolygon

- L'étape `split` permet de sous-diviser des objets afin de jeter des parties qui tomberont en dehors le pyramide de vues
 - Similaire aux algorithmes de clipping en rasterisation

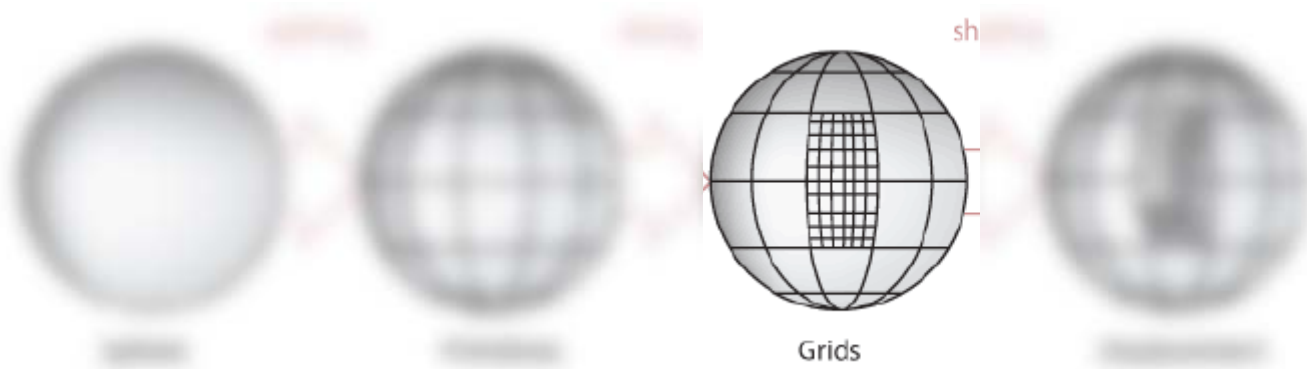
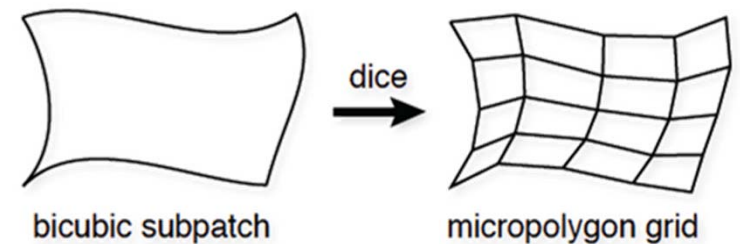
```
for( chaque objet o )  
    o' = BoundSplit(o);  
    mp = Dice(o');  
  
    Shade(mp);  
    SampleHide(mp);
```



Famille 3: Micropolygon

- L'étape `Dice` prends les primitives sous-divisés et les utilise pour créer des micropolygons avec des tailles d'environ un pixel
- Ces micropolygons sont charger dans un tampons appeler le *microgrid*, chacun avec leurs normals, position, et d'autres informations géométriques

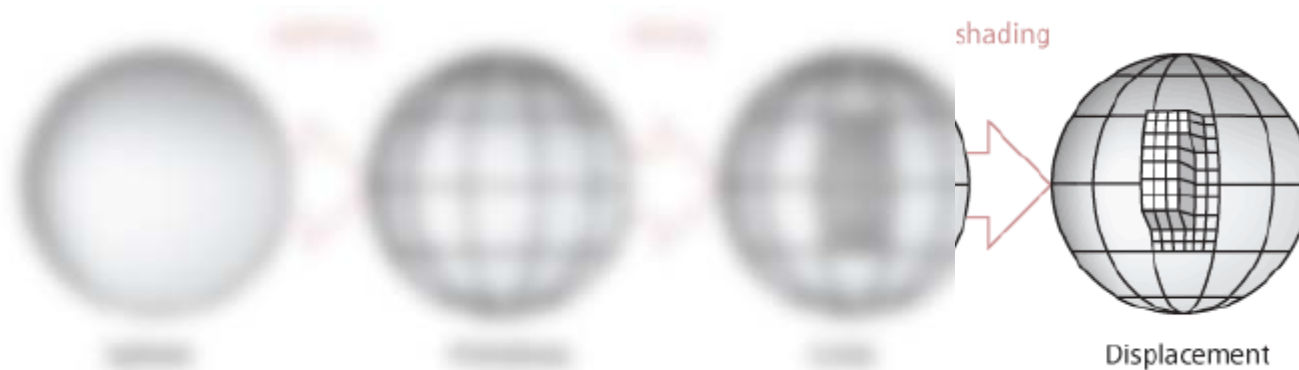
```
for( chaque objet o )  
    o' = BoundSplit(o);  
    mp = Dice(o');  
  
    Shade(mp);  
    SampleHide(mp);
```



Famille 3: Micropolygon

- L'étape **shade** calcule le *displacement* et le *shading* à chaque sommet des micropolygons selon des shaders procédurales
- Tous les propriétés géométriques sont disponibles à une résolution sous-pixel
- Cela peut-être facilement parallélisé

```
for( chaque objet o )  
  o' = BoundSplit(o);  
  mp = Dice(o');  
  
  Shade(mp);  
  SampleHide(mp);
```



Famille 3: Micropolygon

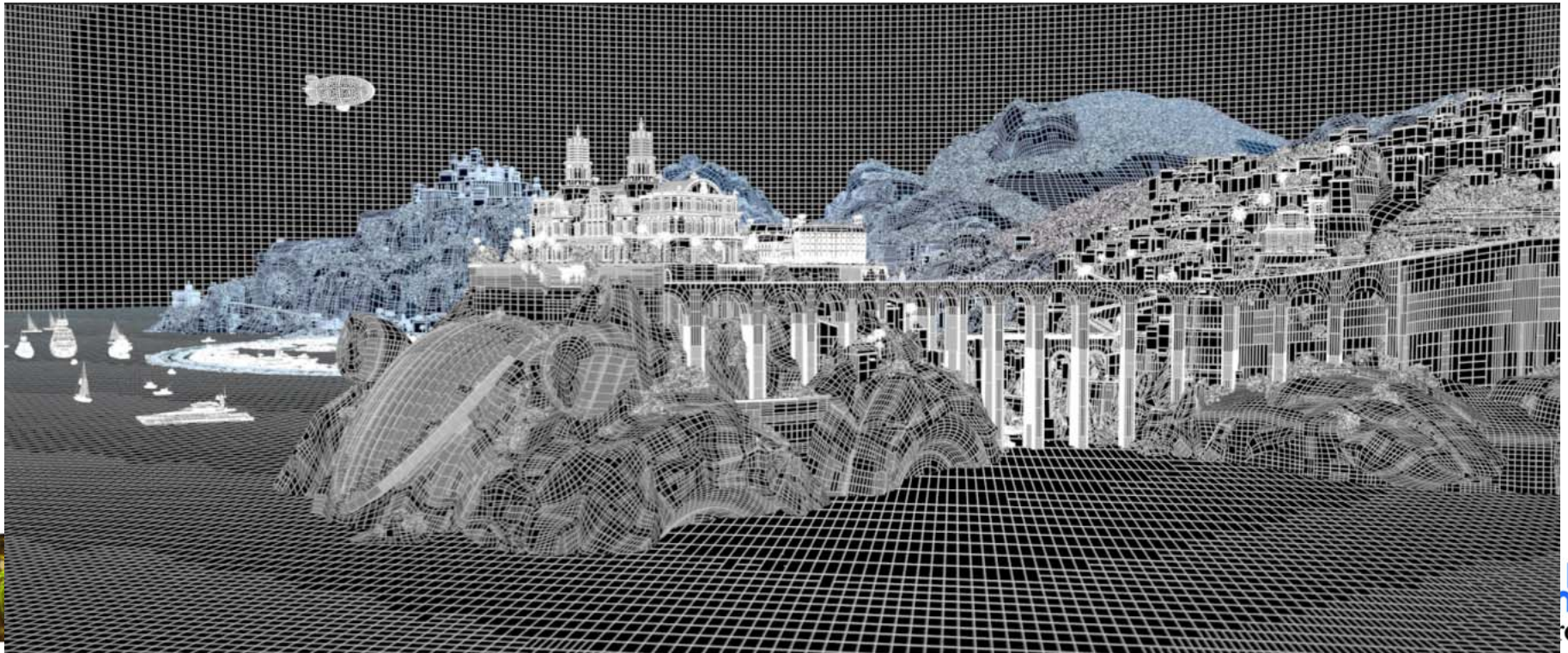
- L'étape `SampleHide` affiche les micropolygons sur l'image finale, après l'application d'une opération de *busting*, tout en éliminant les surfaces cachés
 - Ici les algorithmes de rasterisation et/ou *z-buffer* peuvent être modifiés afin de déterminer la visibilité de l'œil
- À noter: la complexité du shading est découplé de la complexité de la détermination de visibilité

```
for( chaque objet o )  
    o' = BoundSplit(o);  
    mp = Dice(o');  
  
    Shade(mp);  
    SampleHide(mp);
```



Famille 3: Micropolygon

- Récemment plusieurs chercheurs ont investigué les manières d'implémenter un système REYES sur les GPUs, et/ou comment étendre les systèmes de rasterisation accélérés pour supporter les scènes plus complexes avec les effets de rendu avancés



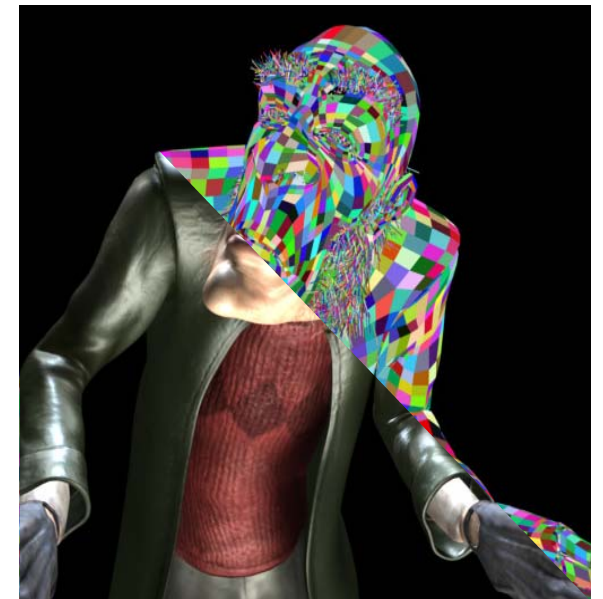
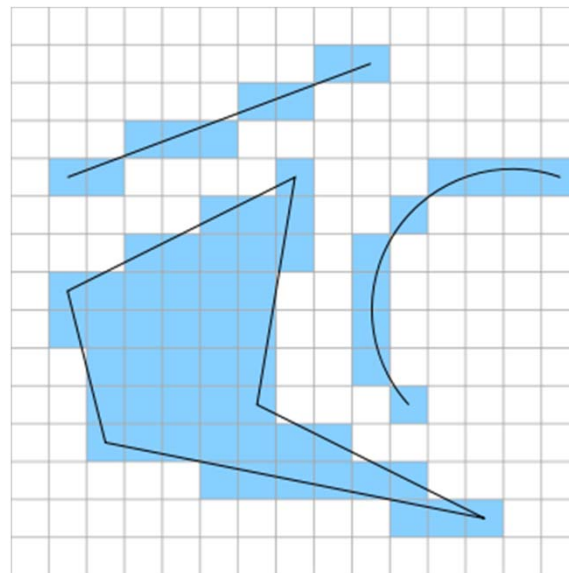
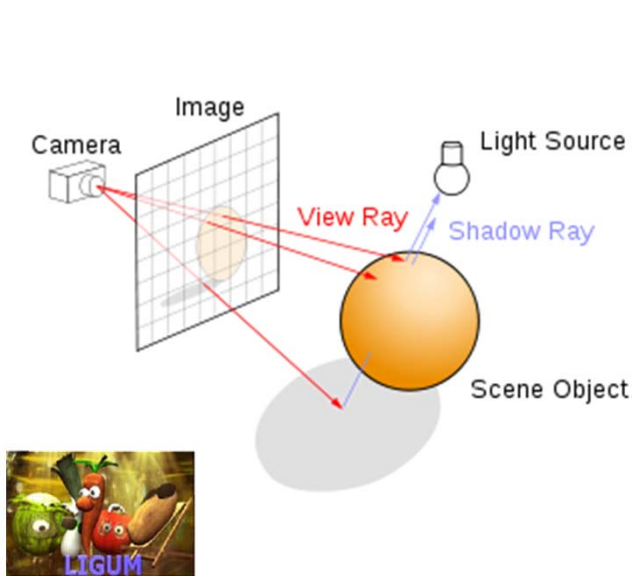
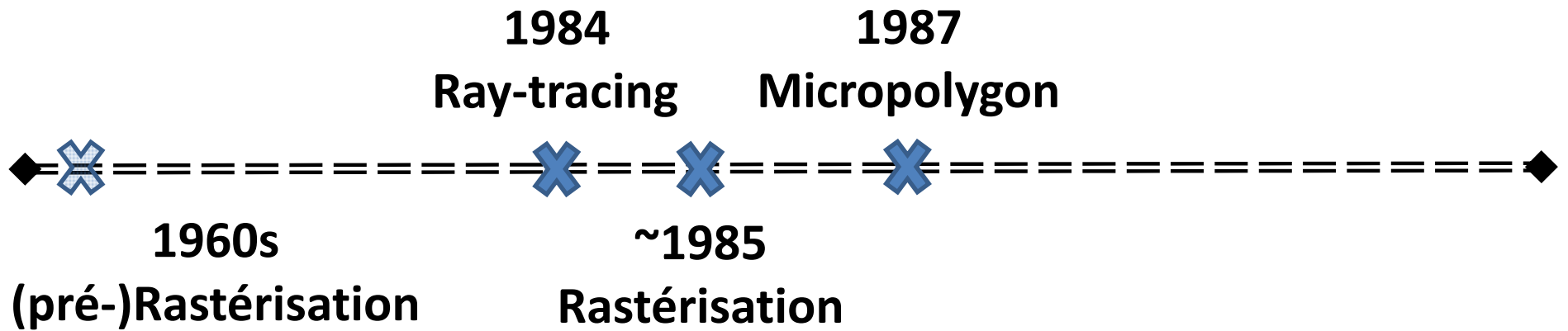
Famille 3: Micropolygon



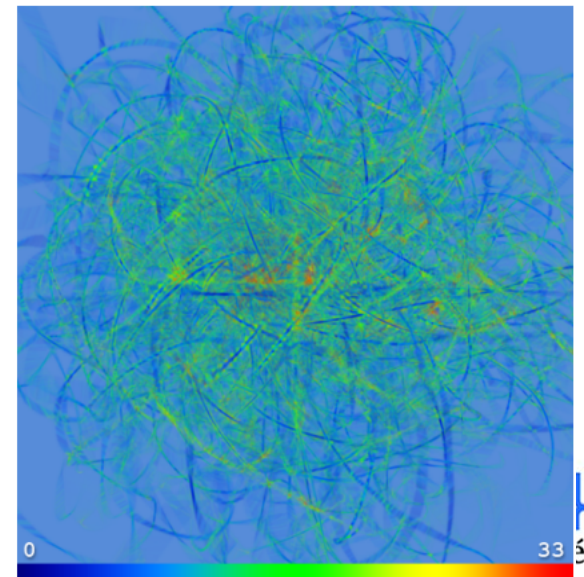
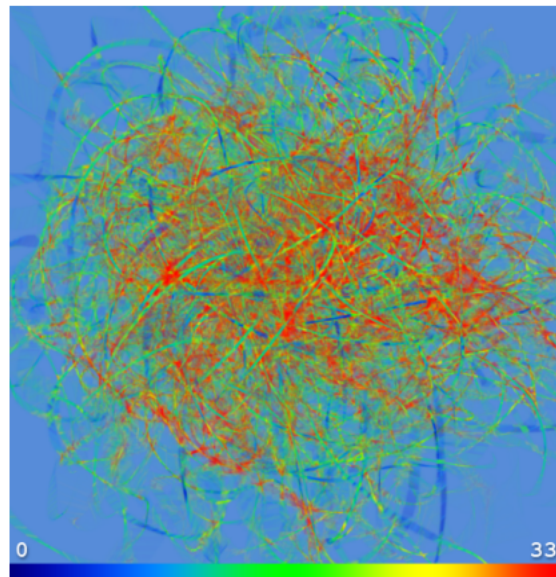
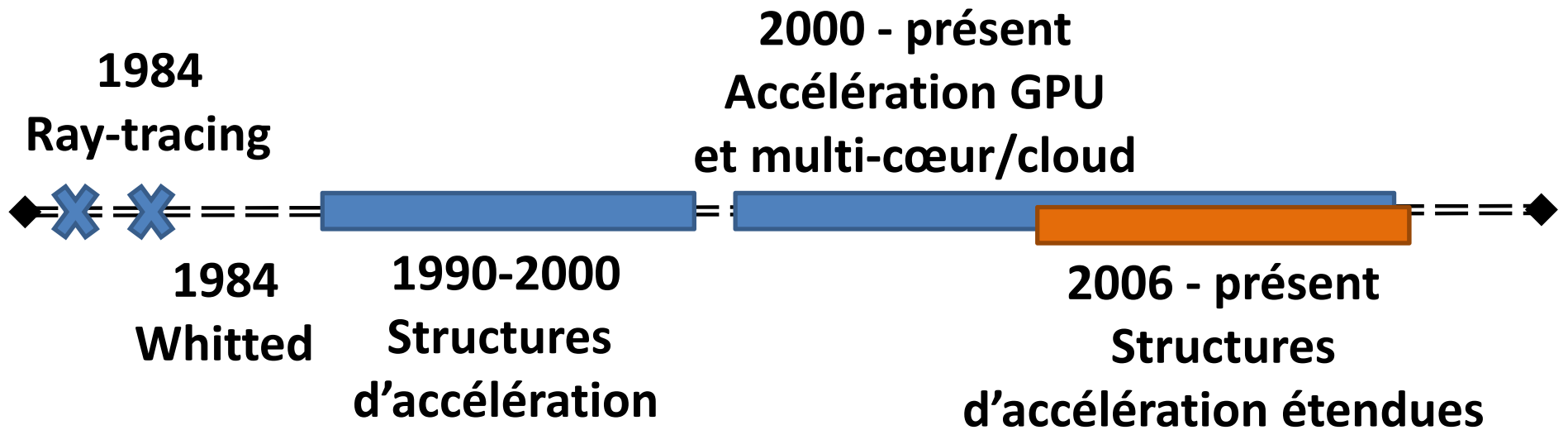
- REYES = algorithme de rasterisation de micropolygon
- Renderman = un standard pour permettre les applications de modélisation / animation de communiquer avec un moteur de rendu
 - N'exige pas un moteur de rendu de micropolygon
- PRMan = premier system qui conformait à la spécification *Renderman*, développer par PIXAR
 - Actuellement un moteur hybrid de micropolygon et lancer de rayons
- REYES est l'algorithme le plus utilisé pour le rendu des films*



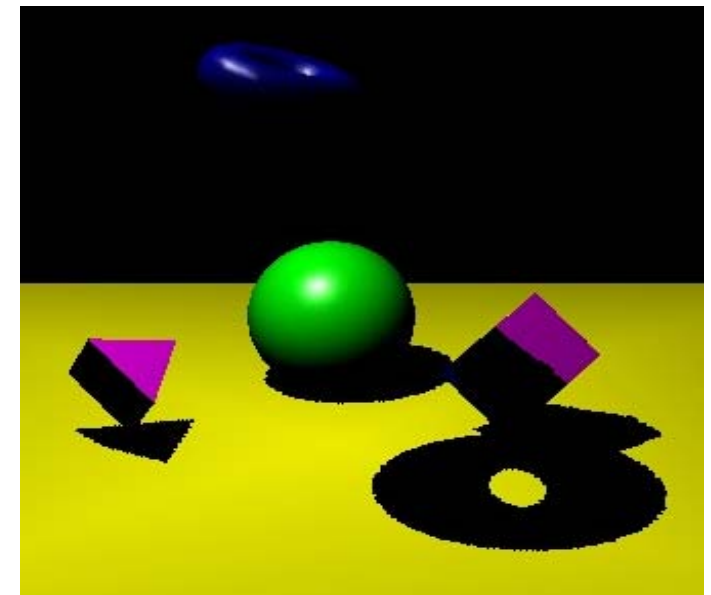
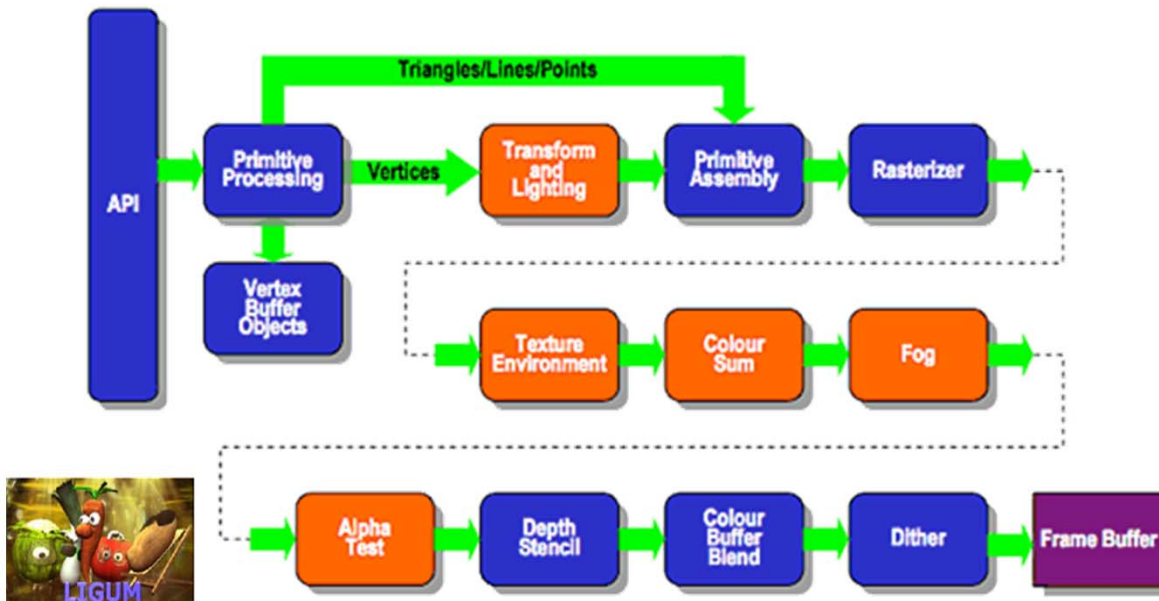
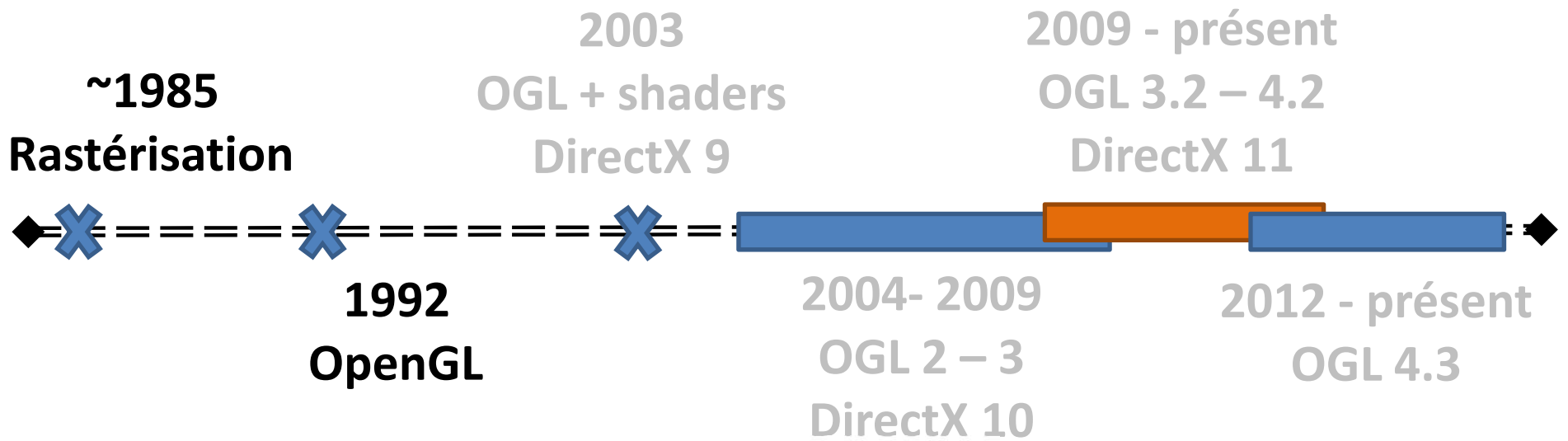
Systemes de rendu: une survol historique



