Rendu temps réel de scènes complexes

Crédits :

Gaël Guennebaud, Joëlle Thollot, Cyril Crassin, Nicolas Thibieroz, Andrew Lauritzen, Michal Valient, Wolfgang Engel, Akenine-Möller, Haines & Hoffman (« *Real-Time Rendering* », AK Peters)

Scène complexe

Proland (https://proland.inrialpes.fr)

Temps réel

Interactif / immersif ⟹ >30Hz

- Généralement pas que le rendu (IA, simulation, etc.)
- Jeux-vidéo : 3-10ms par image pour le rendu
- Résolution croissante des écrans : Full HD, *Retina*, 4K
- Réalité virtuelle : 2 images Full HD à 90Hz
- Matériaux, sources lumineuses et géométrie de complexité croissante

Comment atteindre de telles performances ?

Temps réel

Traitement de l'environnement

- Dépend du point de vue (visibilité)

Traitement de la géométrie

- Niveaux de détail (*Level Of Detail* – LOD)
- Remplacer la géométrie par des images, des points, des voxels

Optimisation du pipeline

Stratégies de rendu avec plusieurs sources lumineuses

Une seule passe

- Pour chaque objet, calcul de l'éclairement selon toutes les sources lumineuses dans un shader
- Shader très complexe, inefficace pour le calcul des ombres

Multi-passes

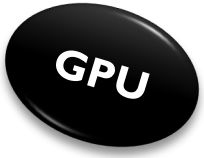
- Pour chaque source de lumière
  - Pour chaque objet affecté par cette source

```
framebuffer += brdf(object, light)
```

- Redondance importante
  (envoi et traitement de la géométrie, accès textures, etc.)

⟹ **Dans les deux cas, calculs coûteux inutiles sur les fragments non-visibles**

Deferred Shading



Motivations

- Scène complexe ⟹ beaucoup de pixels calculés plusieurs fois
- Shaders de plus en plus complexes (ombrages, effets lumineux sophistiqués, nombreuses sources de lumières, etc.)

⟹ **beaucoup de calculs inutiles**

Objectif :

ne pas calculer la couleur des fragments **non visibles**

Idée :

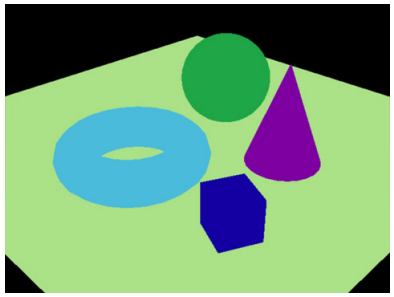
Évaluation paresseuse de l'éclairage

Rendu en 2 étapes :

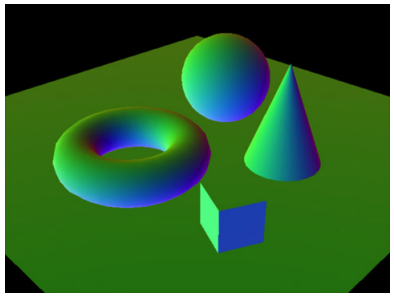
1. calcul d'un **G-buffer** (*geometry/deep buffer*)
2. calcul de l'éclairage

Principe de base

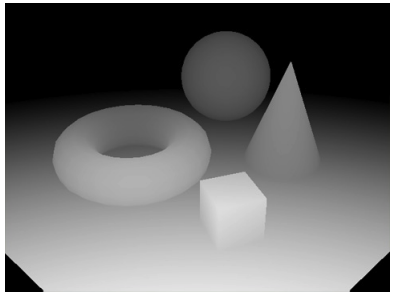
G-buffer



texture color



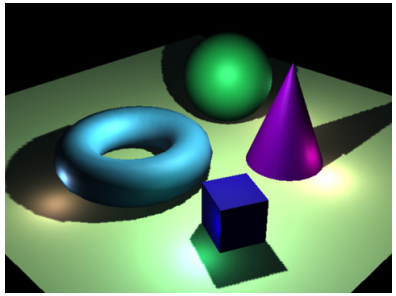
normal buffer



z-buffer (⟹ position 3D)

shading pass

Image finale



Images : Wikipedia

Étape 1 : le G-buffer

Ensemble d'images de même résolution que l'image finale

stocker en chaque pixel toutes les informations propres au pixel et nécessaires pour le calcul d'éclairage :

position, normal, texture/couleur, material id, ..

Rendu en une seule passe (avec les optimisations classiques)

rendu dans plusieurs textures en même temps

- utilisation des FrameBuffer Object
- format flottant
- plusieurs variables « out » dans le fragment shader

FrameBuffer Objects (FBO)

Collection de buffers : lorsqu'un FBO est lié, tous les rendus sont écrits dans les buffers associés

Un FBO dispose de plusieurs ***attachements*** :

- un ou plusieurs ***color buffers*** (max. spécifique au GPU)
  $\Longrightarrow$ ***render buffer*** : optimisé, mais pas réutilisable dans une 2nd passe
  $\Longrightarrow$ **texture**
- au maximum **un** ***depth buffer***
- au maximum **un** ***stencil buffer***

FBO – création

```
glGenFramebuffer(1,&id) // on crée un FBO
glBindFramebuffer(GL_FRAMEBUFFER, id) // on l'active

glGenTexture(…) // on crée une texture pour stocker la couleur
glFramebufferTexture2D(GL_FRAMEBUFFER,
            GL_COLOR_ATTACHMENT0 …) // on l'attache au FBO

glGenRenderbuffer(…) // on crée un render buffer
                     // pour stocker la depth map
glBindRenderbuffer(…) // on l'active
glRenderbufferStorage(..) // on alloue l'espace mémoire
glFramebufferRenderbuffer(GL_FRAMEBUFFER,
            GL_DEPTH_ATTACHMENT, …)  // on l'attache au FBO
```

FBO : http://www.opengl.org/wiki/Framebuffer_Object

***Render buffers* : http://www.opengl.org/wiki/Renderbuffer_Object**

FBO – utilisation

```
glBindFramebuffer(GL_FRAMEBUFFER,id) // On l'active
// On associe la variable out du fragment shader out_color
// au color buffer 0
glBindFragDataLocation(prg_id, 0, "out_color");
// Important : on redéfini le viewport à la taille du FBO
glViewport(0, 0, fbo_width, fbo_height);
// On réinitialise les couleurs et le depth buffer
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
// on dessine dans le FBO
glBindFramebuffer(GL_FRAMEBUFFER,0) //on le désactive
// on peut utiliser la texture et/ou le depth buffer
// pour une seconde passe de rendu
```

FBO : http://www.opengl.org/wiki/Framebuffer_Object

***Render buffers* : http://www.opengl.org/wiki/Renderbuffer_Object**

Étape 1 : le G-buffer

Fragment Shader type :