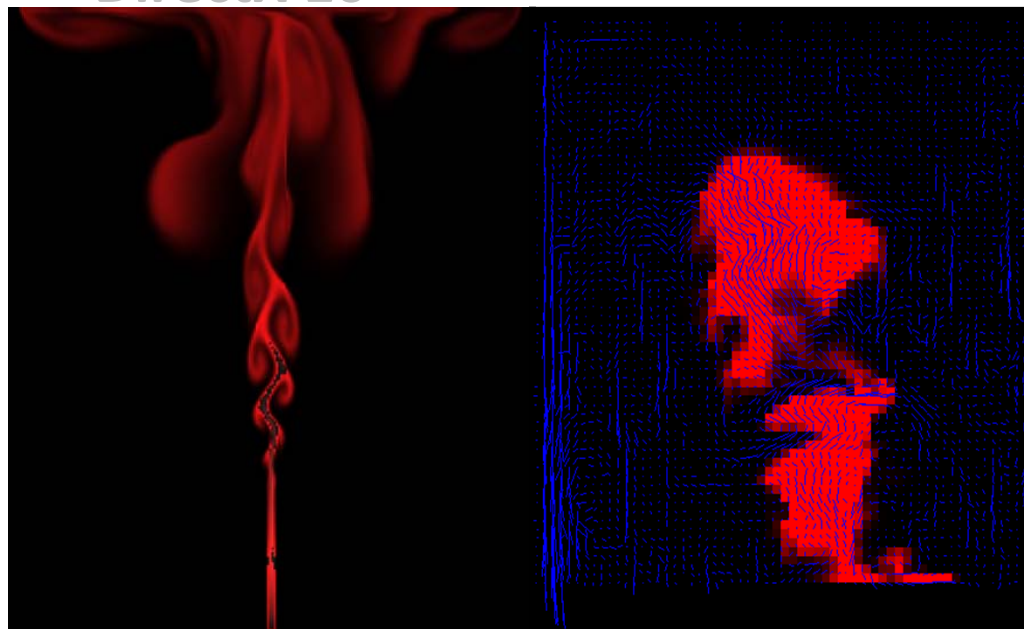
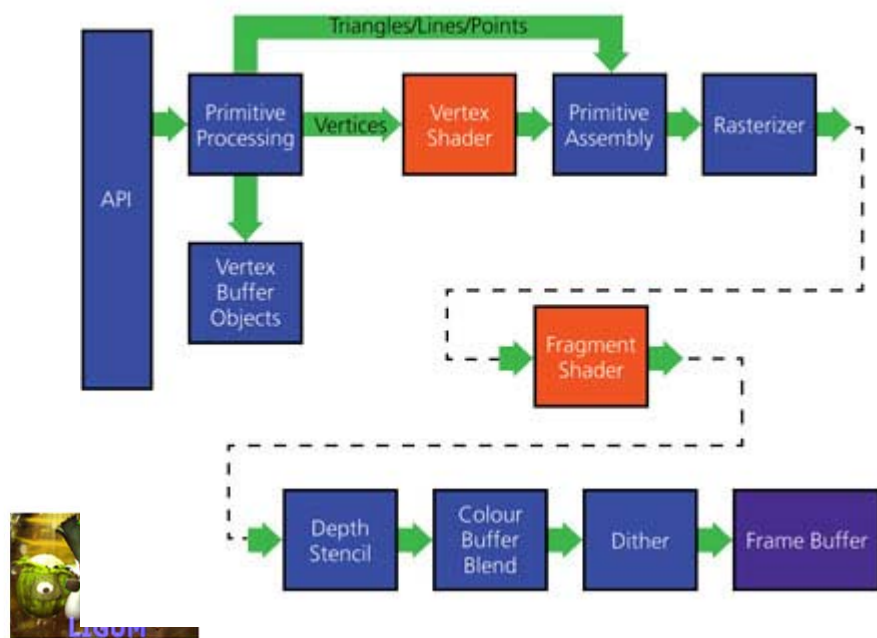
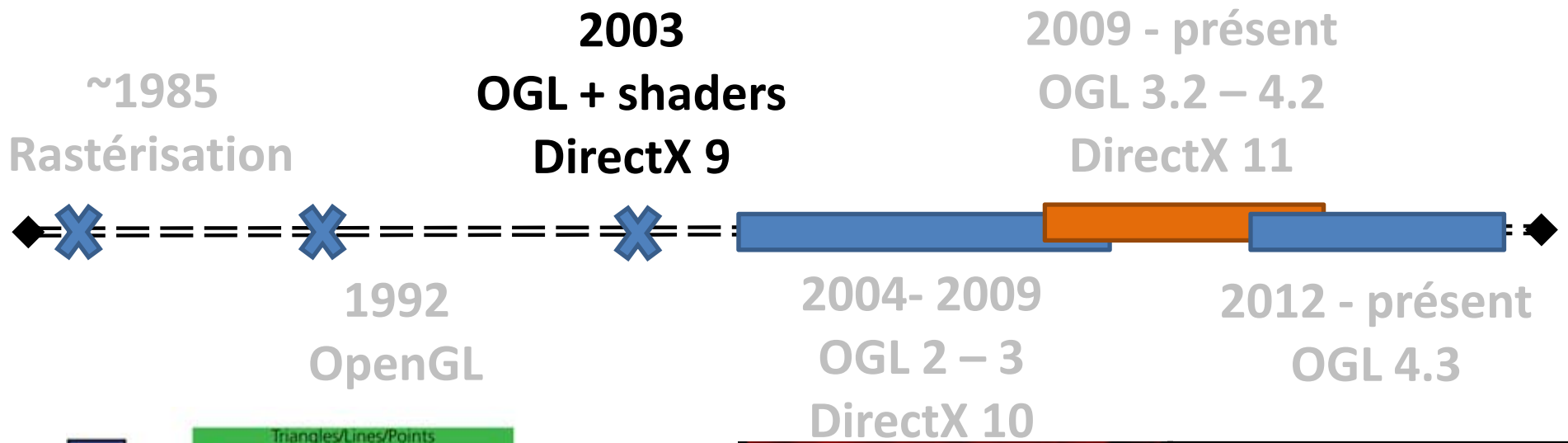
Lancer de rayons: évolution historique



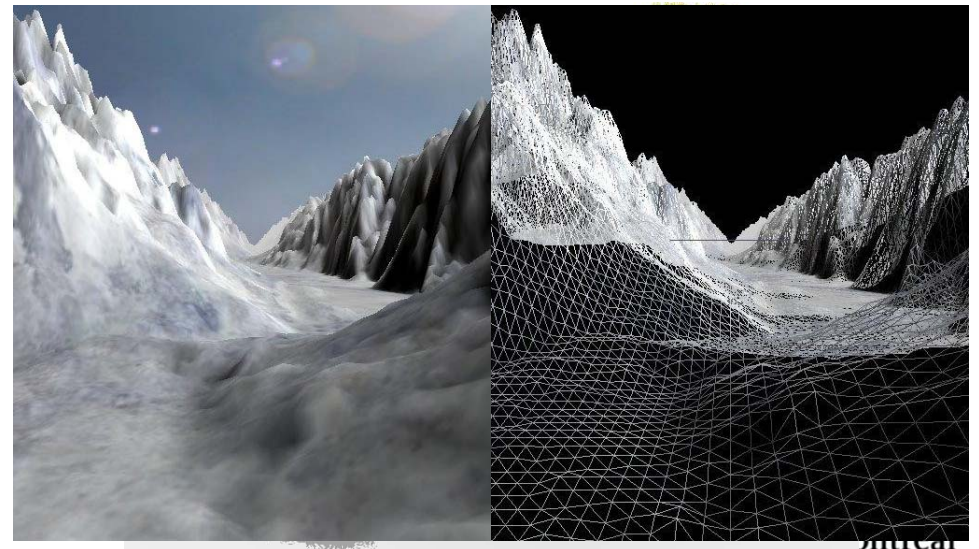
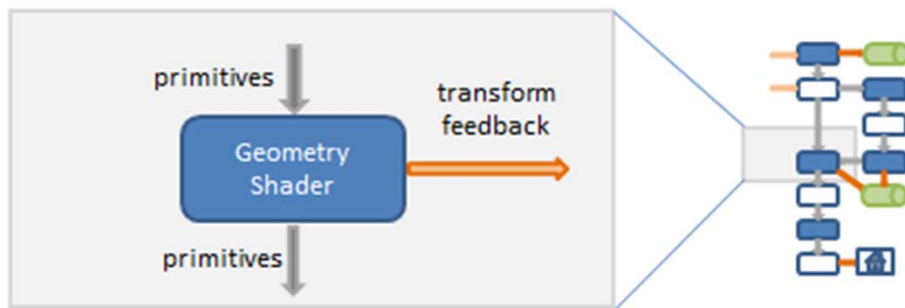
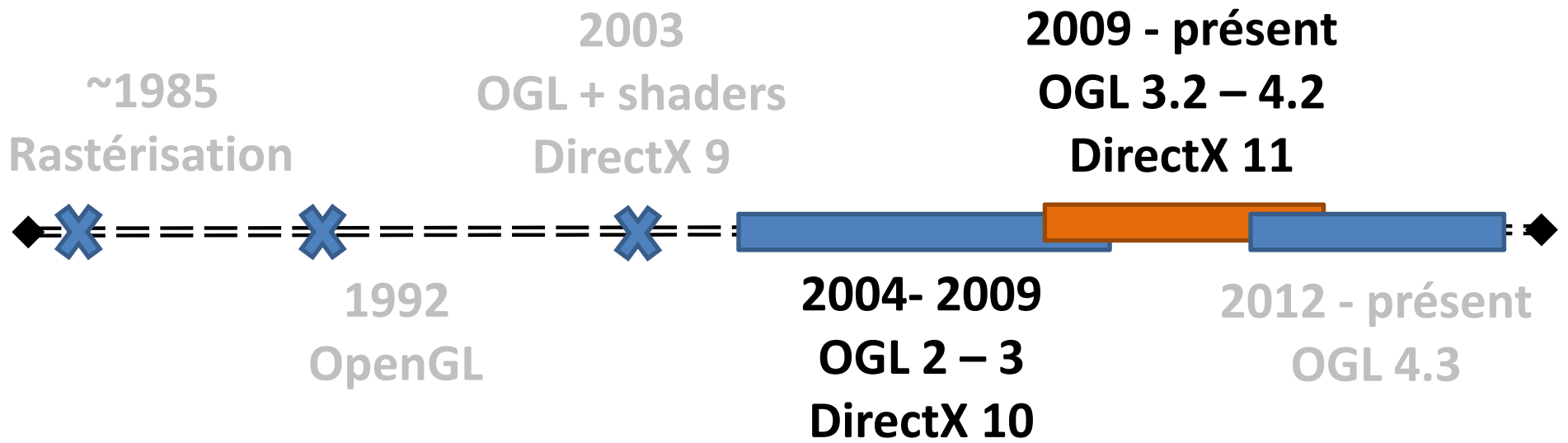
Rastérisation: évolution historique



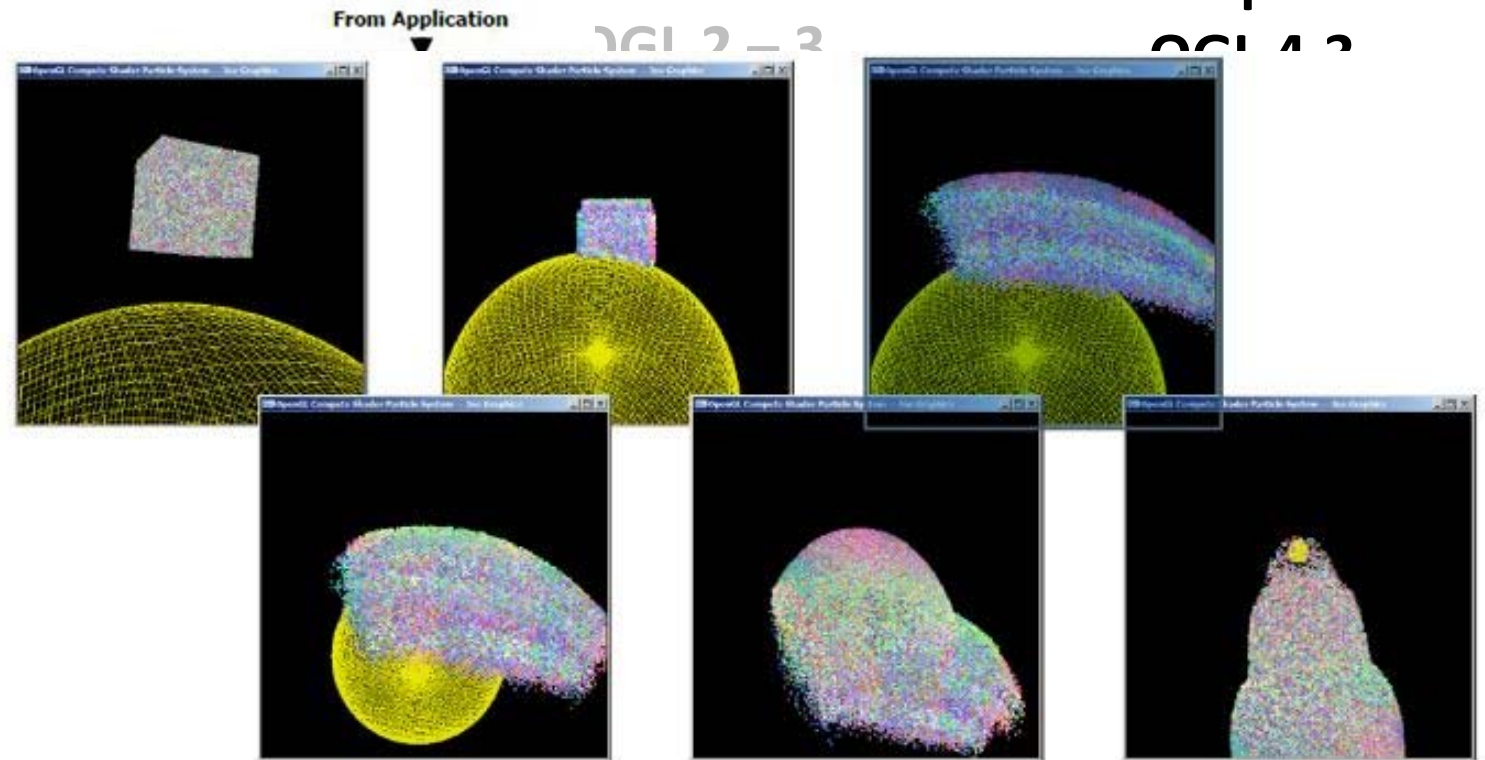
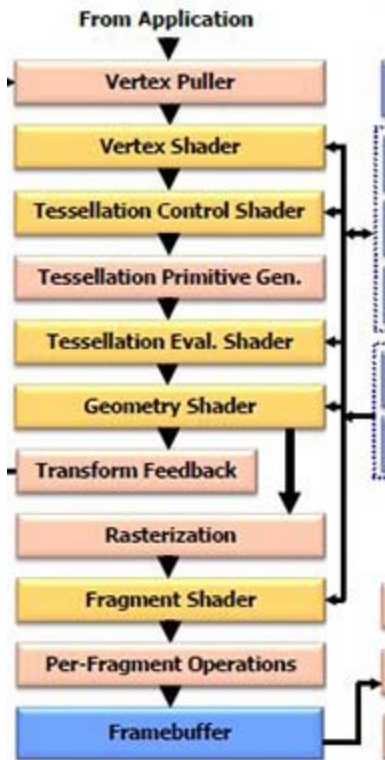
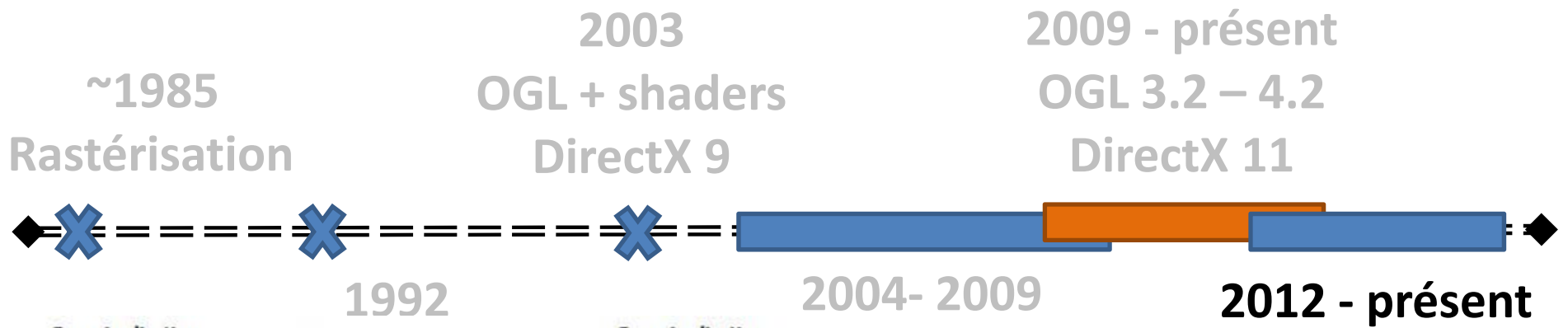
Rastérisation: évolution historique



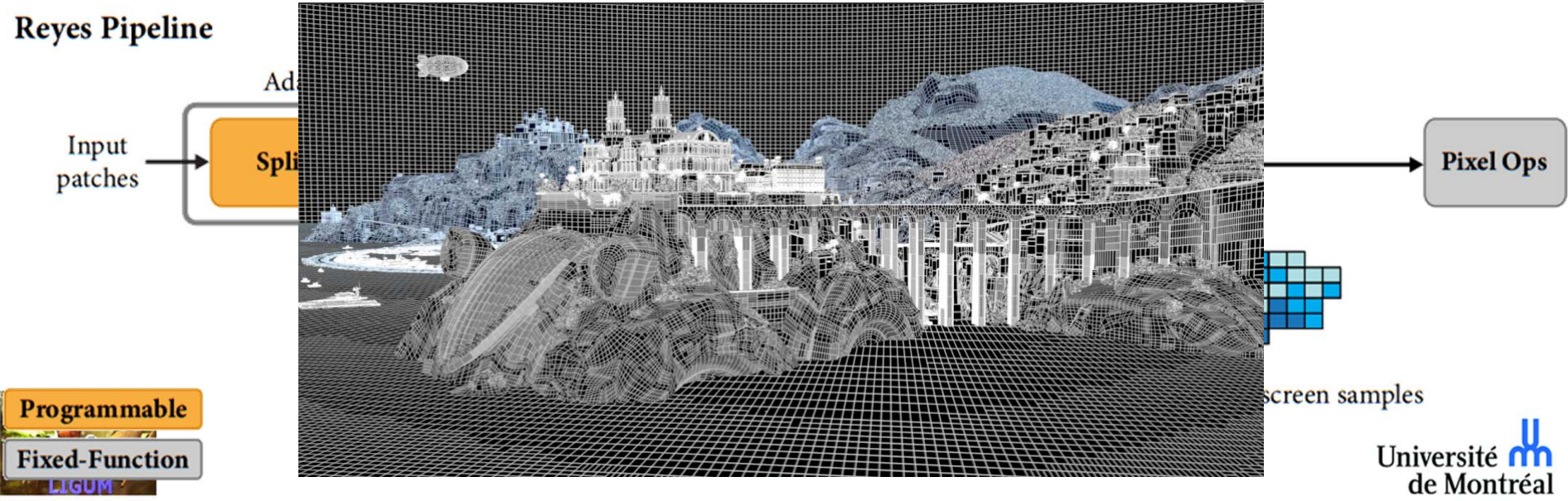
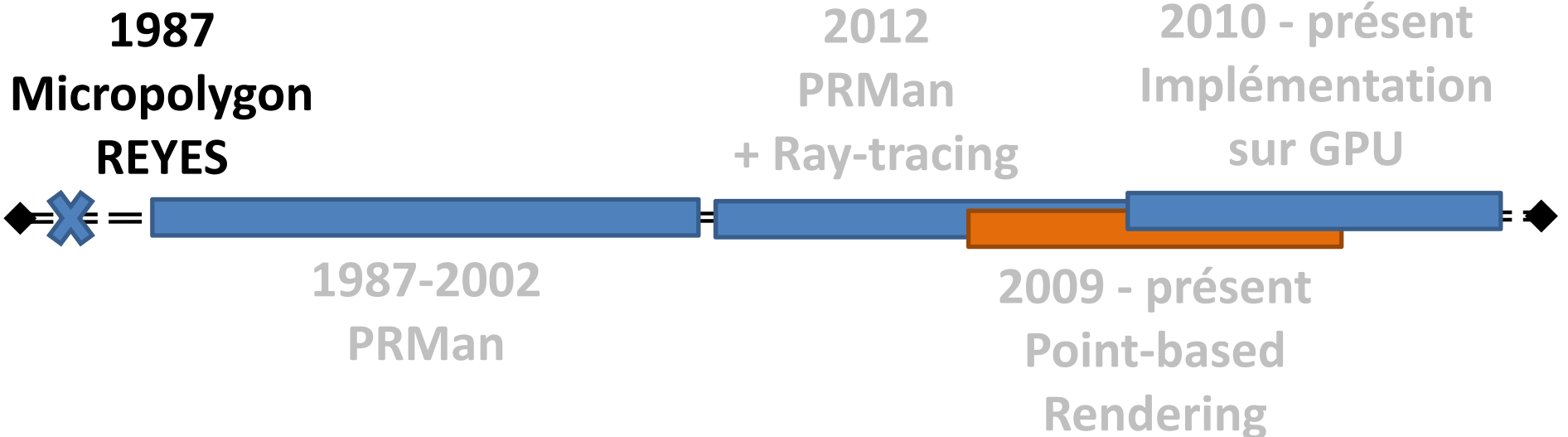
Rastérisation: évolution historique



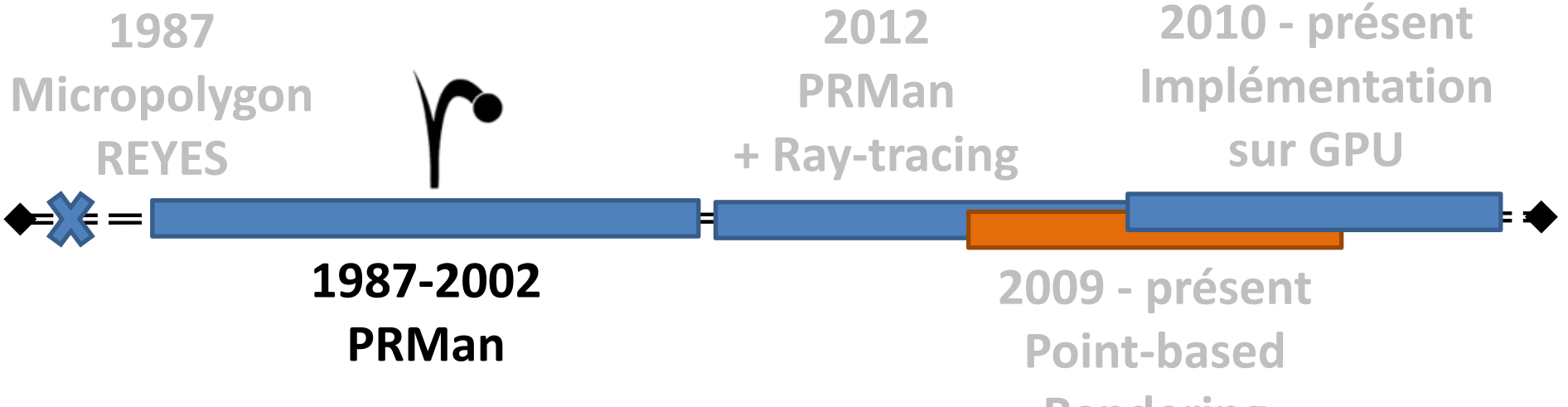
Rastérisation: évolution historique



Micropolygon: évolution historique



Micropolygon: évolution historique



23 Years of Pixar's RenderMan

"The original motivation to do the research that led to these technological discoveries was to expand the range of possibilities for storytellers."
Ed Catmull, President of Walt Disney and Pixar Animation Studios

THE COMMERCIAL YEARS
From 1990 to 1996 Pixar created many innovative 3D commercials for television, while developing its rendering technology — Photorealistic RenderMan.

1988 RI SPEC 3.0 PUBLISHED
With the first use of the word "RenderMan," the RenderMan Interface Specification was the culmination of years of Pixar's rendering development and set the standard for describing 3D data.

1989 RENDERMAN TOOLKIT 3.0
Stochastic Sampling
REYES Algorithm
Micropolygons

1990 RENDERMAN TOOLKIT 3.1
RenderMan Interface Bytestream (RIB)
Constructive Solid Geometry
Vector RenderMan

1991 RENDERMAN TOOLKIT 3.2
Pixar "Looks" Support
Zthreshold Shadow Opacity
Better Memory Utilization

1992 RENDERMAN TOOLKIT 3.3
"netrender"

1993 ACHIEVEMENT AWARD
Scientific & Engineering Achievement Award[®]
from the Academy of Motion Picture Arts and Sciences.

1994 RENDERMAN TOOLKIT 3.4
Trim Curves
Extreme Displacement
Vertex Motion blur

1995 RENDERMAN TOOLKIT 3.5
64 Bit Processor Support
Filterstep

1996 RENDERMAN TOOLKIT 3.6
RIB Archives
Vertex parameters
True RiSphere primitive

1997 RENDERMAN TOOLKIT 3.7
Procedural Primitives
RiPoints & RiCurves
Level of Detail

1998 RENDERMAN TOOLKIT 3.8
Subdivision Surfaces
Arbitrary Output Variables
DSO Shadeops

1999 RENDERMAN TOOLKIT 3.9
Bobby Implicit Surfaces
Accumulated Opacity Culling
Centered Derivatives

2000 RENDERMAN PRO SERVER 10.0
Facevarying Class Specifier
Conditional RI Evaluation
Non-Raster Oriented Dicing

2001 RENDERMAN PRO SERVER 11.0
Ray Tracing
Ambient Occlusion
Deep Shadow Maps

2002 RENDERMAN PRO SERVER 12.0
3D BrickMaps
OpenEXR Support
Ri Filters

2003 RENDERMAN PRO SERVER 12.5
Scene Analysis Acceleration
Hider Subpixel Output
Deep Shadow API

2004 RENDERMAN PRO SERVER 13.0
Multi-Threaded Rendering
Brick Maps as Geometry
Point Based Colorbleeding

2005 RENDERMAN PRO SERVER 12.5
Accelerated Ray Tracing
Hierarchical Subdivs
Point Cloud API

2006 RENDERMAN PRO SERVER 13.0
Stereo Multi-Camera
Shader Objects
Multi-Threaded

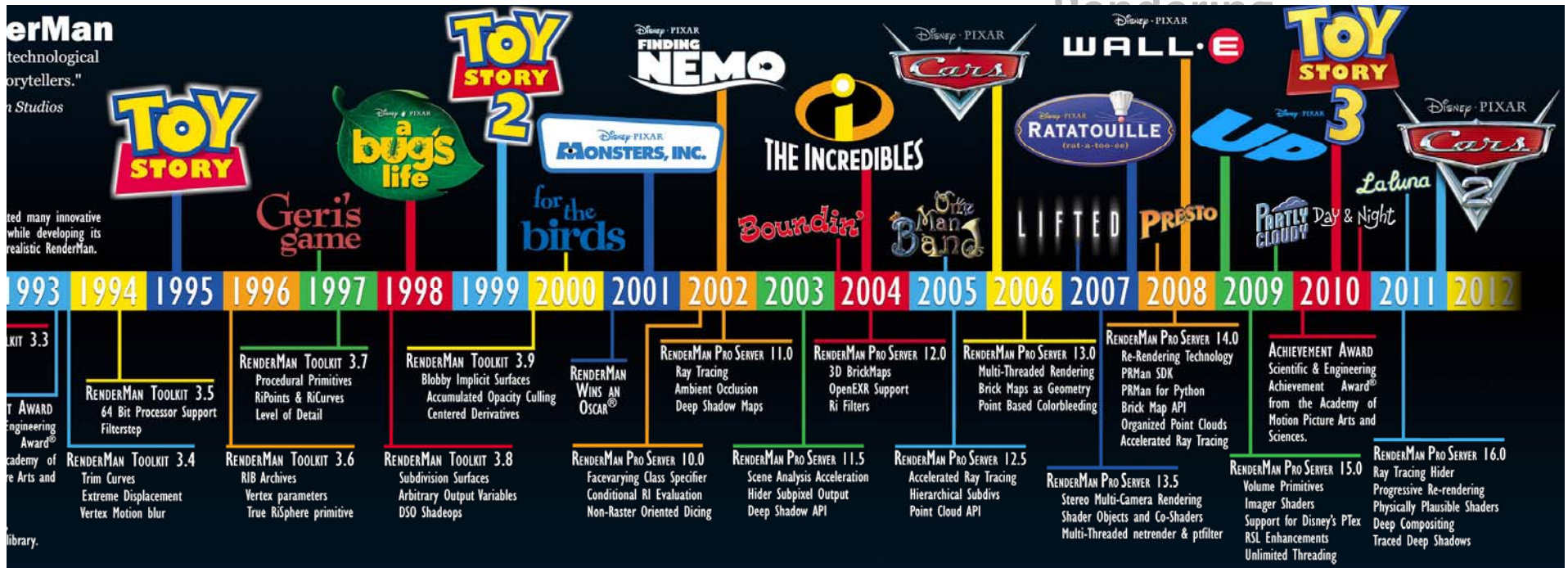
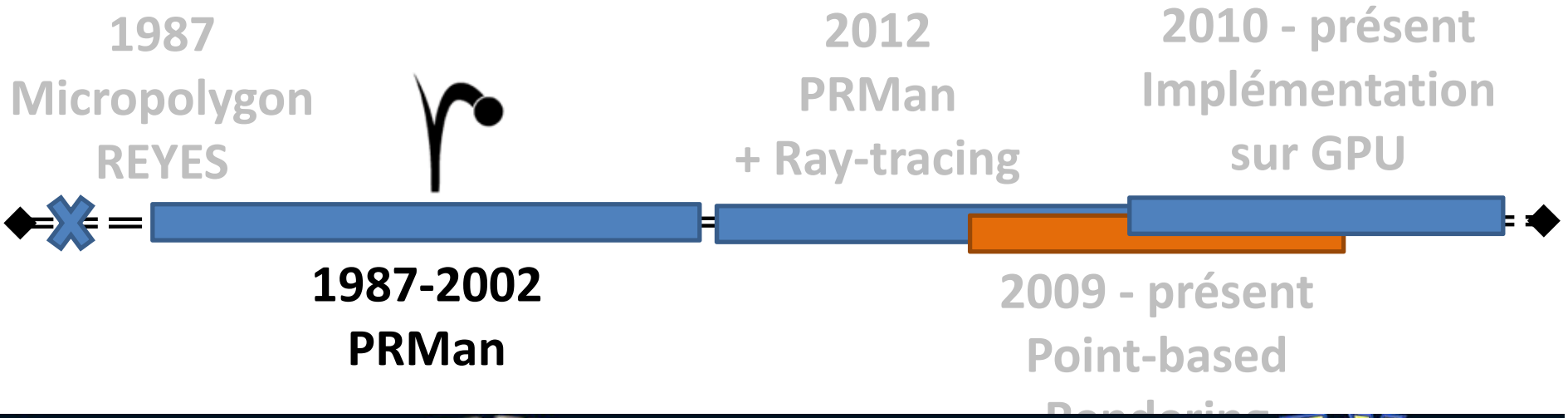
2007 RENDERMAN PRO SERVER 13.0
Stereo Multi-Camera
Shader Objects
Multi-Threaded

TOY STORY
Toy Story 2
Monsters, Inc.
for the birds
Boundin'
Man of Steel
LIFTED

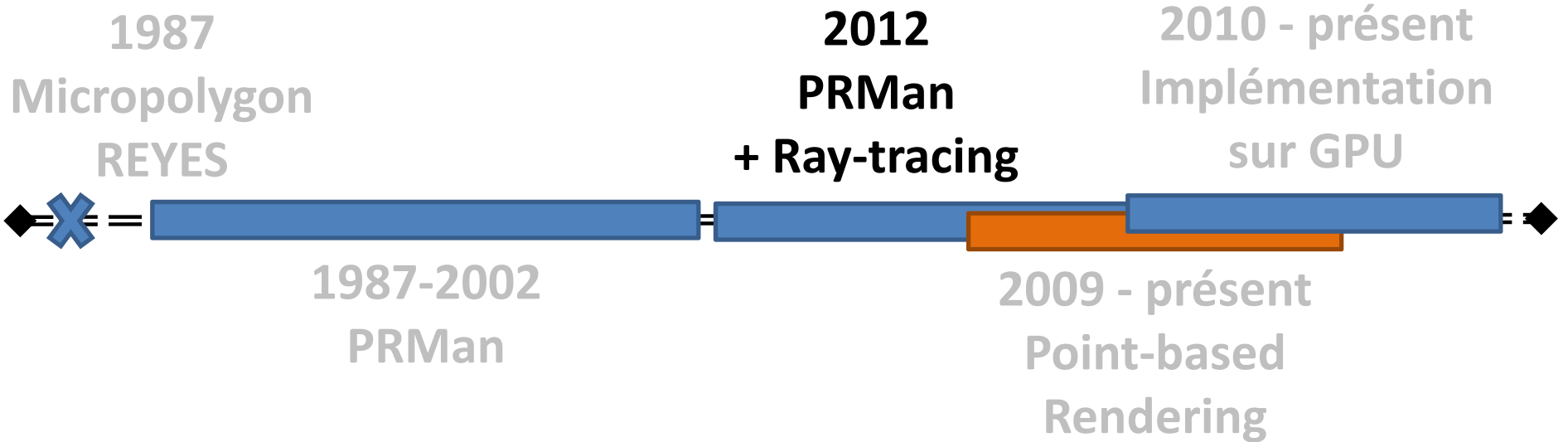
Disney Pixar
Geris game
Disney Pixar
bugs life
Disney Pixar
MONSTERS, INC.
Disney Pixar
NEMO
Disney Pixar
THE INCREDIBLES
Disney Pixar
Cars
Disney Pixar
RATATOUIL
(1988-01-01-00-00)

TIN TOY
knickknack

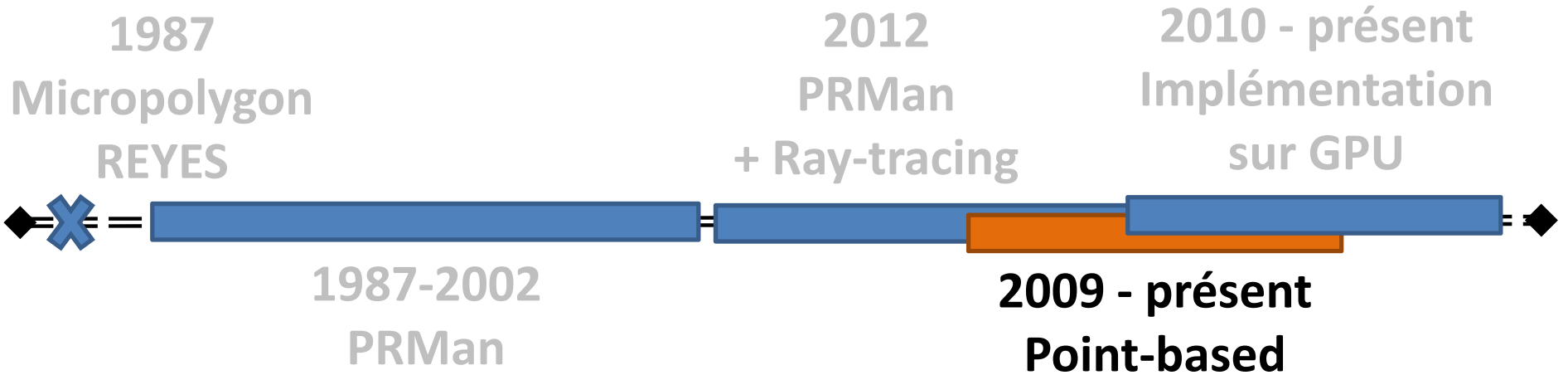
Micropolygon: évolution historique



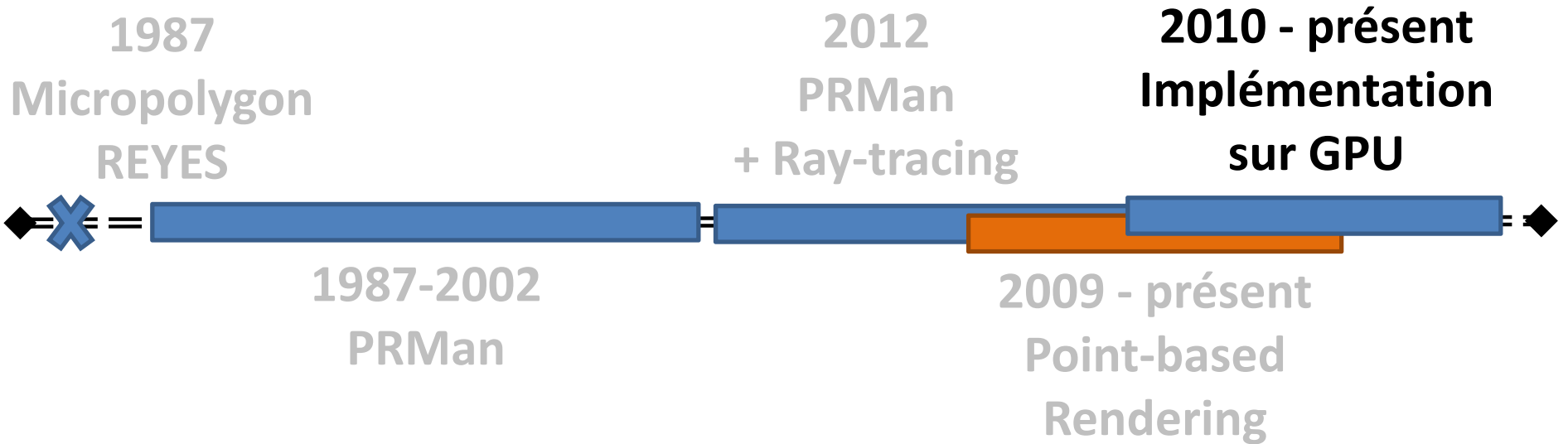
Micropolygon: évolution historique



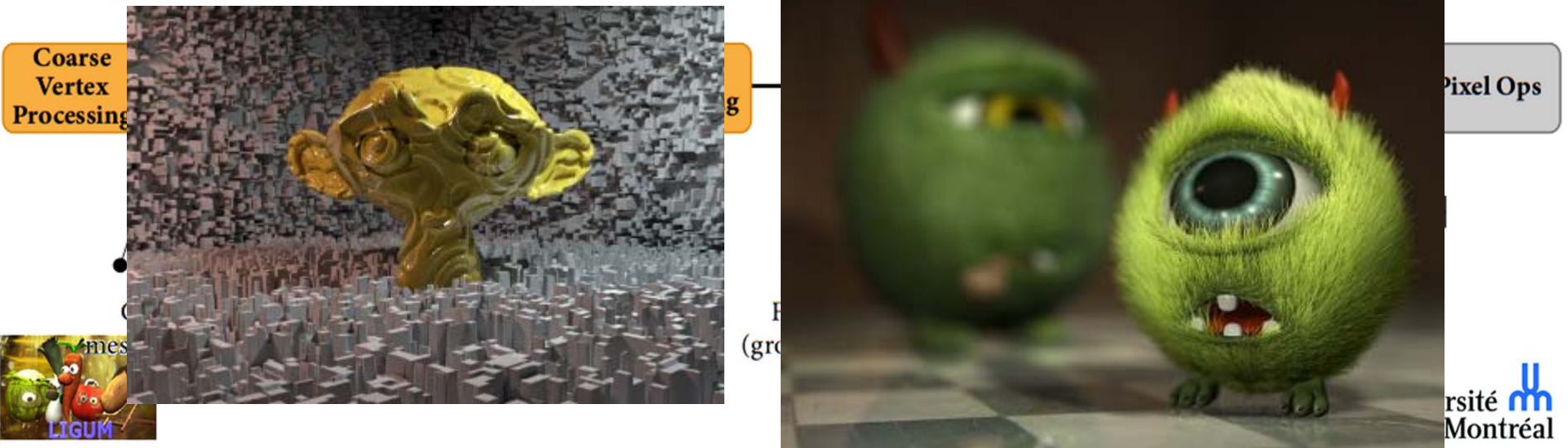
Micropolygon: évolution historique



Micropolygon: évolution historique



GPU Pipeline



Débat religieux: croyez-vous en Rayons ou en Rastériseur?

- Question piège: ça dépend sur le bût de ton système, les contraintes, les types d'effets, etc.

<http://c0de517e.blogspot.ca/2011/09/raytracing-myths.html>

<https://plus.google.com/105941772928736706414/posts/TFm2kjThKSF>

<http://www.dailytech.com/John+Carmack+on+Ray+Tracing+vs+Rasterization/article11095.htm>

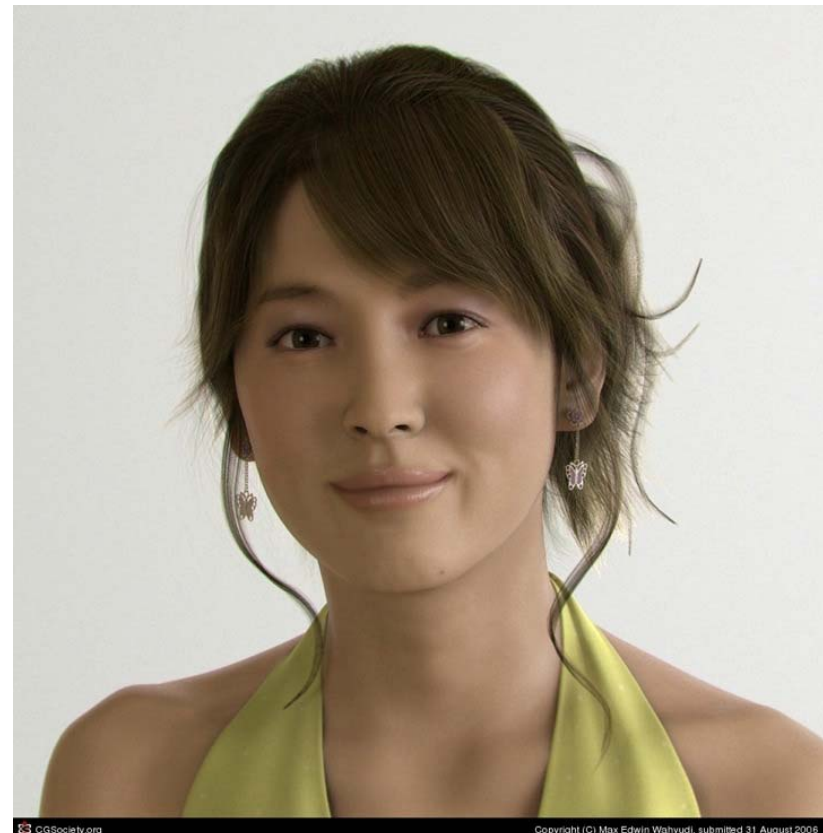
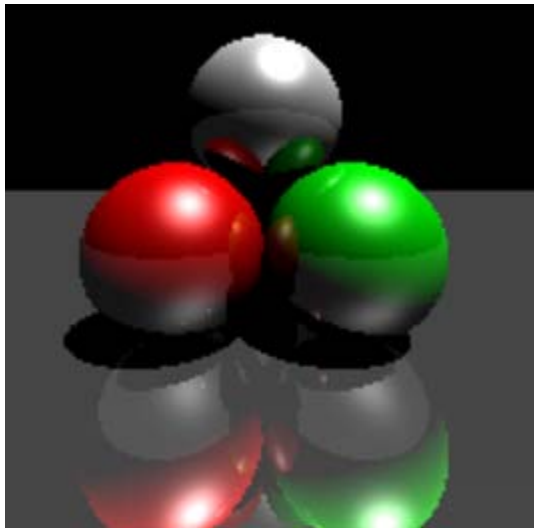
<http://www.scarydevil.com/~peter/io/raytracing-vs-rasterization.html>

- Quels sont les tendances actuels en industrie?
- ... et en recherche académique?



Évolution et extensions

- Chaque architecture a évolué de leur implémentation de base au fil de temps afin de supporter de plus en plus de fonctionnalité



CGSociety.org

Copyright (C) Max Edwin Wahyudi, submitted 31 August 2006

Évolution et extensions

- Chaque architecture a évolué de leur implémentation de base au fil de temps afin de supporter de plus en plus de fonctionnalité



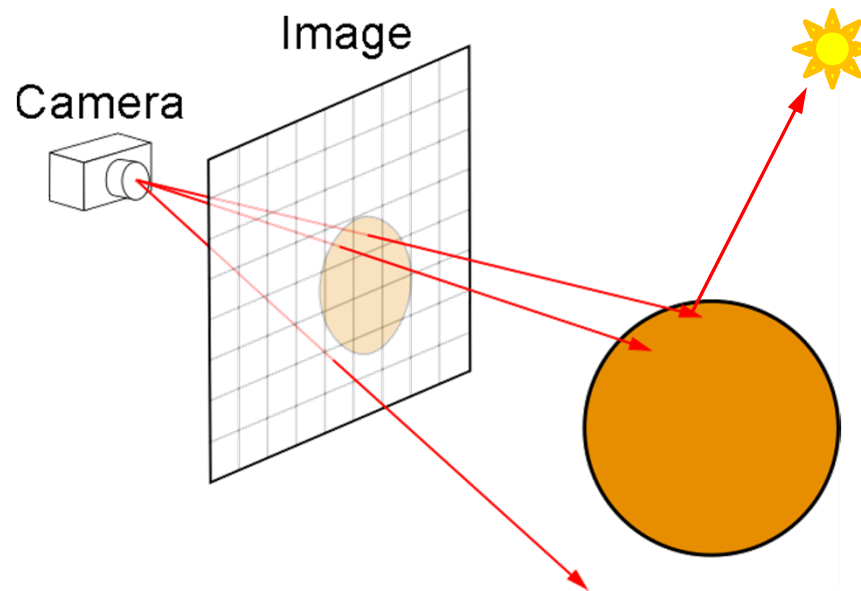
Évolution et extensions

- Chaque architecture a évolué de leur implémentation de base au fil de temps afin de supporter de plus en plus de fonctionnalité



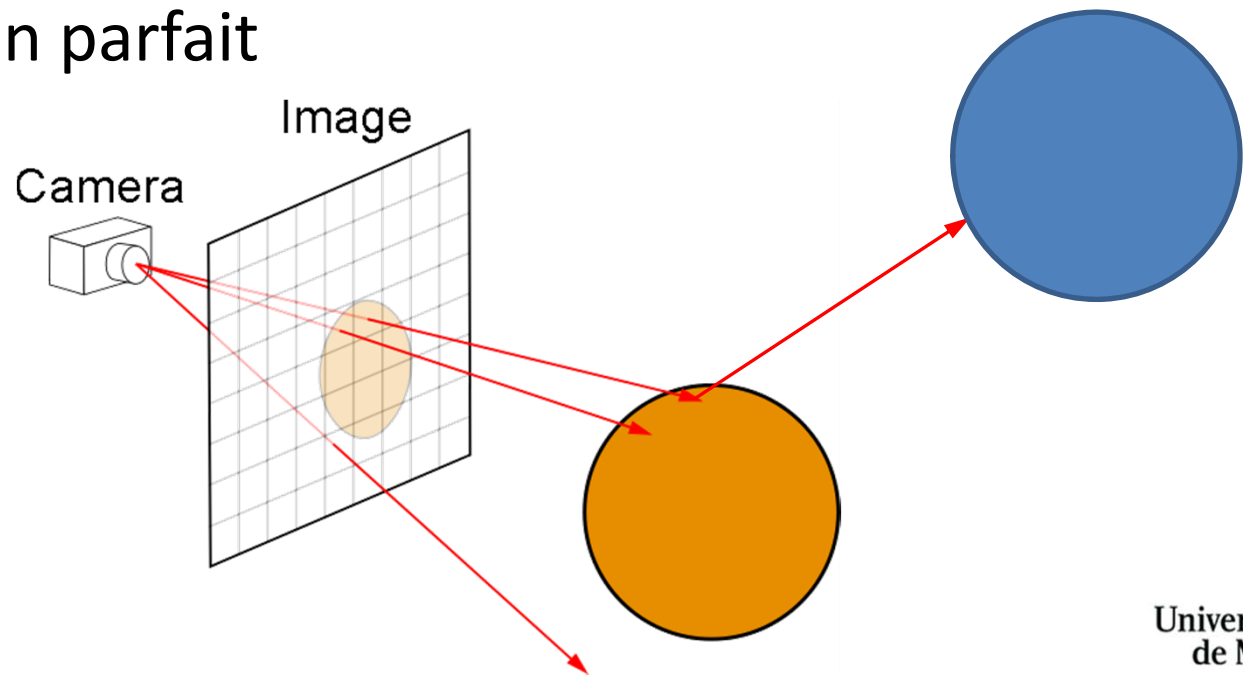
Évolution & extensions: lancer de rayons