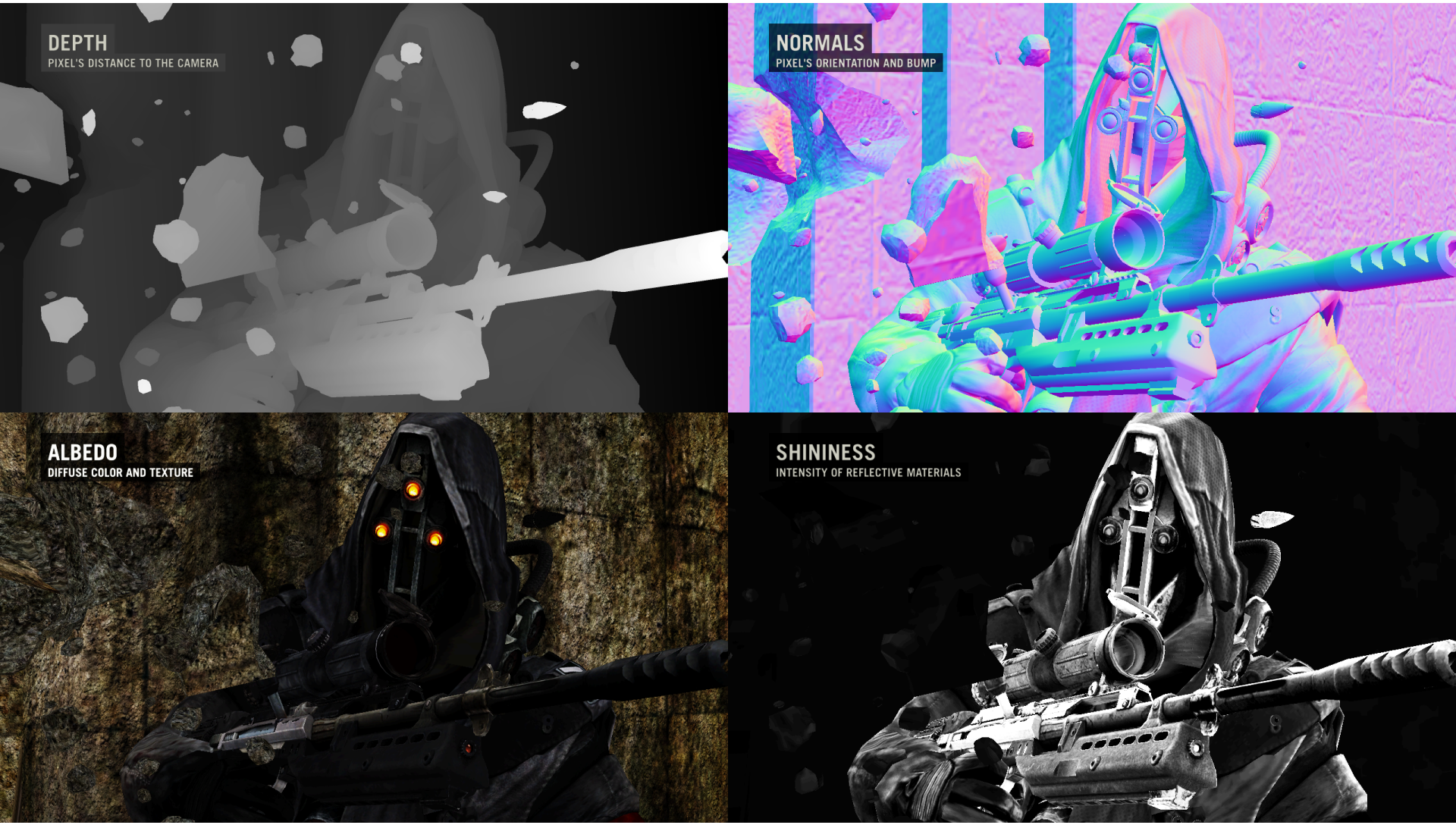
```
in vec2 texcoord;
in vec3 normal;

uniform sampler2D color_map;
uniform float shininess;

out vec4 out_color_s;
out vec4 out_normal_z;

void main()
{
   out_color_s.rgb = texture(color_map,texcoord).rgb;
   out_color_s.a = shininess;
   out_normal_z.xyz = normal;
   out_normal_z.w = gl_FragCoord.w ;
}
```

Étape 1 : le G-buffer



Killzone 2, Guerrilla BV

Étape 1 : le G-buffer

Limitations

- Nécessite beaucoup de mémoire graphique
- Va nécessiter de nombreuses lectures en mémoire (*bandwidth*)

Étape 2 : shading

Passe d'éclairage : calculer l'éclairage **en chaque pixel**

- Tracer un quad recouvrant l'écran $\Longrightarrow$ 1 fragment = 1 pixel
- Calcul de l'éclairage dans un Fragment Shader

Détails

- Récupération des données à partir des textures du *G-buffer* (profondeur, normale, matériau…)
- Déduire la position 3D à partir des coordonnées écran du pixel (`gl_FragCoord`) et de la profondeur
- Appliquer le modèle d'éclairage (Blinn-Phong, par exemple)

Étape 2 : shading

Difficultés

- **Quid des sources lumineuses multiples ?**
- Matériaux multiples ?
- Antialiasing ?
- Transparence ?