- Un des premier extensions dans l'évolution des lanceurs de rayons est le système de Whitted
 - Fournit une seule couche de recursion, permettant la simulation des ombres durs et des réflexion / réfraction parfait



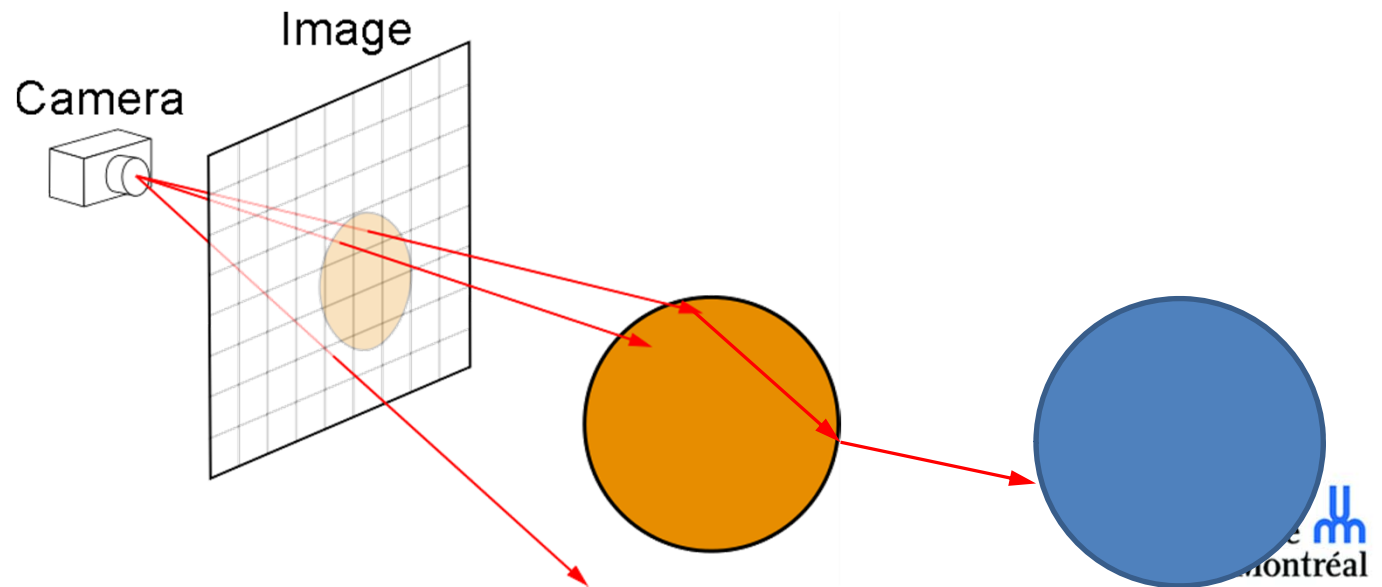
Évolution & extensions: lancer de rayons

- Un des premier extensions dans l'évolution des lanceurs de rayons est le système de Whitted
 - Fournit une seule couche de recursion, permettant la simulation des ombres durs et des réflexion / réfraction parfait



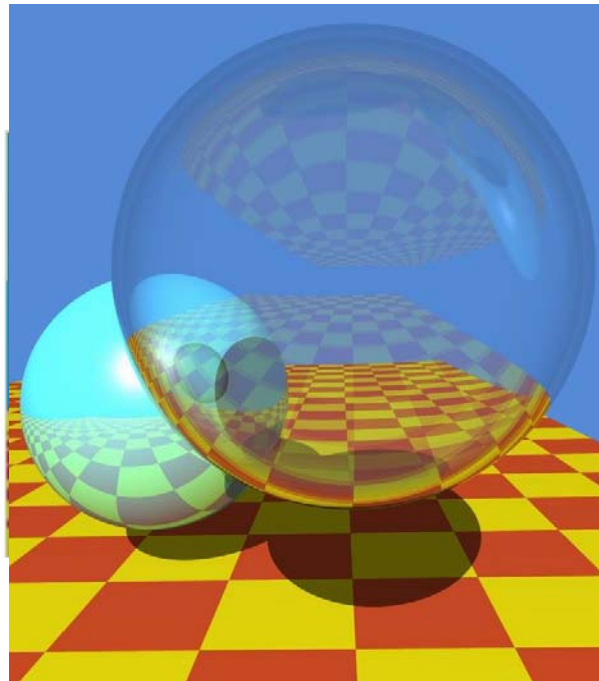
Évolution & extensions: lancer de rayons

- Un des premier extensions dans l'évolution des lanceurs de rayons est le système de Whitted
 - Fournit une seule couche de recursion, permettant la simulation des ombres durs et des réflexion / réfraction parfait



Évolution & extensions: lancer de rayons

- Un des premier extensions dans l'évolution des lanceurs de rayons est le système de Whitted
 - Fournit une seule couche de recursion, permettant la simulation des ombres durs et des réflexion / réfraction parfait



Évolution & extensions: lancer de rayons

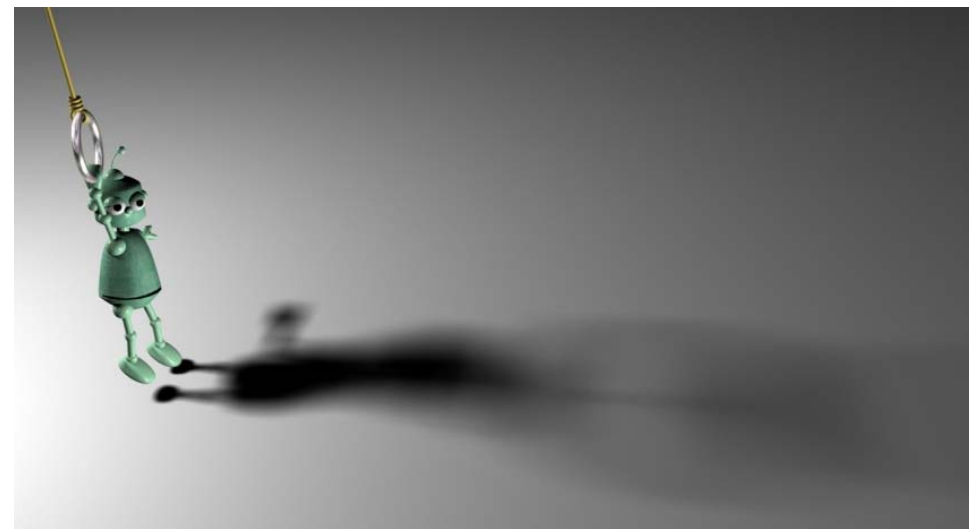
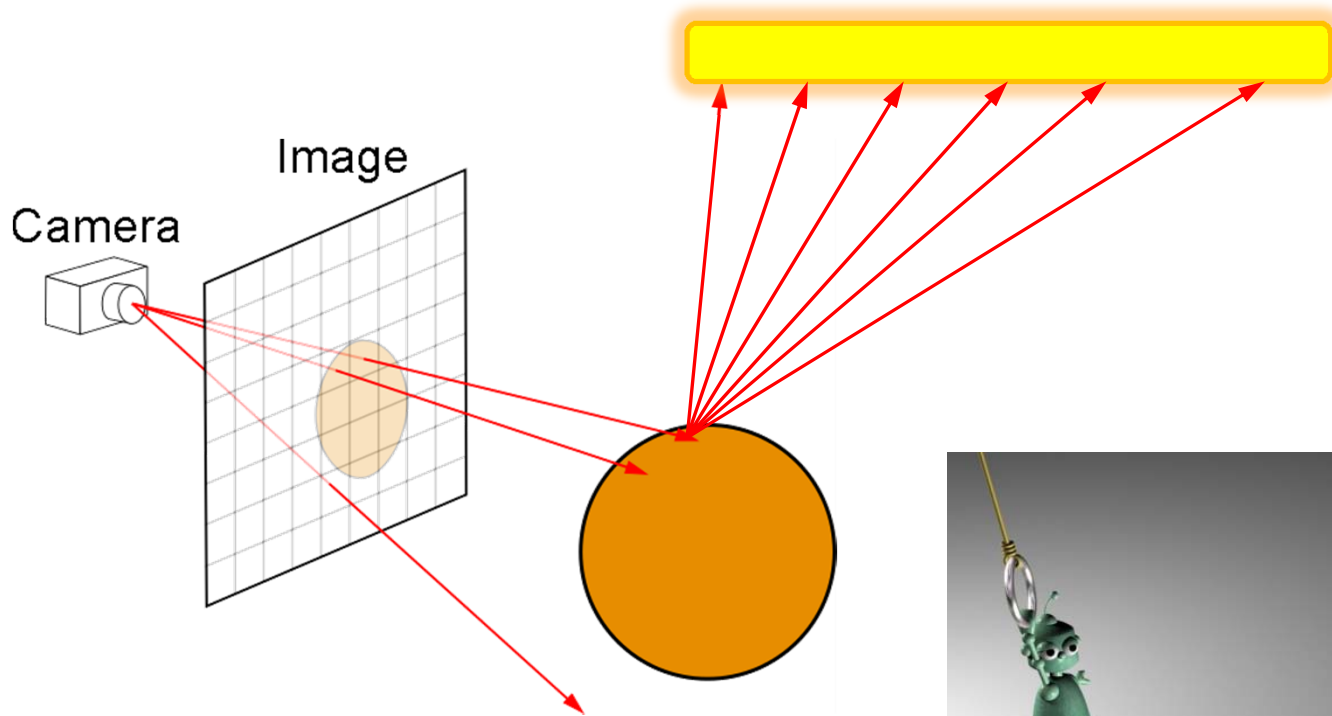
- Il a fallu plusieurs années avant que les gens ont non-seulement formulé l'équation de rendu mais aussi noter qu'un lanceur de rayons pouvait servir à le résoudre*
 - (sous forme d'un estimateur Monte Carlo)
- Plusieurs effets d'illumination avancés peuvent être **facilement** implémentés avec un arbre de rayons récursif et/ou distribué ...



(et alors dans un lanceur de rayons)

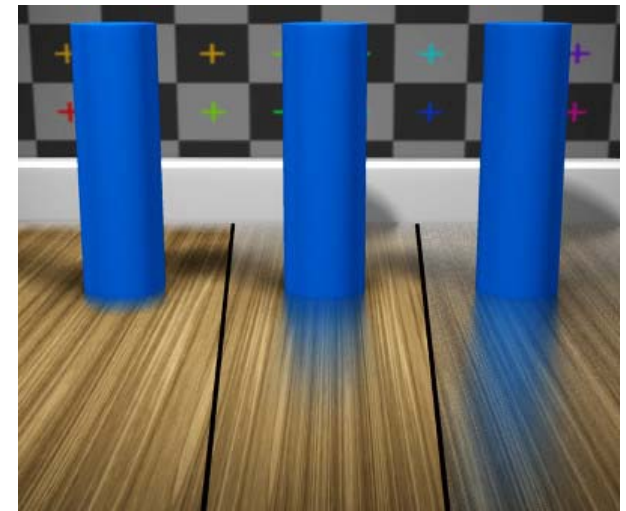
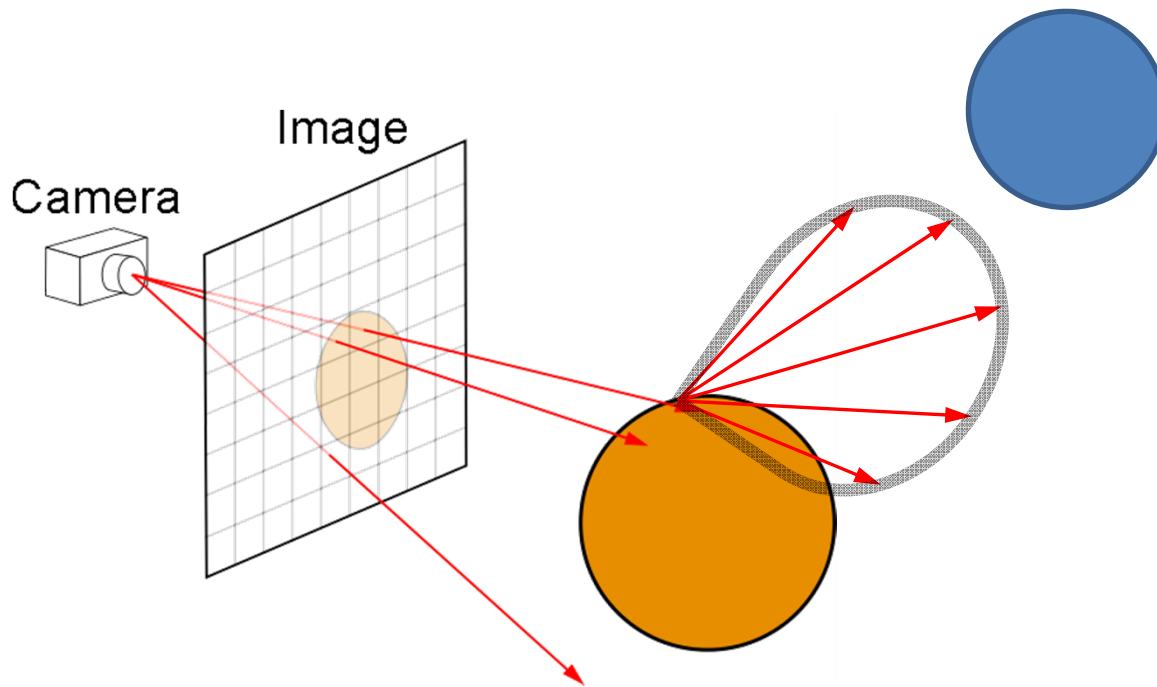
Évolution & extensions: lancer de rayons

- Ombres étendues



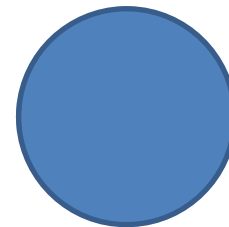
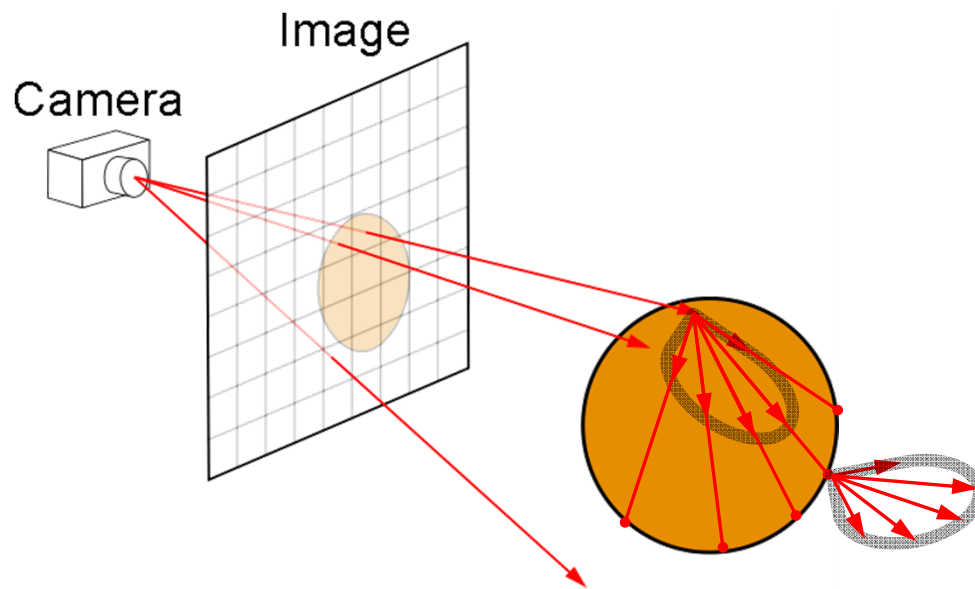
Évolution & extensions: lancer de rayons

- Réflexion *glossy* (distribué)



Évolution & extensions: lancer de rayons

- Transmission/réfraction *glossy* (distribué)

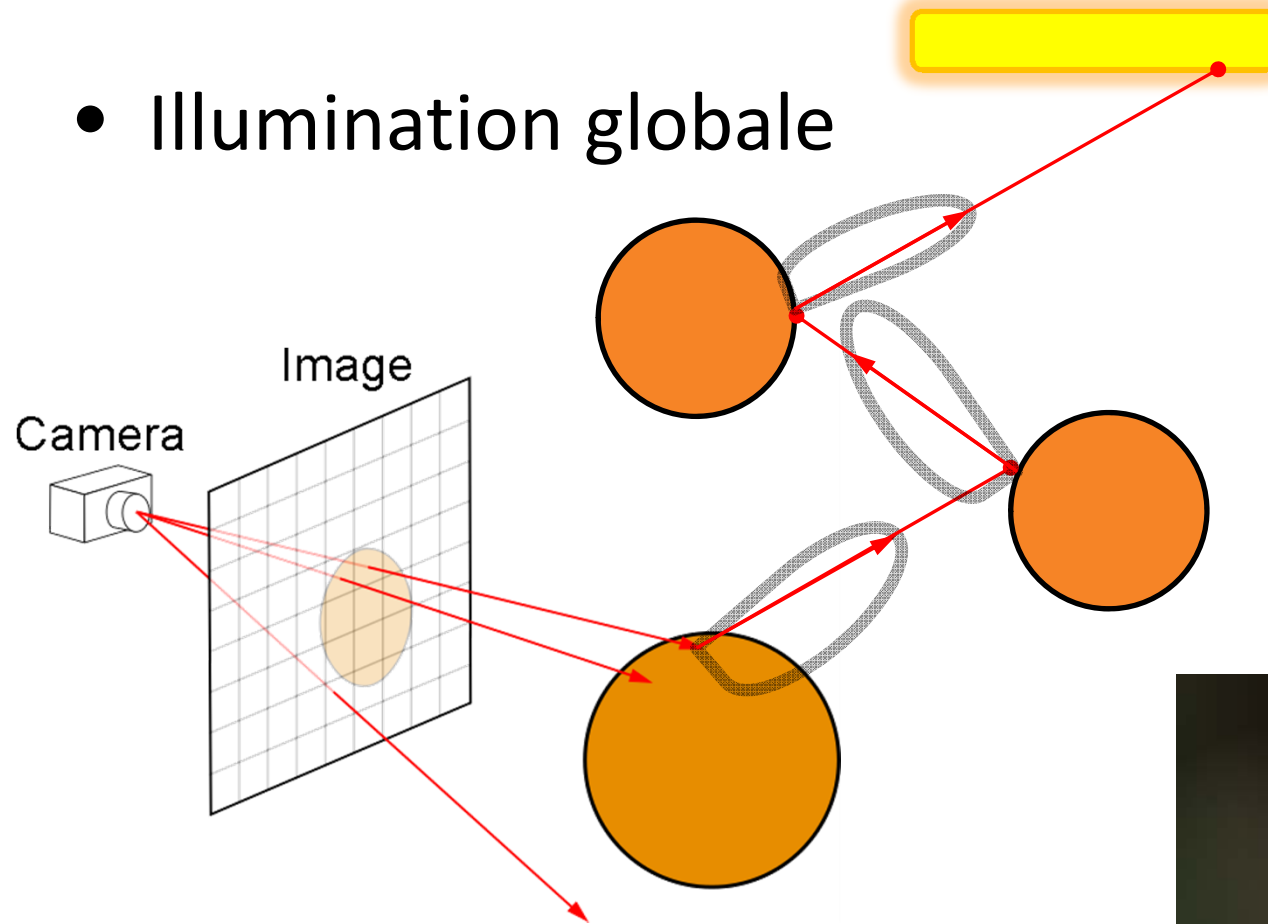


de Montréal



Évolution & extensions: lancer de rayons

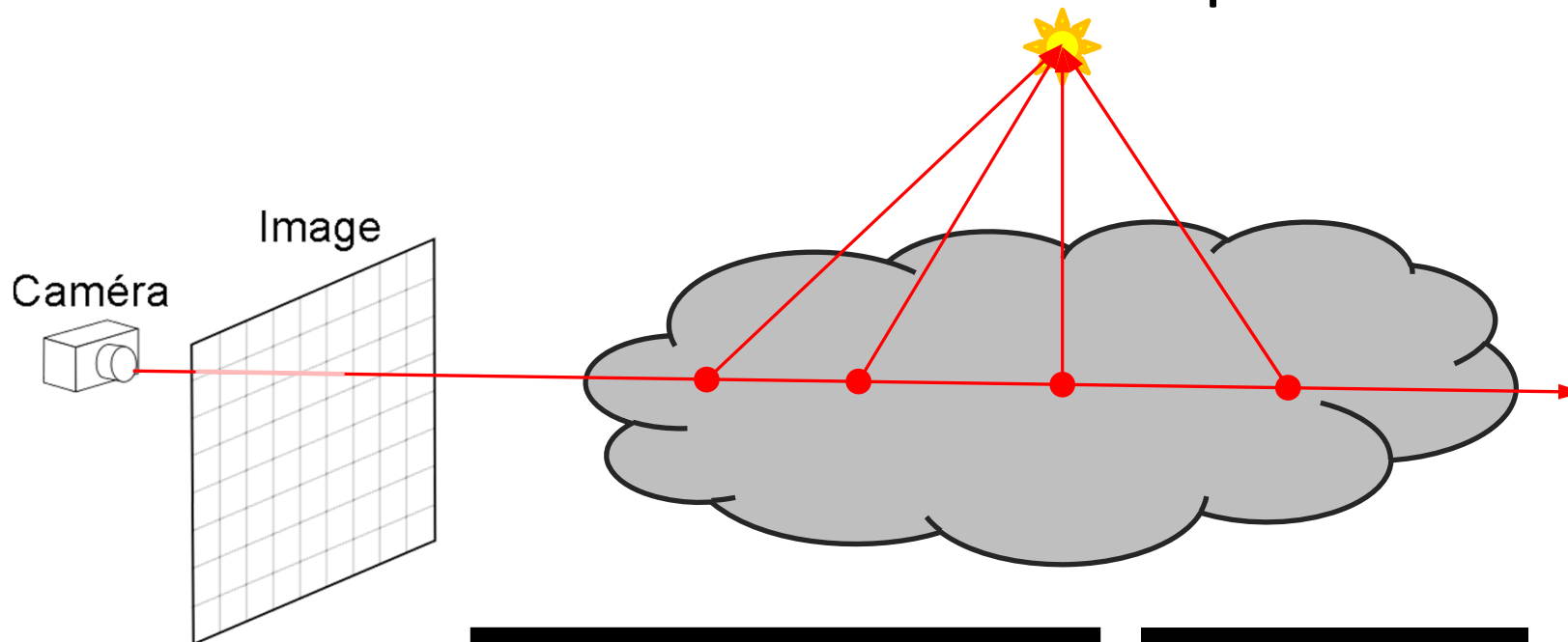
- Illumination globale



de Montreal

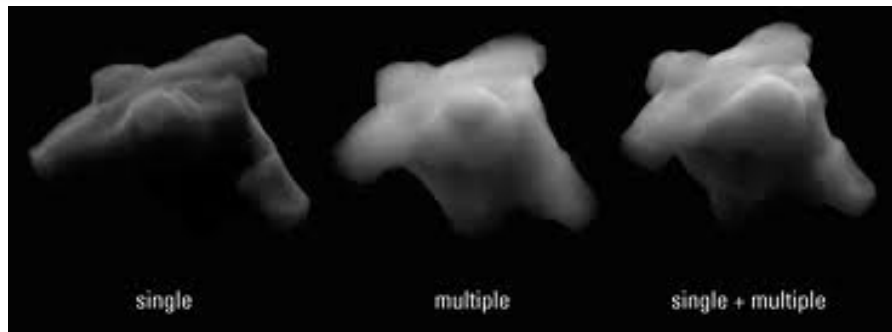
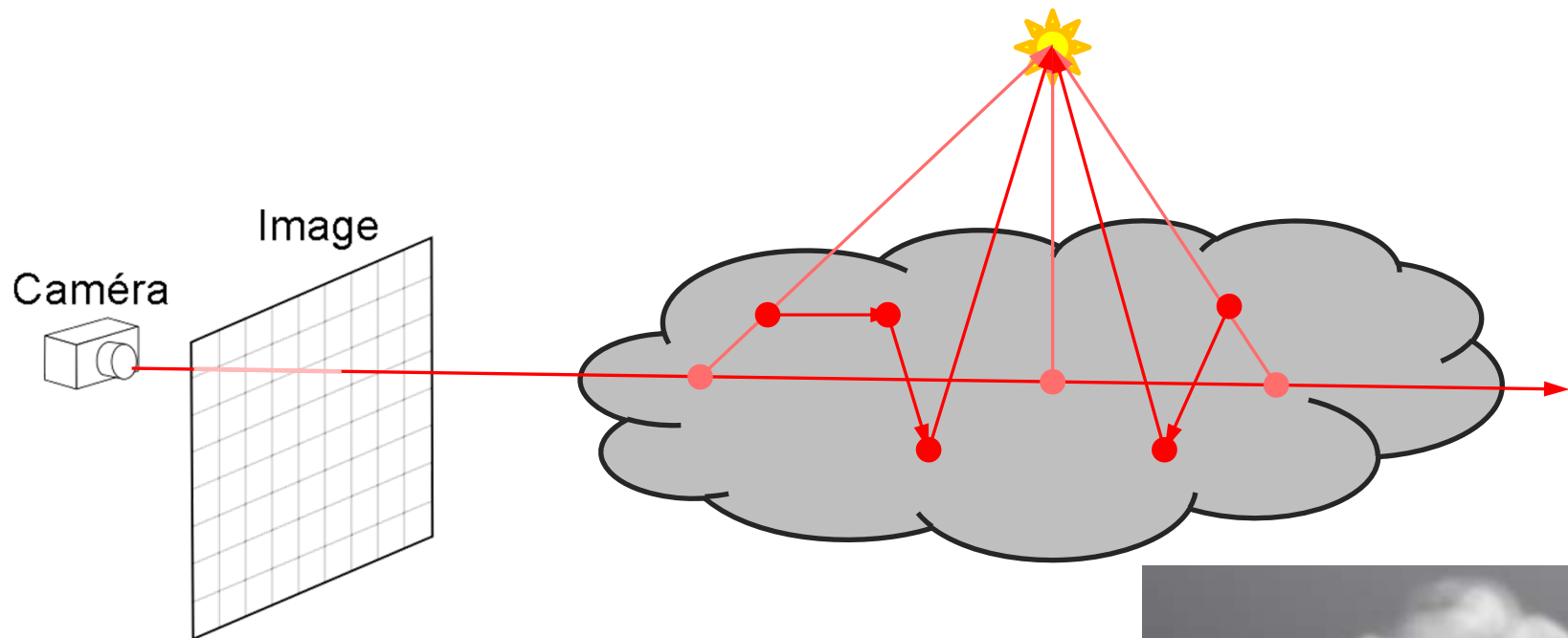
Évolution & extensions: lancer de rayons

- Illumination directe volumétrique



Évolution & extensions: lancer de rayons

- Illumination globale volumétrique



Le future: lancer de rayons

- Comme l'incorporation l'illumination complexes est plus simple dans ce système, un des défis fondamentaux se concerne avec les calculs d'intersection:
 - Quand la scène change dynamiquement
 - Quand la géométrie s'approche une résolution sous-pixel
 - Quand la cohérence en espace rayons n'est maintenu
 - Les implémentations sur HW
- Celui-ci comprend la conception des nouveaux structures d'accélération. Par exemple, ...
- Les améliorations aux modèles mathématiques d'illumination sont souvent facile à intégrer dans un lanceur de rayons



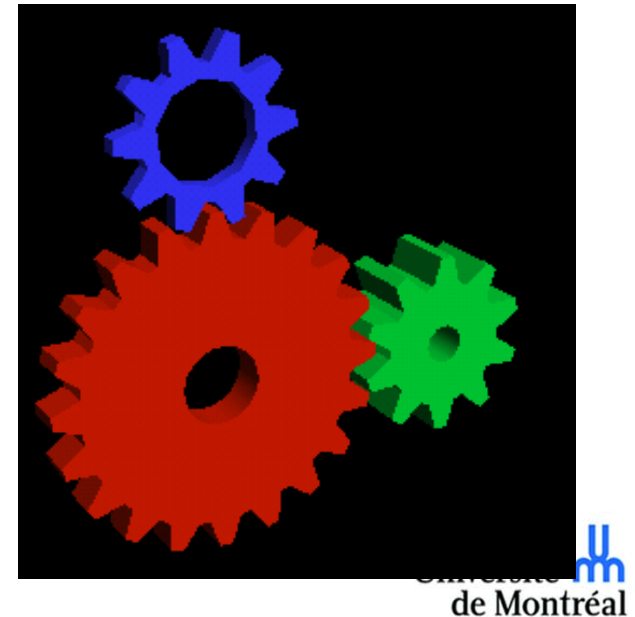
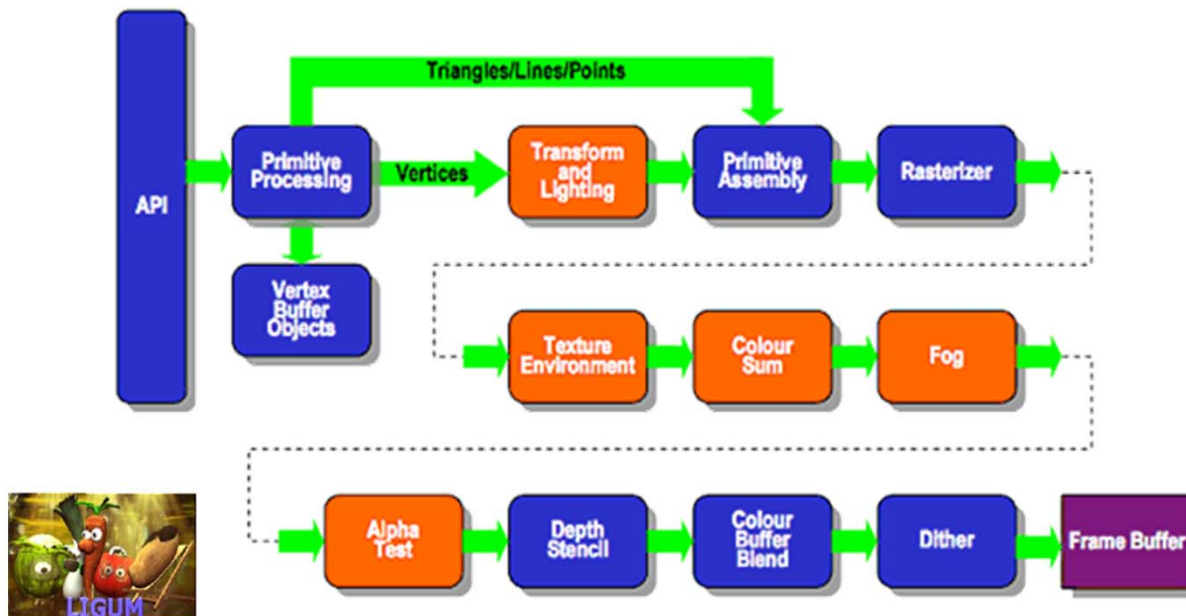
Évolution & extensions: rastérisation

- Un système de rastérisation de base doit convertir entre une représentation vectorielle de la scène à un représentation *raster*/image en:
 - Appliquant des transformations pour apporter tous les objets 3D en espace monde
 - Suivi par un transformation de projection pour les projeter sur le plan de vue
 - L'application des algorithmes de clipping (aussi faisable en espace non-projeter)
 - Et finalement le remplissage et les tests de visibilité



Évolution & extensions: rastérisation

- Le premier système de rastérisation standardiser était celle de OpenGL v1 introduit par SGI en 1992
- Le spécification pouvait être réalisé en logiciel ou sur HW selon les étapes **fixed** suivants:

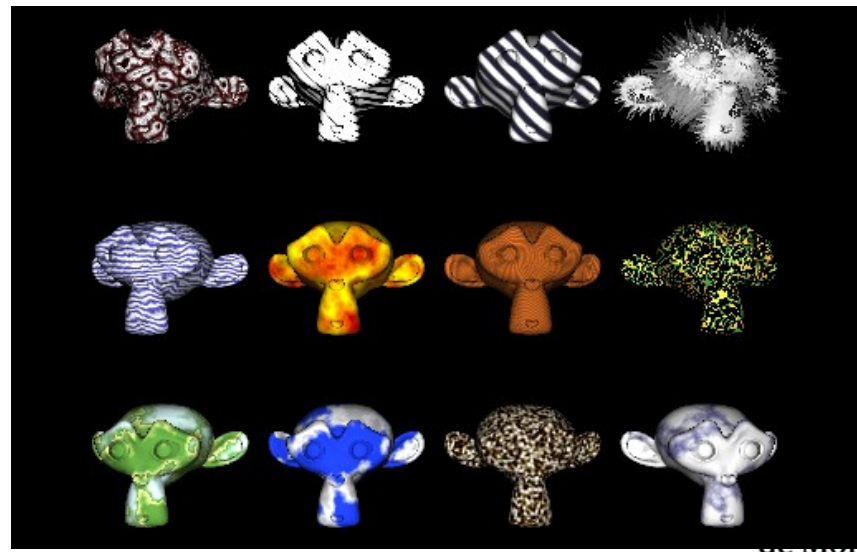
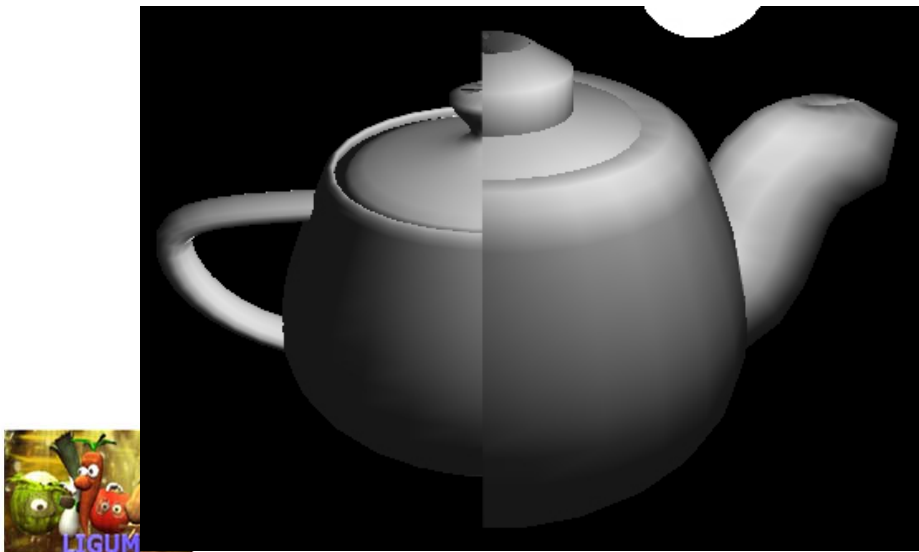
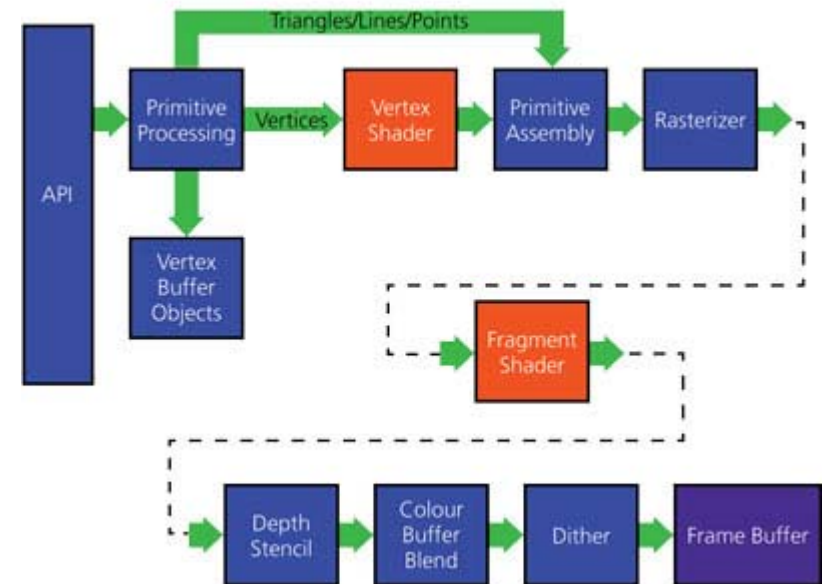
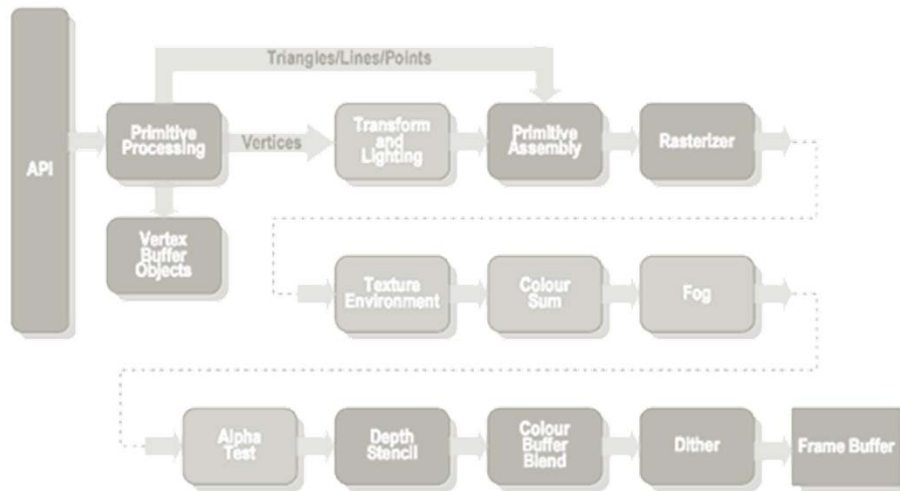


Évolution & extensions: rasterisation

- Le succès des composants programmable dans le pipeline REYES a promu l'utilisation des *texture combiners* pour le calcul des effet faiblement programmable, et cela servira comme catalyseur pour l'évolution du pipeline OGL
- OpenGL v1.5 introduit les premières composantes programmable dans son pipeline:
 - Les vertex shaders et les pixel (fragment) shaders
 - Ces deux composantes peuvent facilement être parallélisé



Évolution & extensions: rasterisation

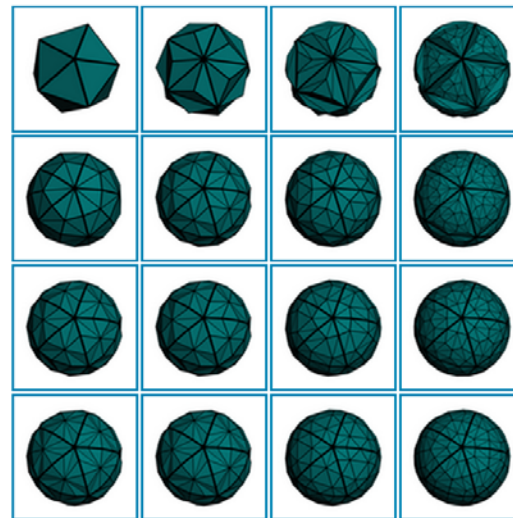
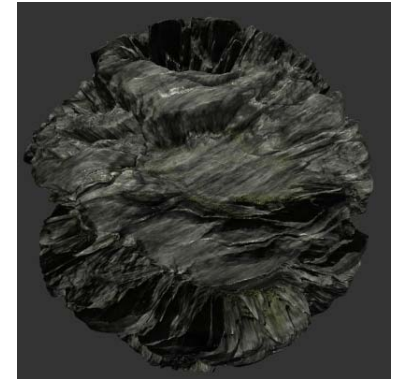
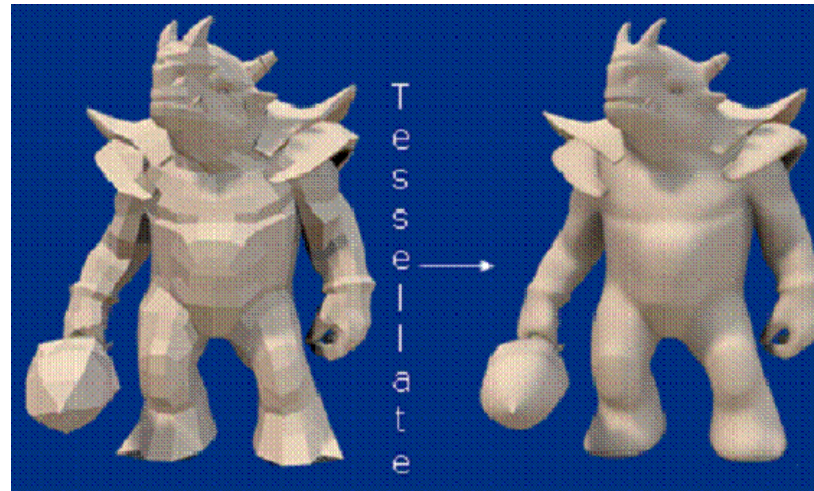
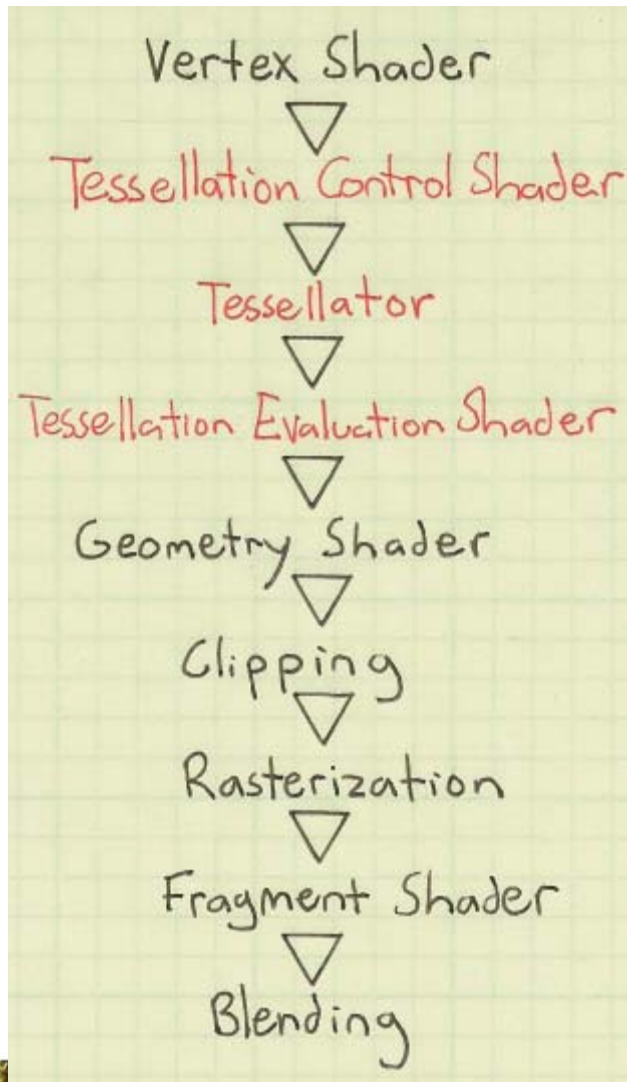
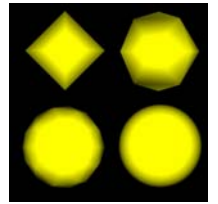


Évolution & extensions: rastérisation

- Les prochaines quelques phases d'évolution du pipeline (programmable) de rastérisation ce concerne avec la génération et modification procédurale de géométrie dans la scène
 - Les choses commencent à ressembler à un moteur de micropolygon...
- Les versions OGL 3.2 et 4.0 ont introduit les *geometry shaders*, et les *tessellation-control* et *tessellation-evaluation shaders**



Évolution & extensions: rastérisation



Évolution & extensions: rastérisation

- Un point un peu dehors le sujet est le **GPGPU**
- Les développeurs d'infographie, étant un groupe super créatif, on trouvé des manières d'utiliser les shaders (originellement conçu exclusivement pour des calculs liées à la synthèse d'images) afin d'implémenter des algorithmes plus générales toutes en prenant avantage de la puissance de calcul parallèle des GPUs
- Éventuellement, l'industrie a répondu en dessinant des architectures de GPU plus générales et des APIs cibler pour le GPGPU: CUDA, OpenCL, etc.



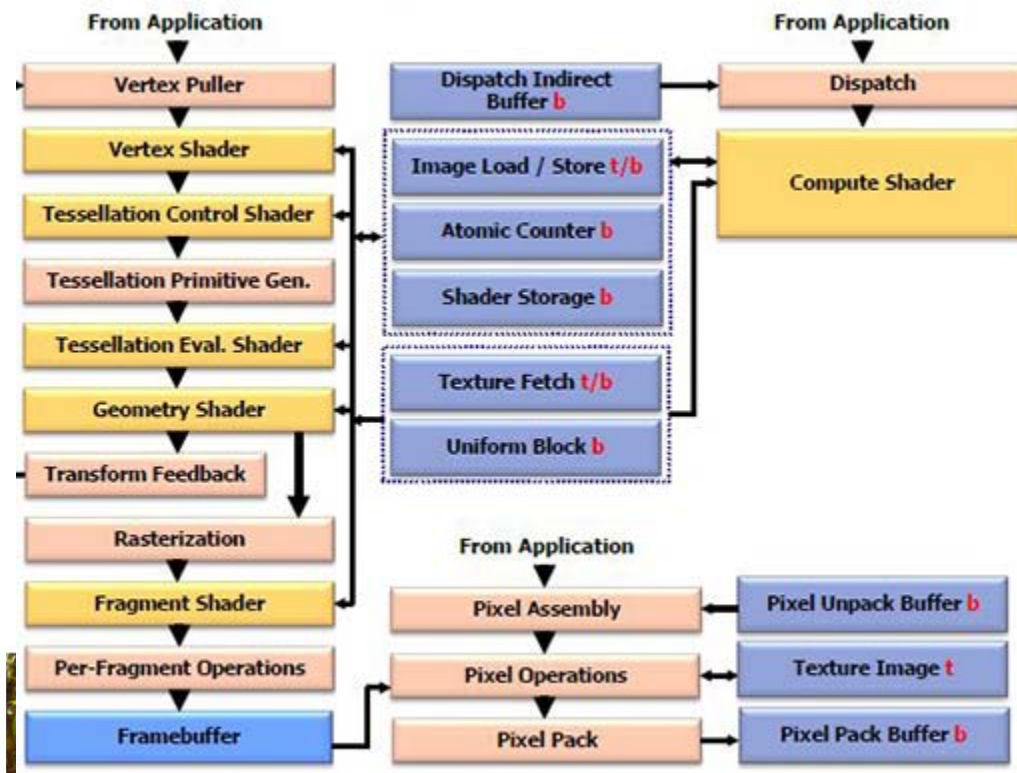
Évolution & extensions: rastérisation

- Ces APIs sont excellents si vous voulez seulement coder un algorithme parallèle et le faire tourner sur un GPU
 - Le moment que tu voudras intégrer ton algorithme de GPGPU avec un composant infographique (ex: visualisation d'un système de particules simuler avec le GPGPU), la communication entre l'API GPGPU et celle du pipeline rastérisation devenait un bottleneck
- Ce problème de bottleneck a plus ou moins été réglé, mais il a aussi forcé les gens à se demander: est-ce qu'on a vraiment besoin de des APIs séparés pour l'infographie sur le GPU et le GPGPU?