Une passe de shading par sources

Avec plusieurs sources de lumières

- Pour chaque fragment du `gbuffer` (z, normale, couleur, etc.)
- Pour chaque lumière (`light`)

```
out_color += brdf(gbuffer,light);
```

Avantage : shaders simplifiés

Inconvénient : accès multiples au g-buffer
(autant de fois qu'il y a de sources)

Une passe de shading par sources

Killzone 2, Guerrilla BV

Une passe de shading par sources

La couleur des fragments **est mélangée** à celle des pixels du **color buffer :**

$$C_{pixel} = C_{pixel} + C_{frag}$$

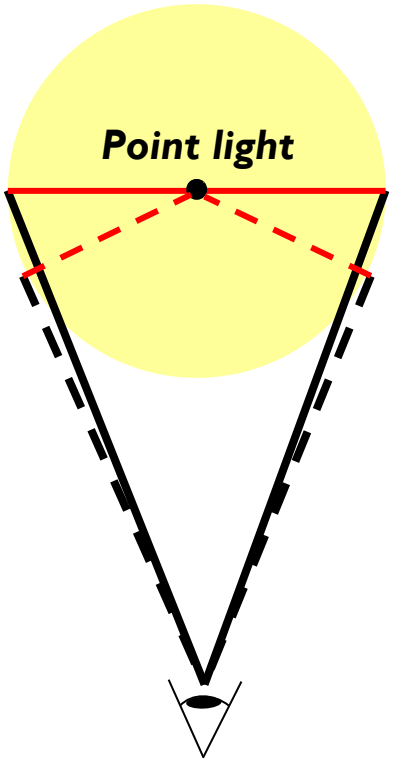
En OpenGL :

- `glBlendFunc(GL_ONE, GL_ONE)`
- `glEnable(GL_BLEND)`

Optimisation

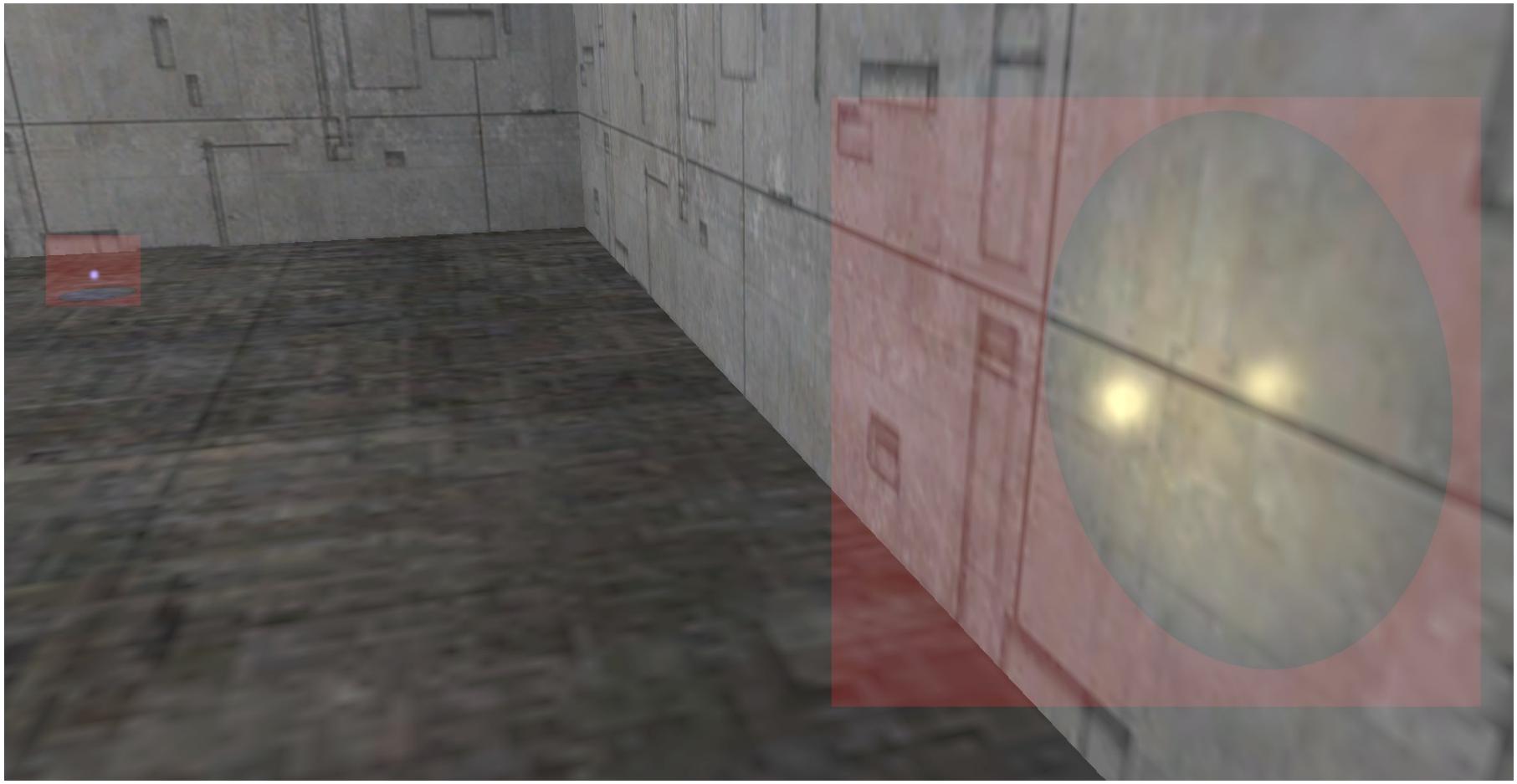
Limiter le calcul aux pixels affectés par chaque source de lumière