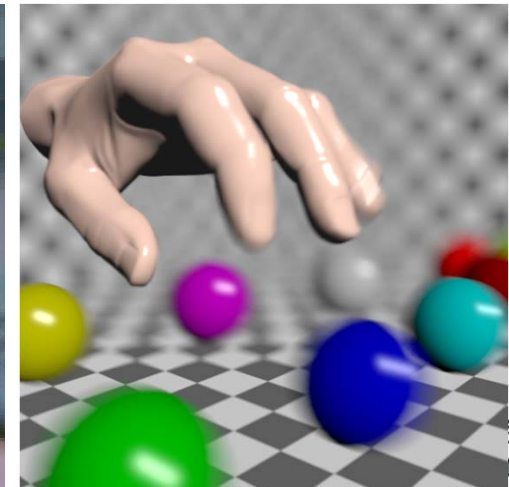
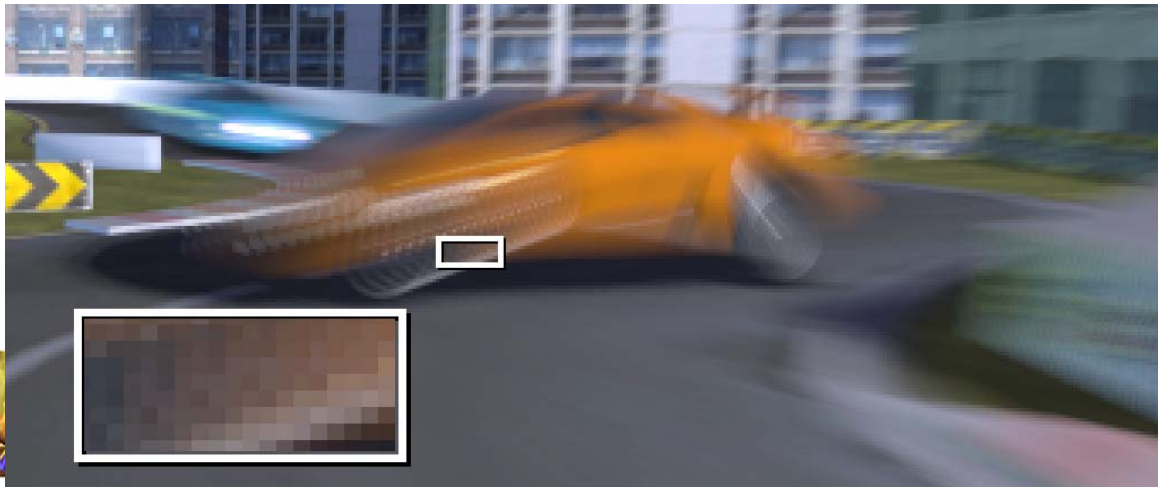
Évolution & extensions: rastérisation

- Alors, les dernières versions de OpenGL et DirectX supportent un nouveau composant programmable: les *compute shaders*
 - Ils permettent le développement d'algorithmes générales* sur le GPU tout avec un API unifier au cas où ces algorithmes seront combinés avec des algorithmes de rendu



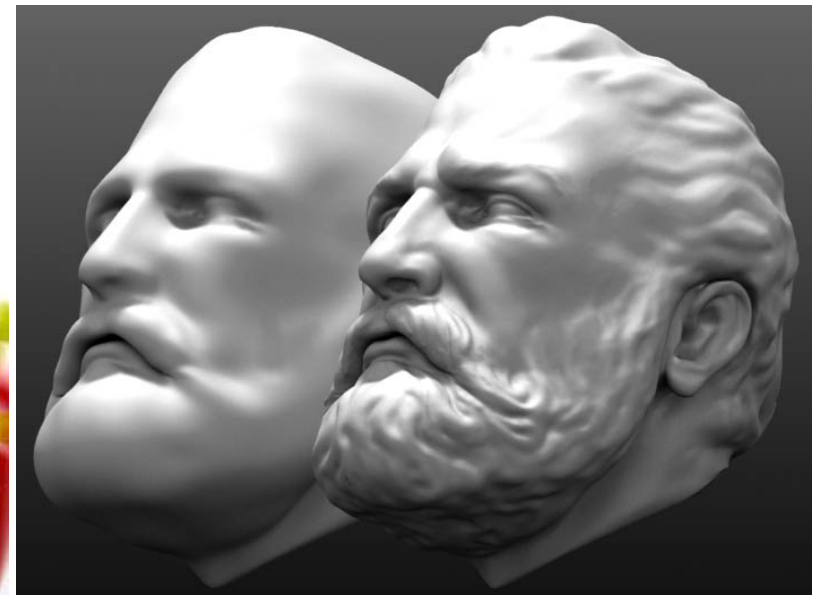
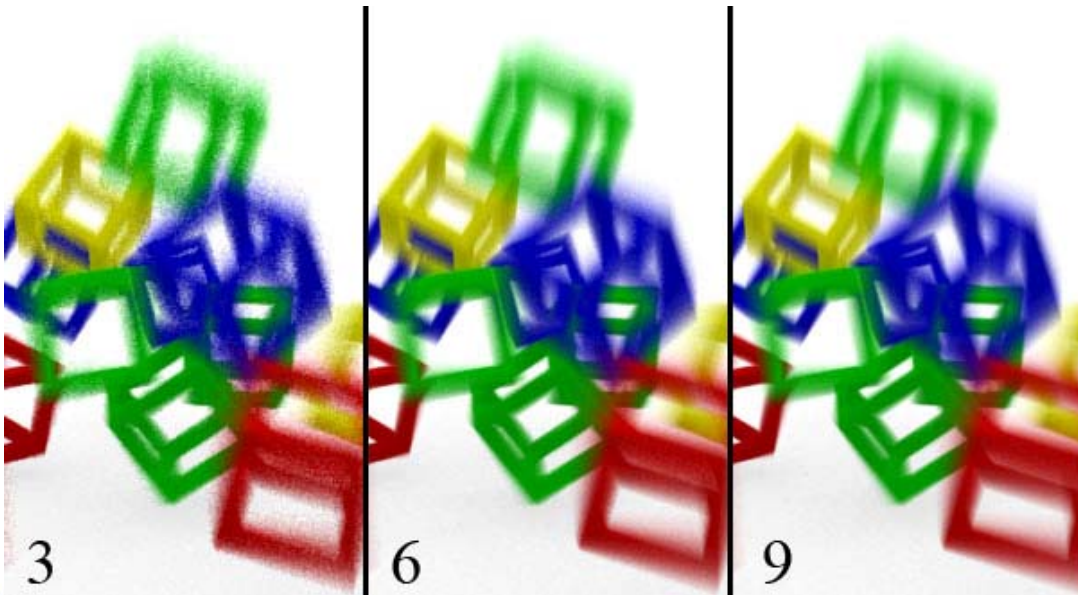
Le future: rastérisation

- Drôlement, tout les systèmes ce rapprochent
- Le future de rastérisation commence à suivre:
 - L'intégration d'un manière fondamentale des effets de distribution avec la *rastérisation stochastique*
 - L'utilisation des structures hiérarchiques pour gérer des scènes plus complexes ainsi que les calculs de visibilité
 - Rendre les calculs incohérent plus cohérent
- Ne risque pas de disparaître dans les prochaines 10 ans



Évolution & extensions: micropolygon

- Comme les buts originaux de l'architecture de rasterisation micropolygon visaient la complexité et le réalisme, il n'est pas surprenant qu'elle a évolué selon ses axes...



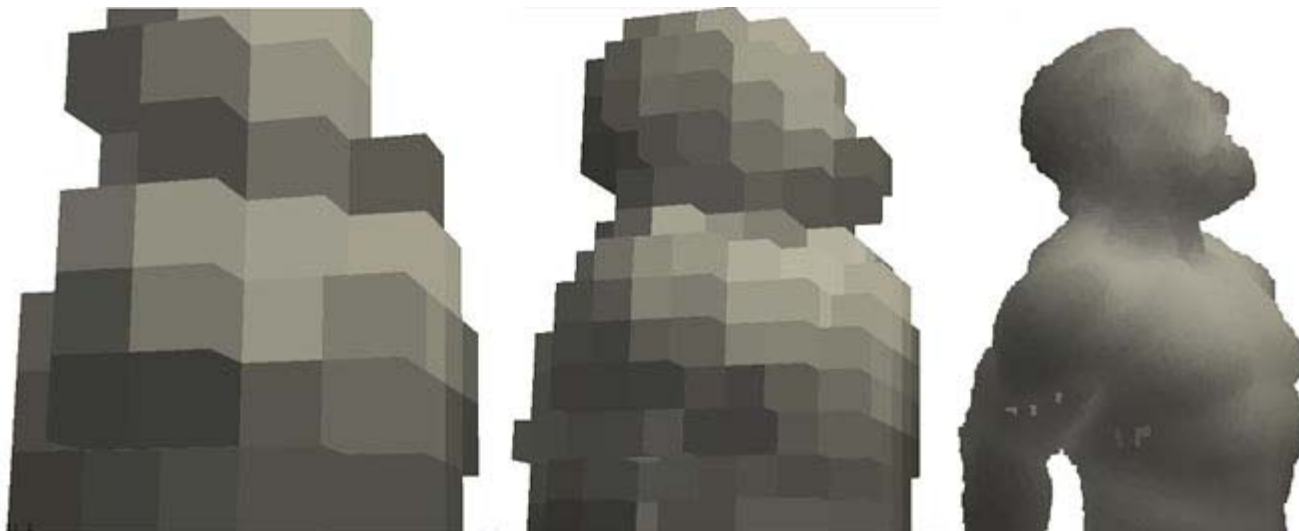
Évolution & extensions: micropolygon

- Afin de gérer des scènes de plus en plus complexes d'une manière adaptatif, un des premiers extensions de PRMan était un représentation hiérarchique d'illumination et de géométrie, le *brickmap*, conçu selon deux observations:
 - le montant de détails nécessaires au loin est réduits, et
 - les effets distribués peuvent échantillonner (avec un taux réduit) selon des représentations filtrés



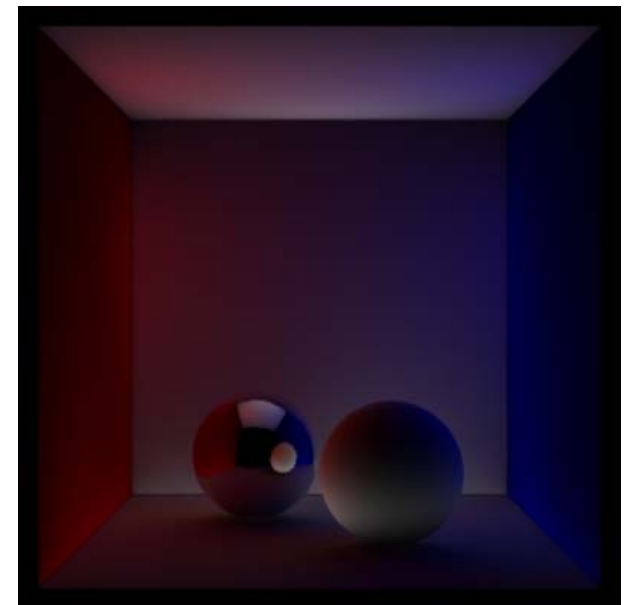
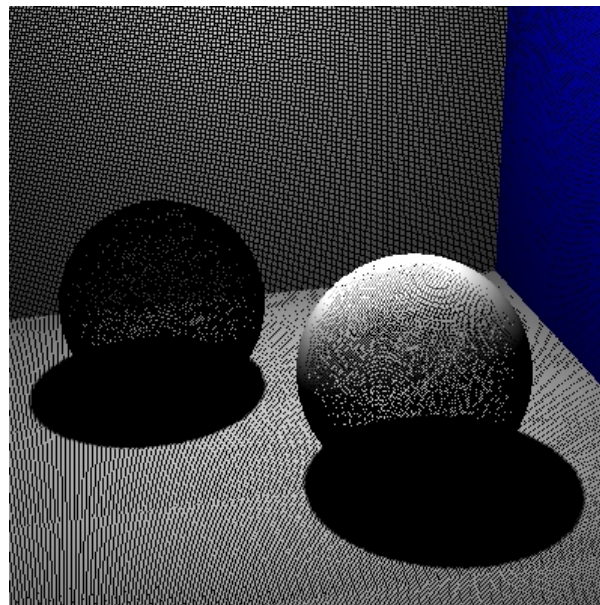
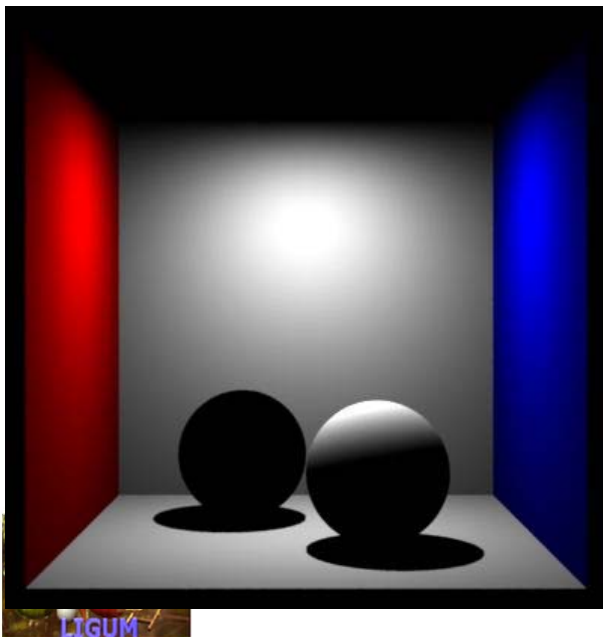
Évolution & extensions: micropolygon

- Afin de gérer des scènes de plus en plus complexes d'une manière adaptatif, un des premiers extensions de PRMan était un représentation hiérarchique d'illumination et de géométrie, le *brickmap*, conçu selon deux observations:
 - le montant de détails nécessaires au loin est réduits, et
 - les effets distribués peuvent échantillonner (avec un taux réduit) selon des représentations filtrés



Évolution & extensions: micropolygon

- Similairement, une représentation de point peuvent être utilisée d'une manière hiérarchique afin de sélectionner un niveau de détail suffisant pour le calcul de différent types d'effets



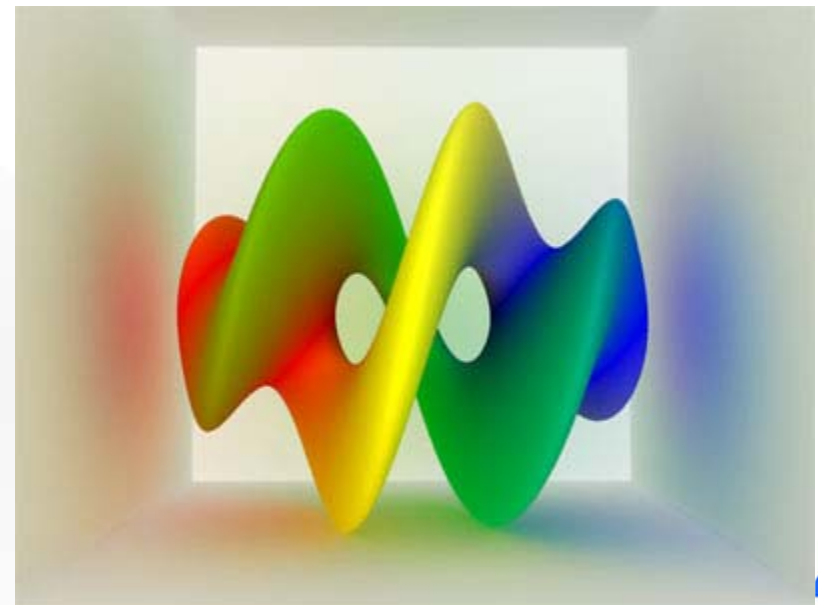
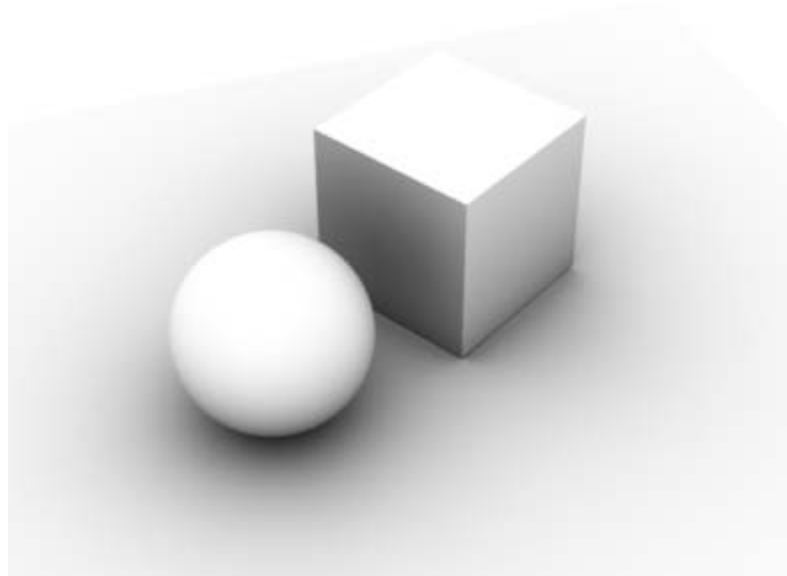
Évolution & extensions: micropolygon

- Similairement, une représentation de point peuvent être utilisée d'une manière hiérarchique afin de sélectionner un niveau de détail suffisant pour le calcul de différent types d'effets



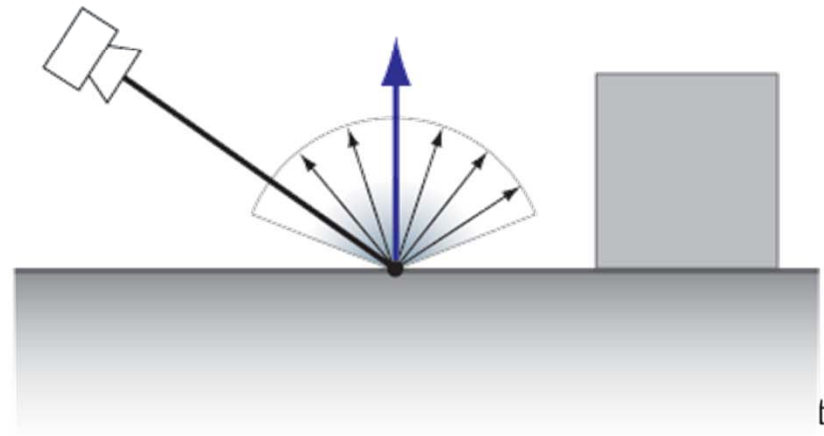
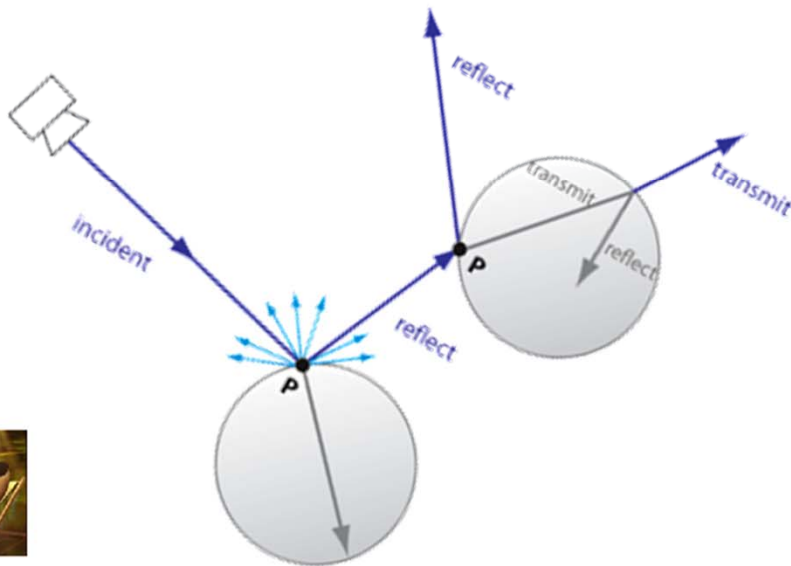
Évolution & extensions: micropolygon

- Similairement, une représentation de point peuvent être utilisée d'une manière hiérarchique afin de sélectionner un niveau de détail suffisant pour le calcul de différent types d'effets



Évolution & extensions: micropolygon

- L'intégration d'un moteur de lancer dans PRMan a permis la simulation des effets plus réaliste
 - Pour (continuer à) gérer les scènes complexes, sans toujours respecter une exactitude mathématique, les représentations brickmap et PB sont aussi supporter dans le cadre de ce sous-système de lancer de rayons



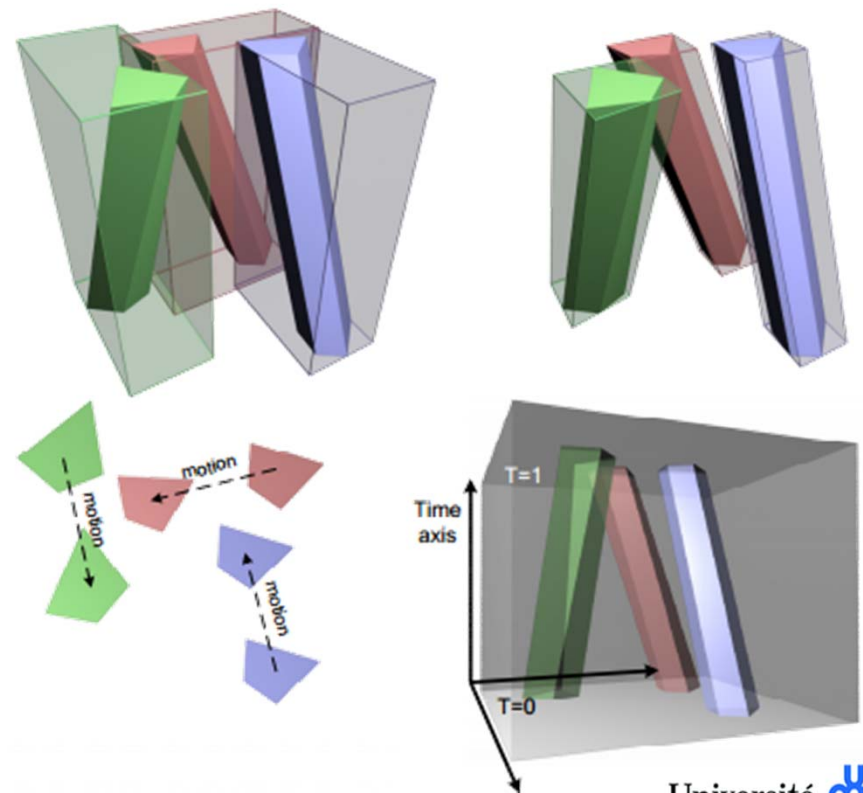
Évolution & extensions: micropolygon

- L'intégration d'un moteur de lancer dans PRMan a permis la simulation des effets plus réaliste
 - Pour (continuer à) gérer les scènes complexes, sans toujours respecter une exactitude mathématique, les représentations brickmap et PB sont aussi supportées dans le cadre de ce sous-système de lancer de rayons



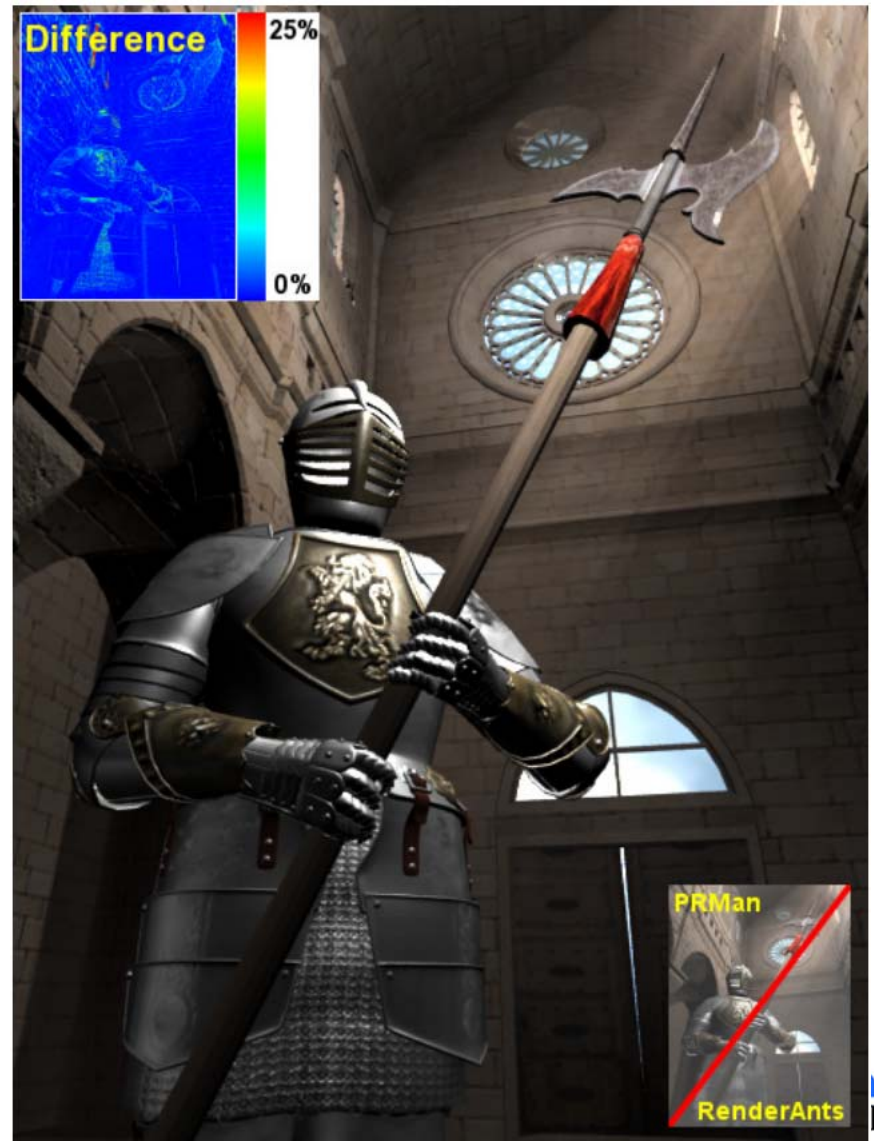
Évolution & extensions: micropolygon

- Il est important de maintenir le support pour chaque effet avec chaque sous-système



Évolution & extensions: micropolygon

- Implémentation GPU
 - Il est très important d'avoir des résultats uniformes à travers les implémentations



Comment choisir un système

- Modèle d'illumination locale ou globale?
 - Besoin de quel type d'information de la scène à chaque élément de calcul?
 - Parallélisations?
- Cohérence (ou non) en “espace rayons”
- Quel type de représentation géométrique
- Est-ce que vous avez besoin: d'efficacité, de précision, tous les deux?
- Il devient de plus en plus fréquent qu'une combinaison de système/représentation sont nécessaires pour atteindre une haute performance



Survol / révision / référence: techniques de rendu...



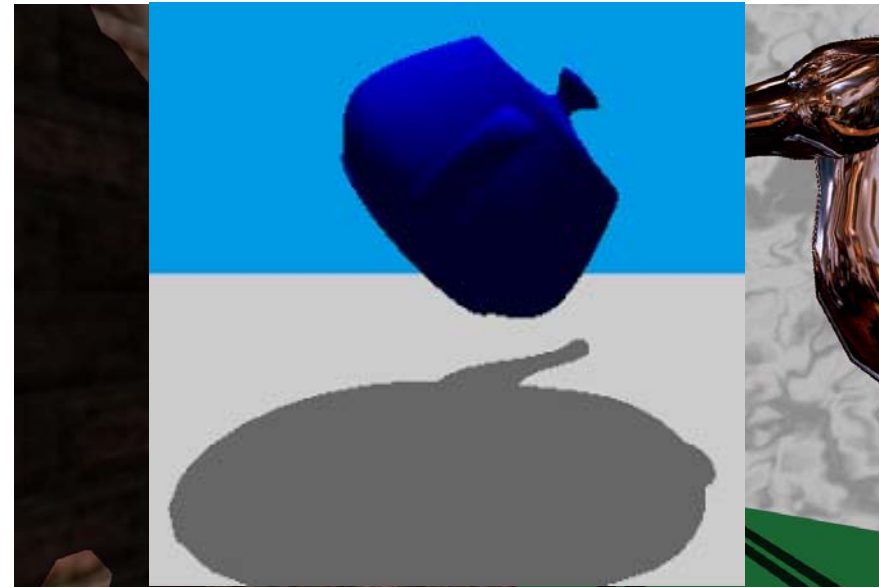
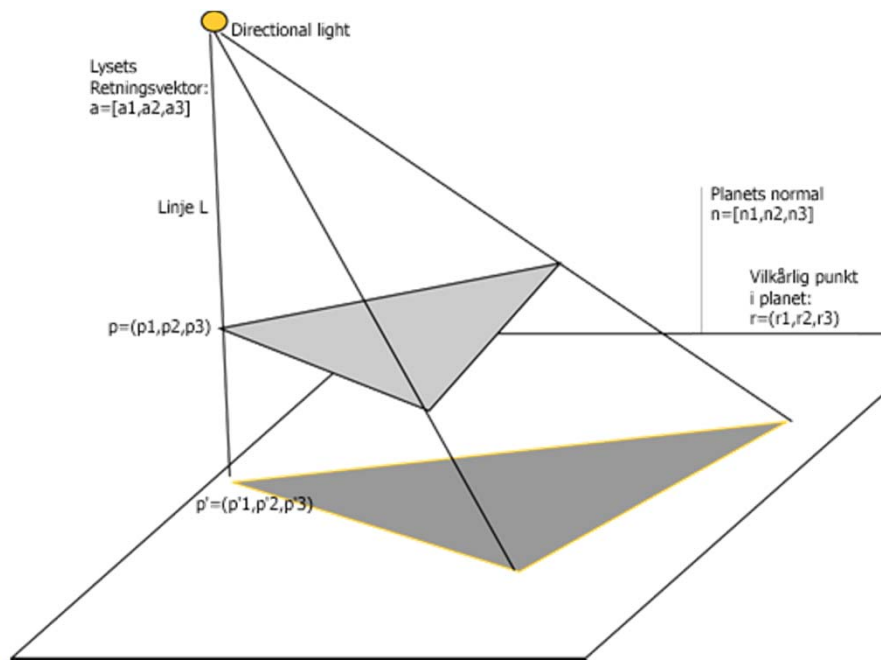
Techniques: Illumination directe

- *Illuminations et matériaux simples*
 - Lancer de rayons, ombres projectives, *shadow mapping*, volumes d'ombre (*shadow volumes*)



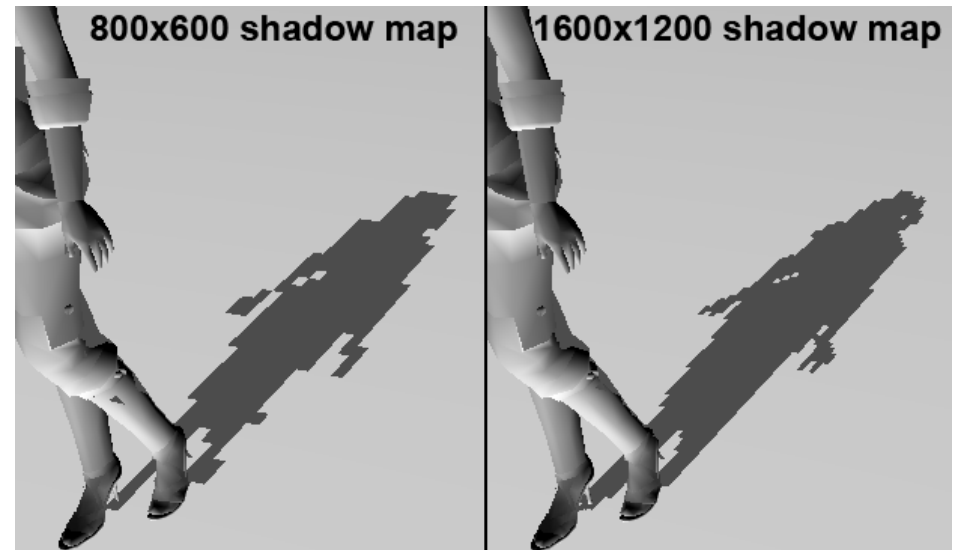
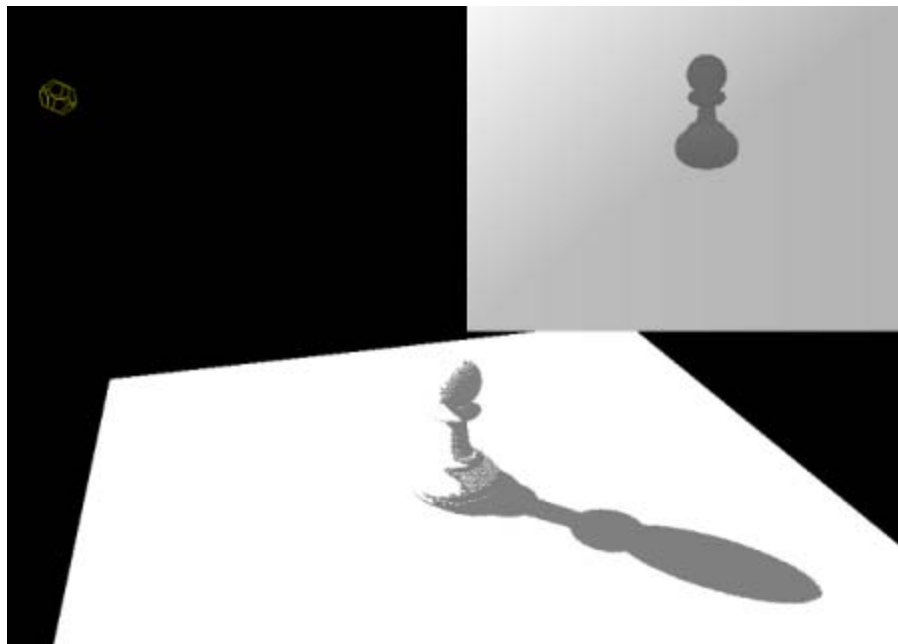
Techniques: Illumination directe

- *Illuminations et matériaux simples*
 - Lancer de rayons, ombres projectives, *shadow mapping*, volumes d'ombre (*shadow volumes*)



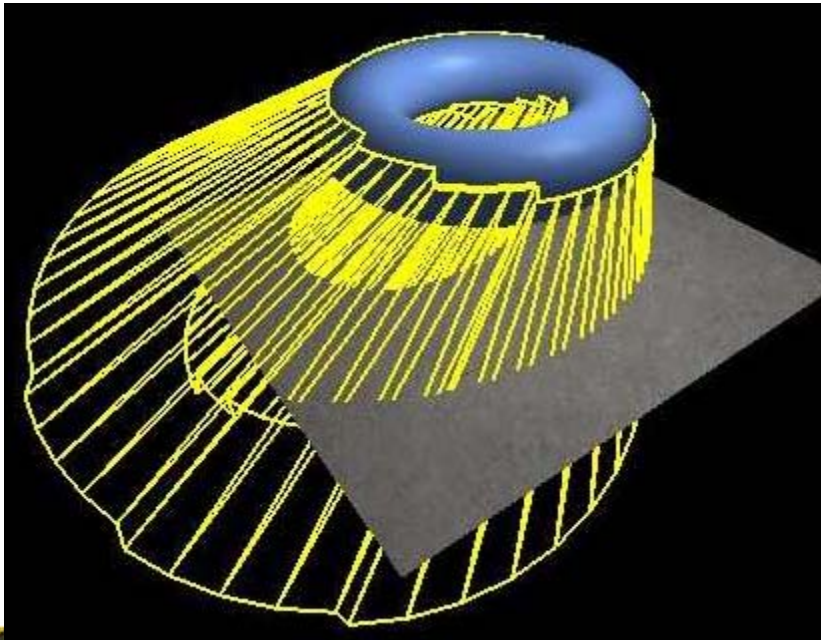
Techniques: Illumination directe

- *Illuminations et matériaux simples*
 - Lancer de rayons, ombres projectives, *shadow mapping*, volumes d'ombre (*shadow volumes*)



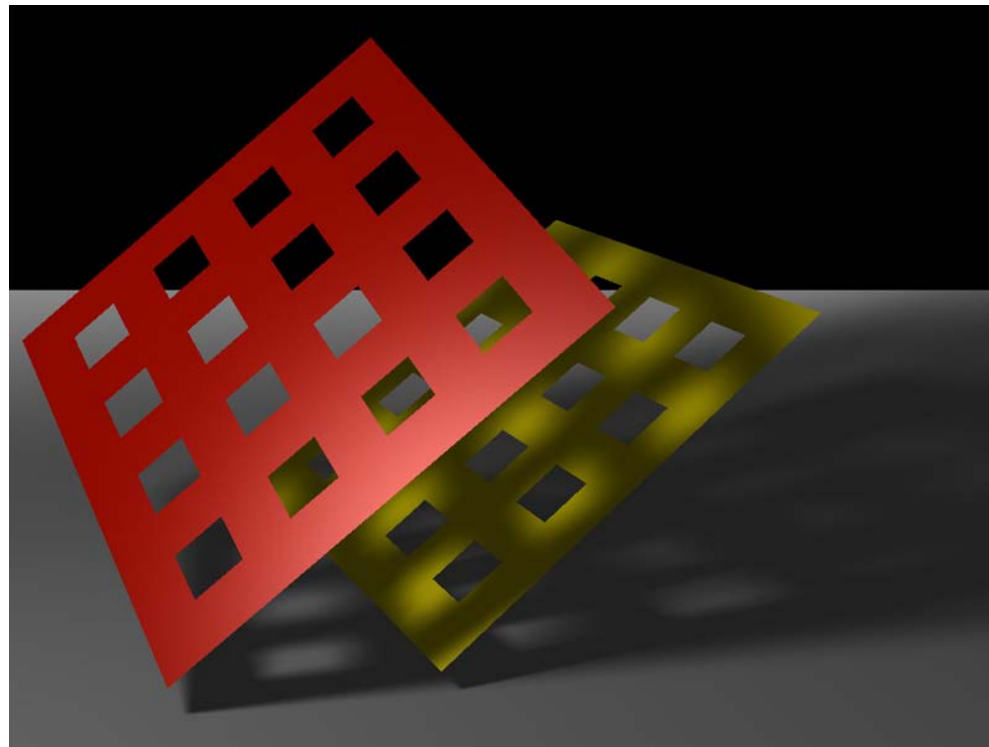
Techniques: Illumination directe

- *Illuminations et matériaux simples*
 - Lancer de rayons, ombres projectives, *shadow mapping*, volumes d'ombre (*shadow volumes*)



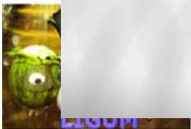
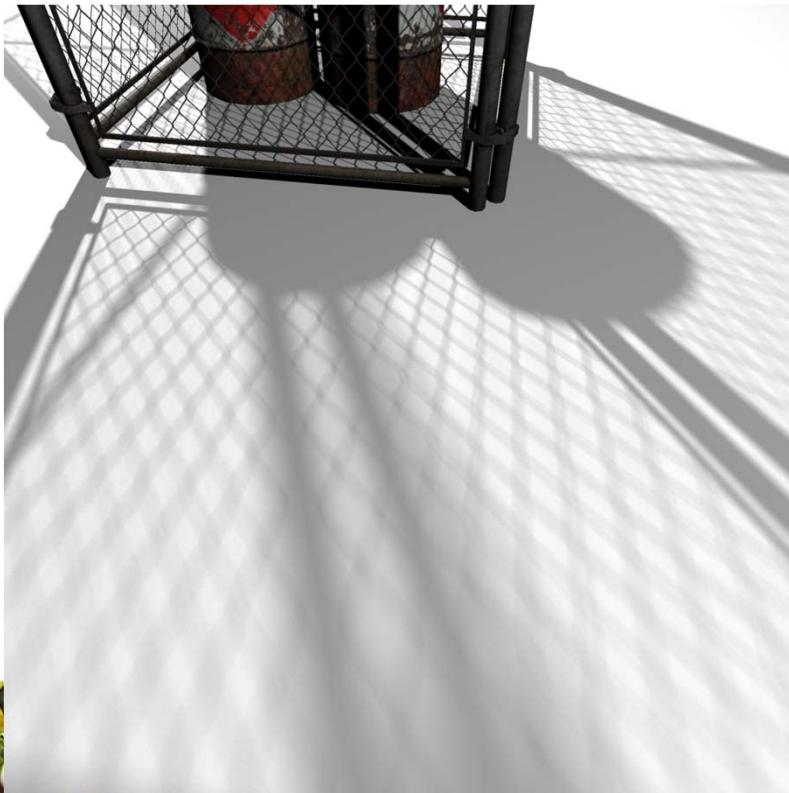
Techniques: Illumination directe

- *Lumières étendues*
 - Lancer de rayons, *soft shadow volumes*, intégration de silhouette



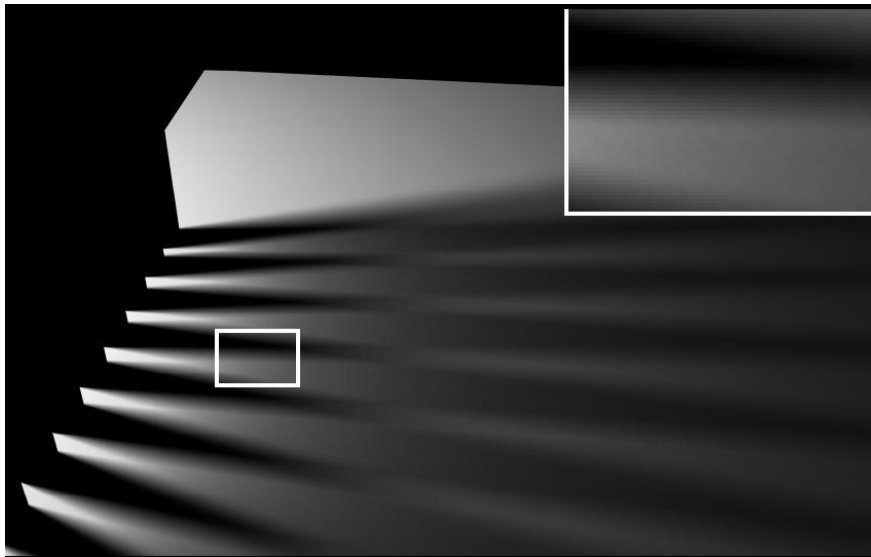
Techniques: Illumination directe

- *Lumières étendues*
 - Lancer de rayons, *soft shadow volumes*, intégration de silhouette



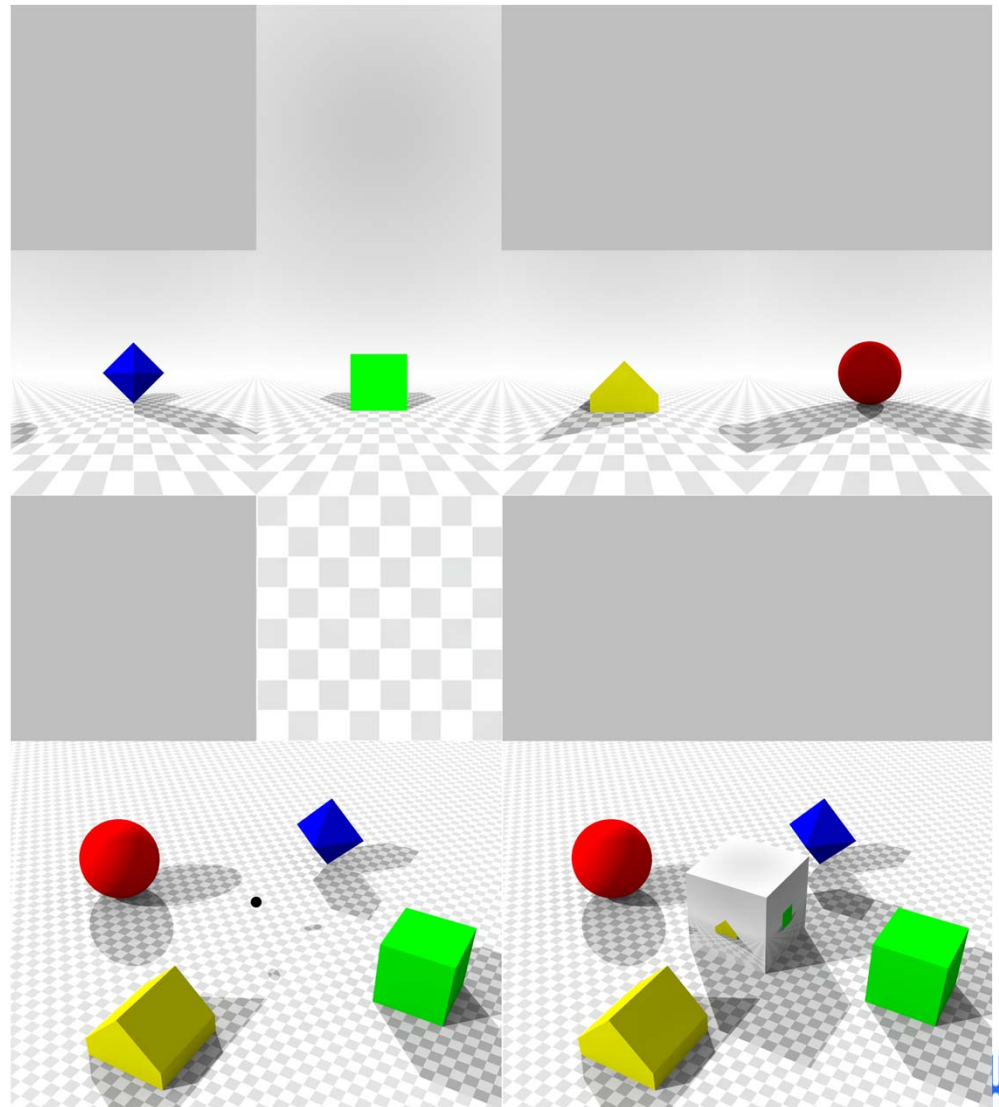
Techniques: Illumination directe

- *Lumières étendues*
 - Lancer de rayons, *soft shadow volumes*, intégration de silhouette



Techniques: Illumination directe

- *Réflexion complexe*
 - Lancer de rayons, réflexion projective, *reflection mapping*, etc...



Techniques: Illumination directe

- *Réflexion complexe*
 - Lancer de rayons, réflexion projective, *reflection mapping*, etc...



Techniques: Illumination directe

- *Réflexion complexe*
 - Lancer de rayons, réflexion projective, *reflection mapping*, etc...



Techniques: Illumination directe

- *Lumières environnementales*
 - Lancer de rayons, *shadow maps*, *precomputed radiance transfer*



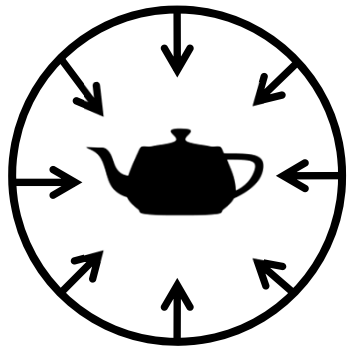
Techniques: Illumination directe

- *Lumières environnementales*
 - Lancer de rayons, *shadow maps*, *precomputed radiance transfer*



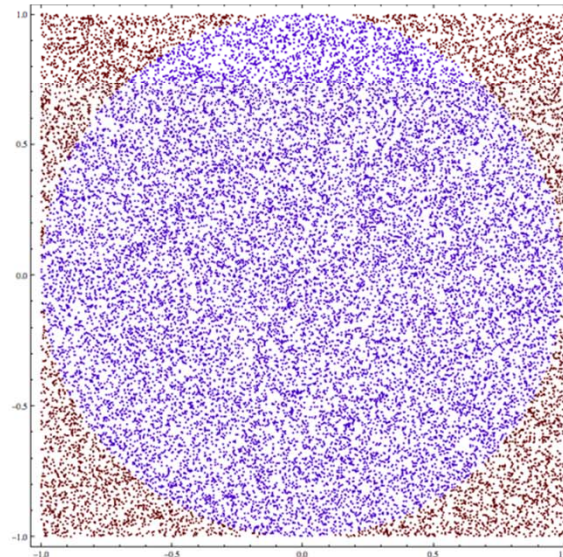
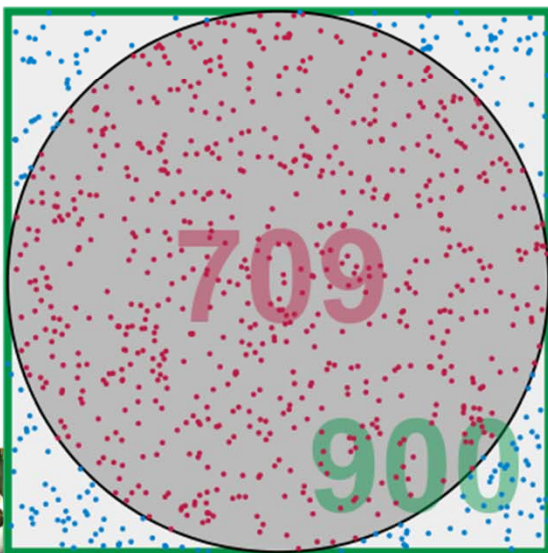
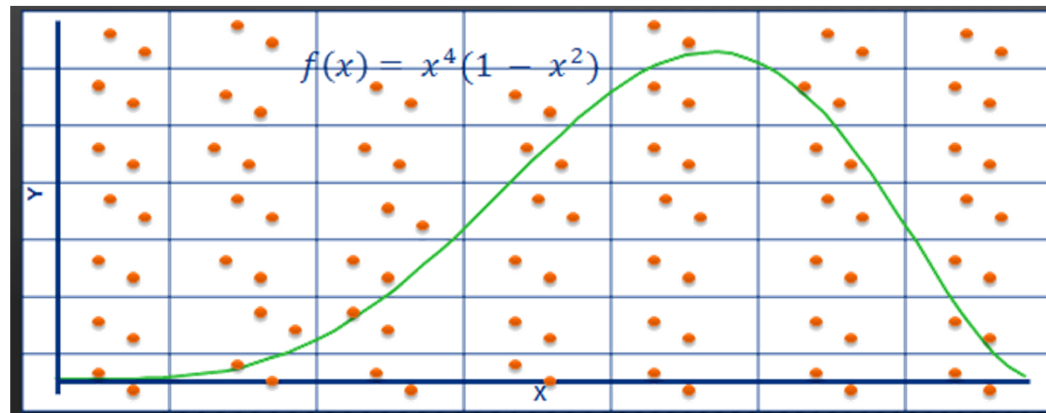
Techniques: Illumination directe

- *Lumières environnementales*
 - Lancer de rayons, *shadow maps*, *precomputed radiance transfer*



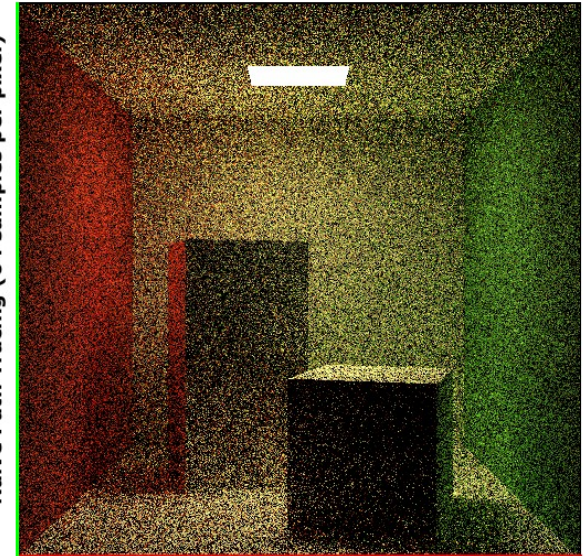
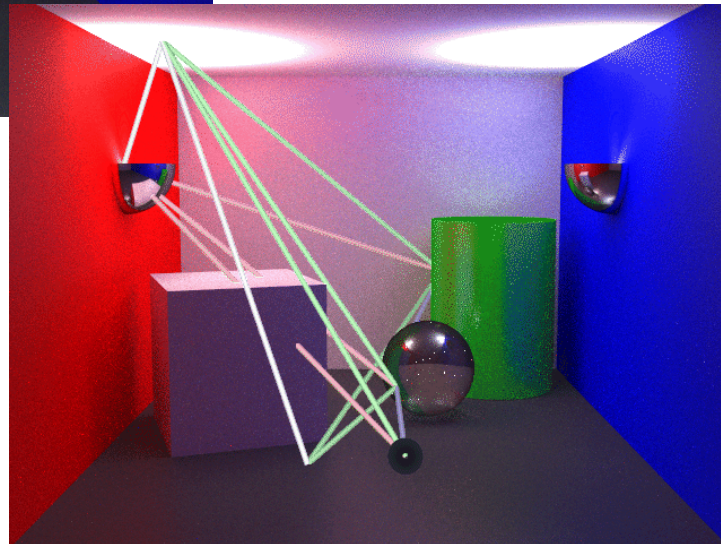
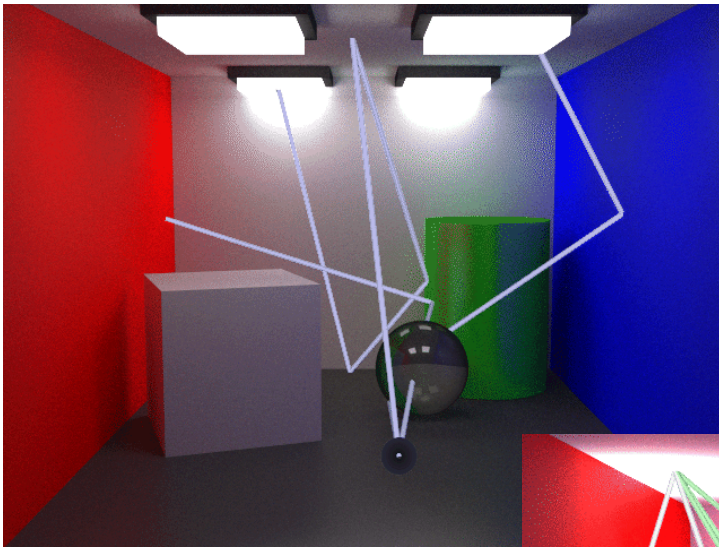
Techniques: Illumination globale

- *Intégration Monte Carlo*



Techniques: Illumination globale

- *Path tracing*



Naive Path Tracing (64 samples per pixel)



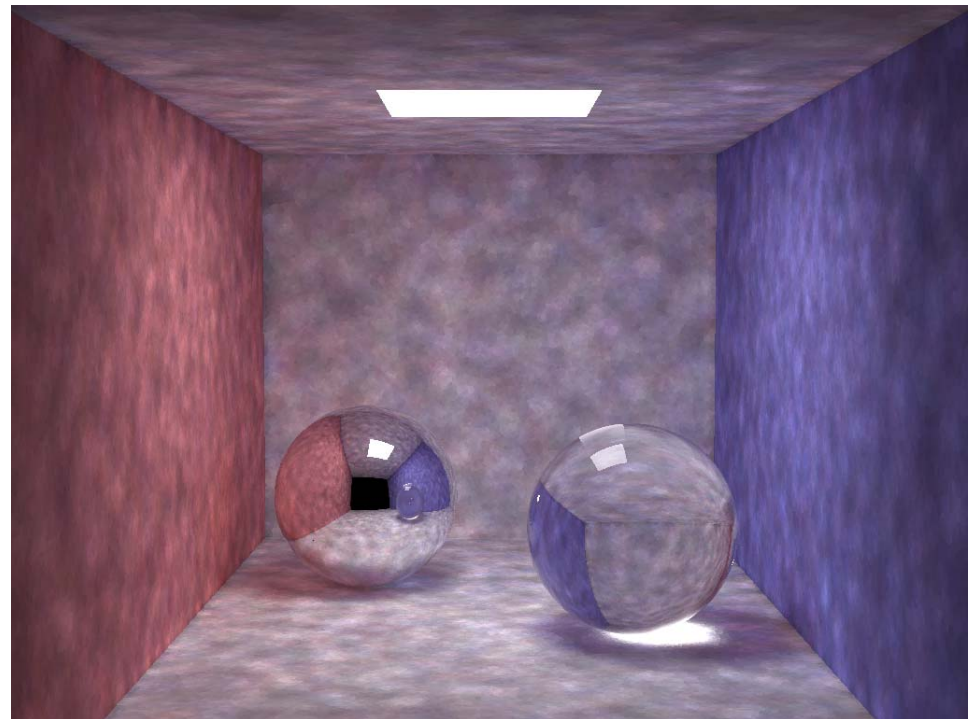
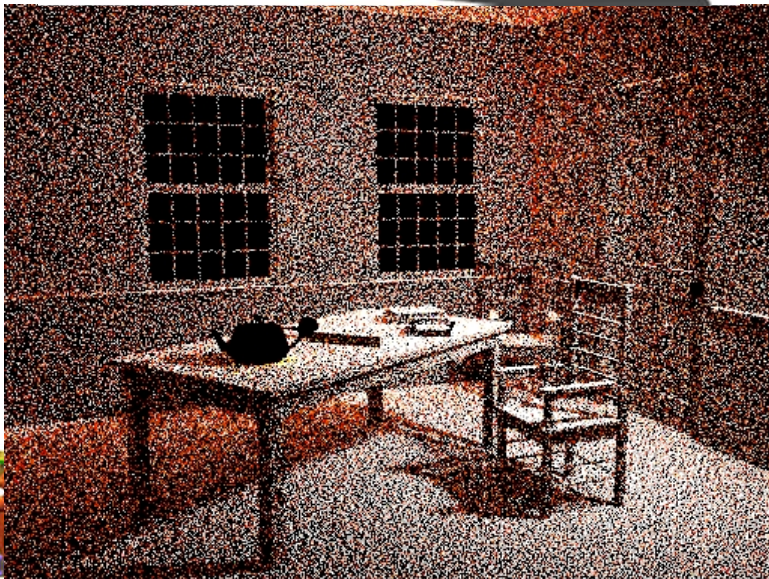
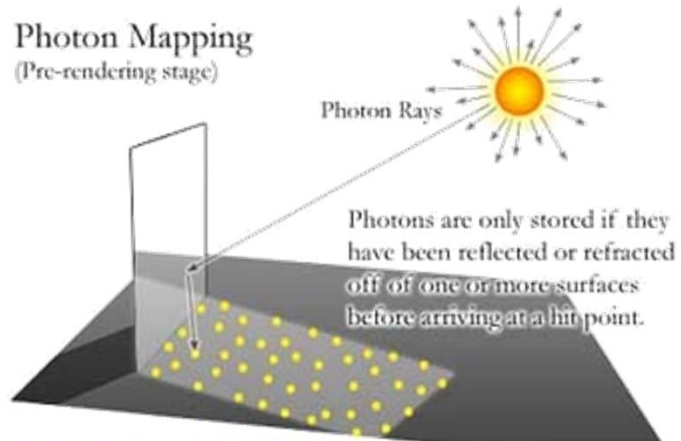
Path Tracing With Explicit Direct (64 samples)



Techniques: Illumination globale

- *Photon mapping*

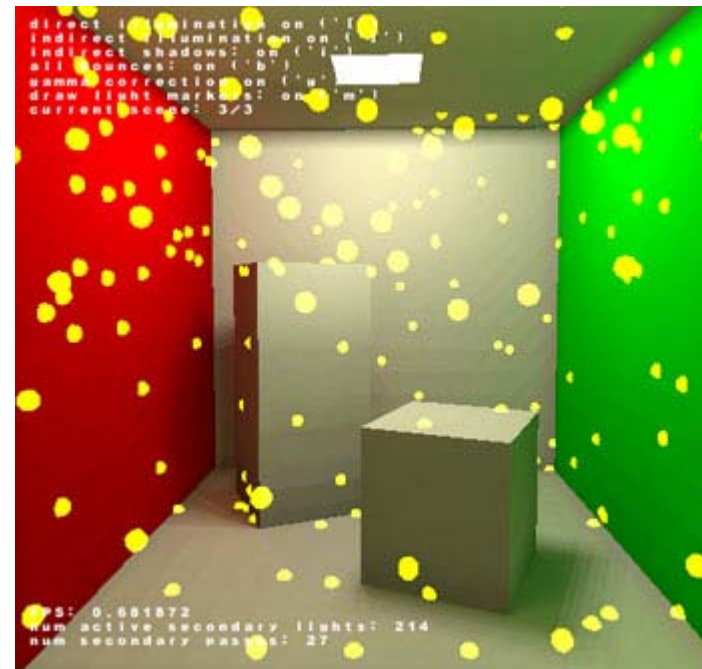
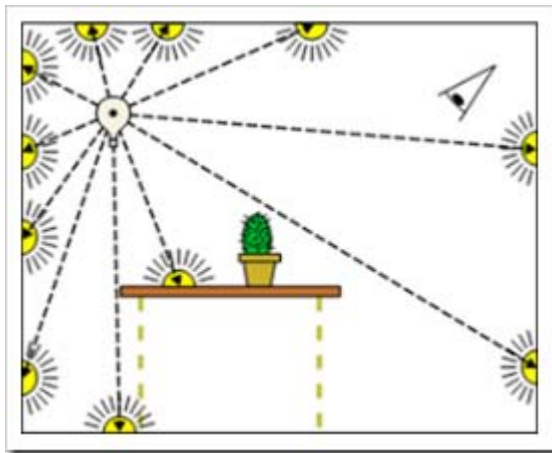
Photon Mapping
(Pre-rendering stage)



Techniques: Illumination globale

- *Virtual Lights*

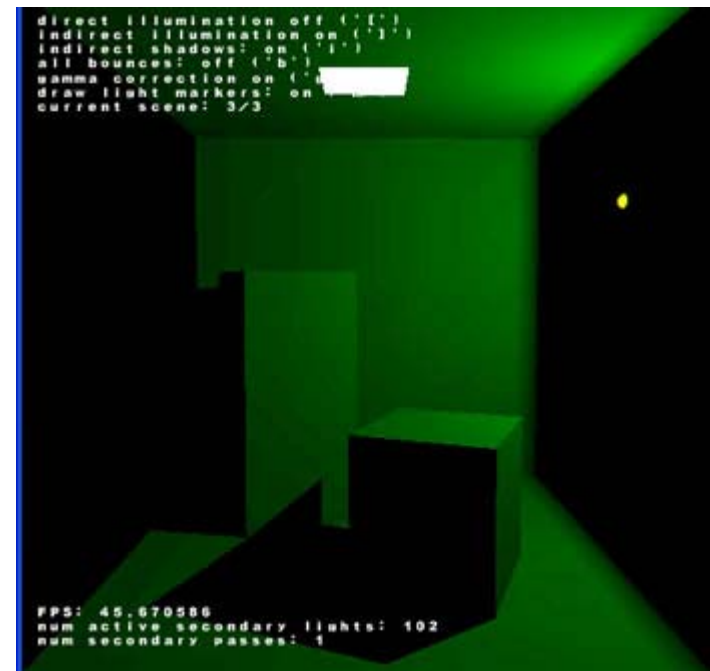
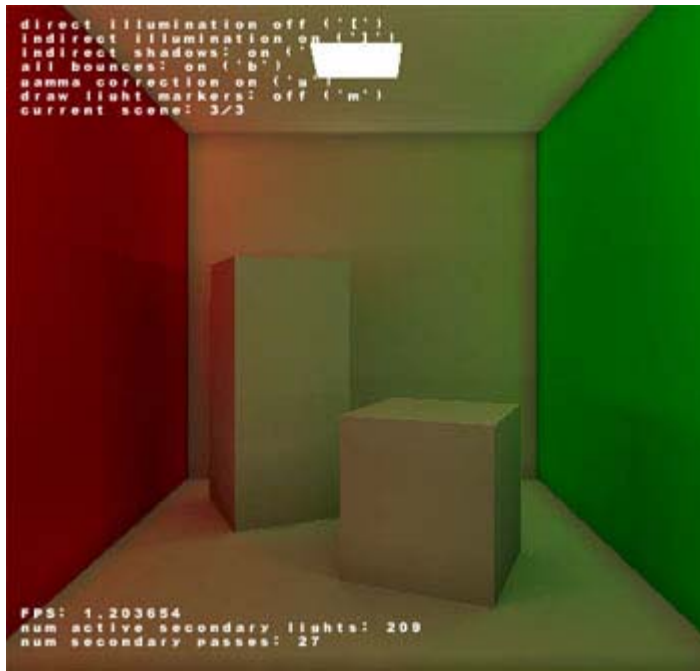
- traitement des singularités faibles, *reflective shadow maps*, *imperfect shadow maps*, *light cuts*



Techniques: Illumination globale

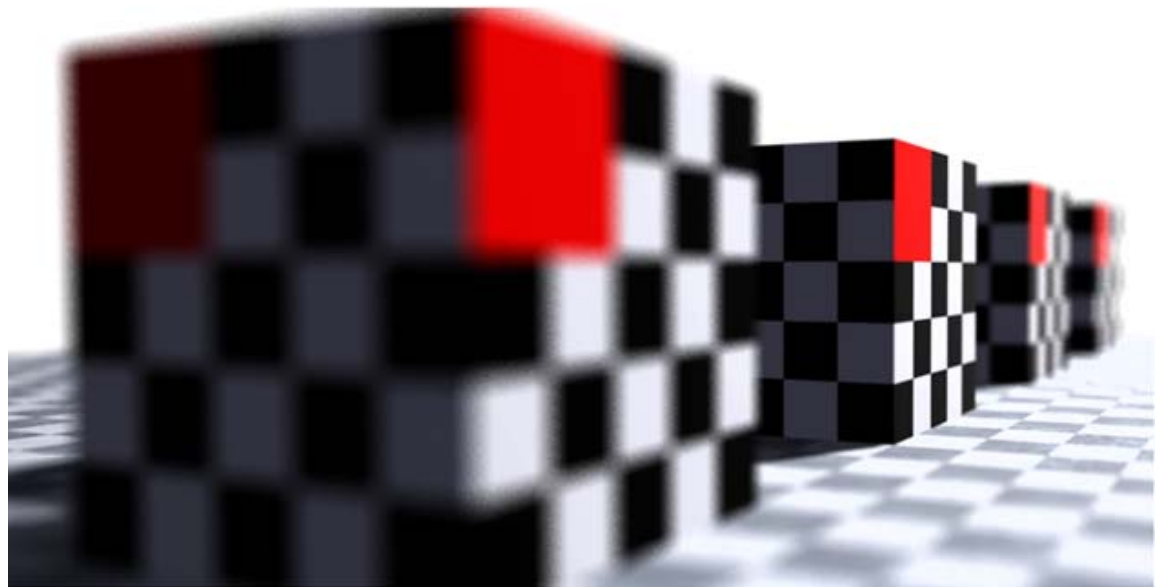
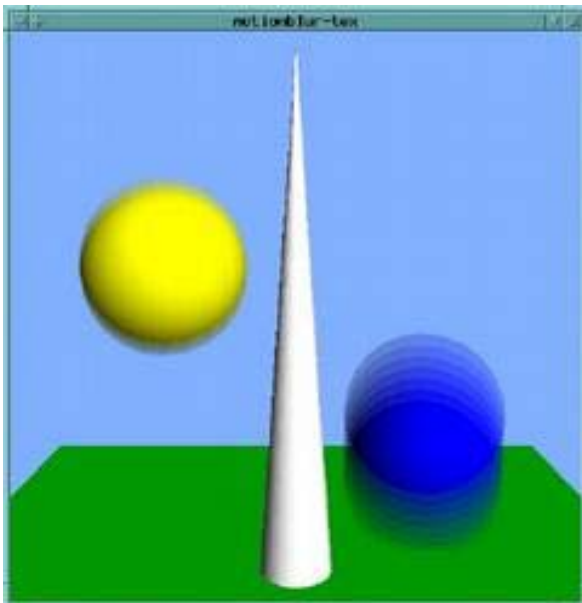
- *Virtual Lights*

- traitement des singularités faibles, *reflective shadow maps*, *imperfect shadow maps*, *light cuts*



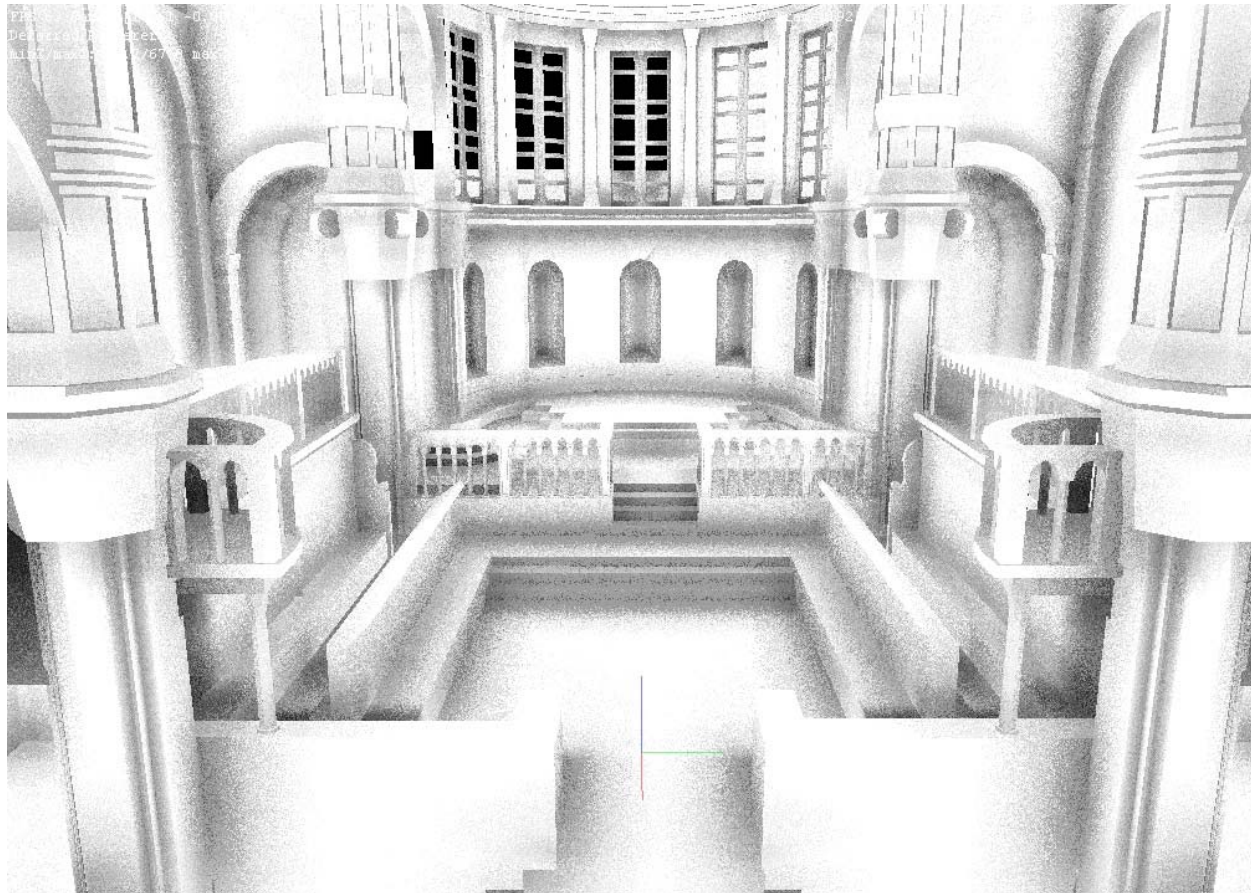
Effets spéciaux / distribués

- *Accumulation buffer*, rasterisation stochastique, techniques *screen-space*, etc...



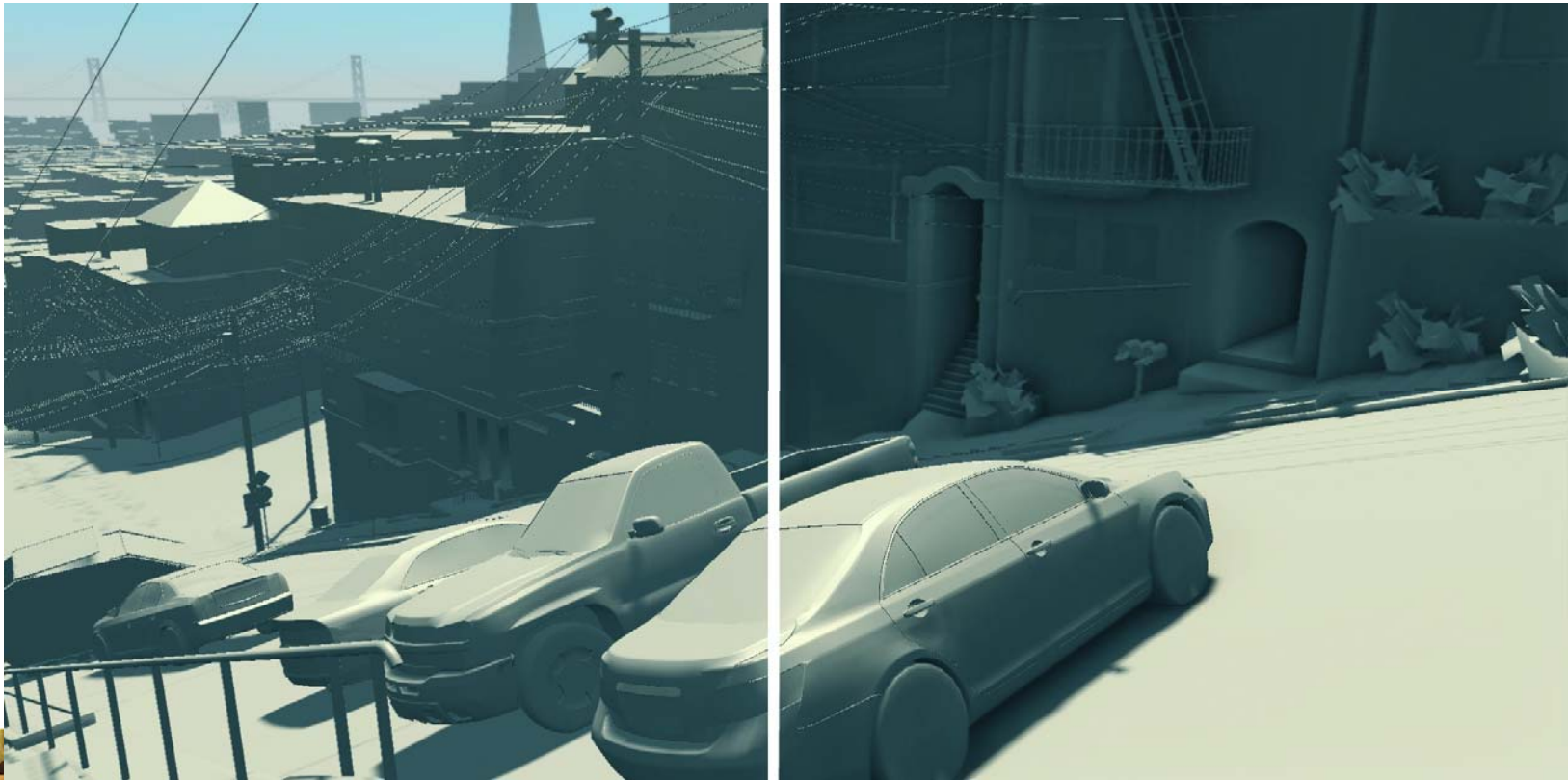
Effets spéciaux / distribués

- *Accumulation buffer*, rasterisation stochastique, techniques *screen-space*, etc...



Effets spéciaux / distribués

- *Accumulation buffer*, rastérisation stochastique, techniques *screen-space*, etc...



Effets spéciaux / distribués

- *Accumulation buffer*, rastérisation stochastique, techniques *screen-space*, etc...