- Proxy 3D
  - sphères (*point light*) et cônes (*spot*)
  - plus coûteux à rastériser
- Quad en espace écran
  - géométrie plus simple
    (mais attention à la projection !)
  - approximation assez grossière



Quads en espace écran

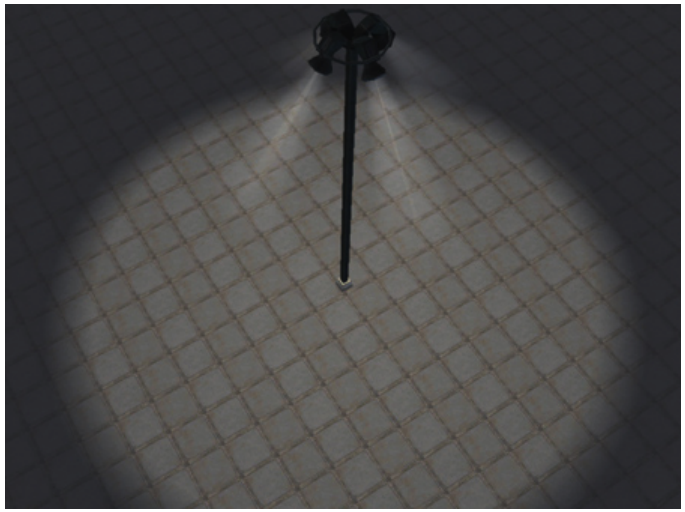
Quads en espace écran



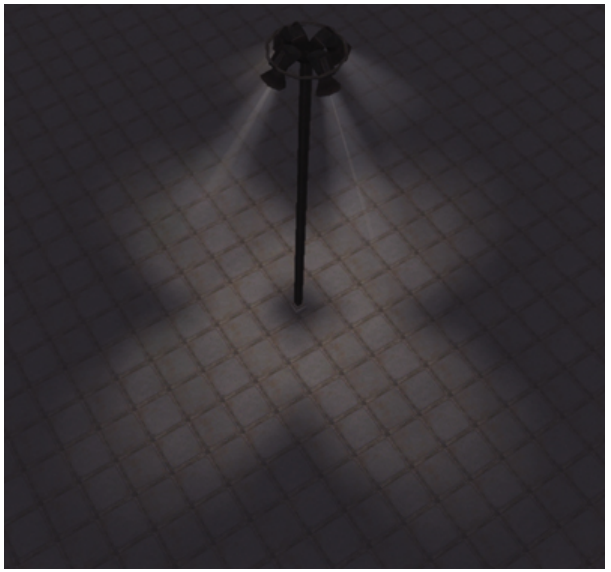
Optimisation

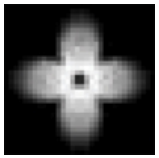
Regrouper les petites sources

1 seul spot



4 spots simulés
par un spot + 1 texture





Optimisation - Rendu par tuiles

[Andersson 2009]

Idée :

Limiter le nombre d'accès au *g-buffer*

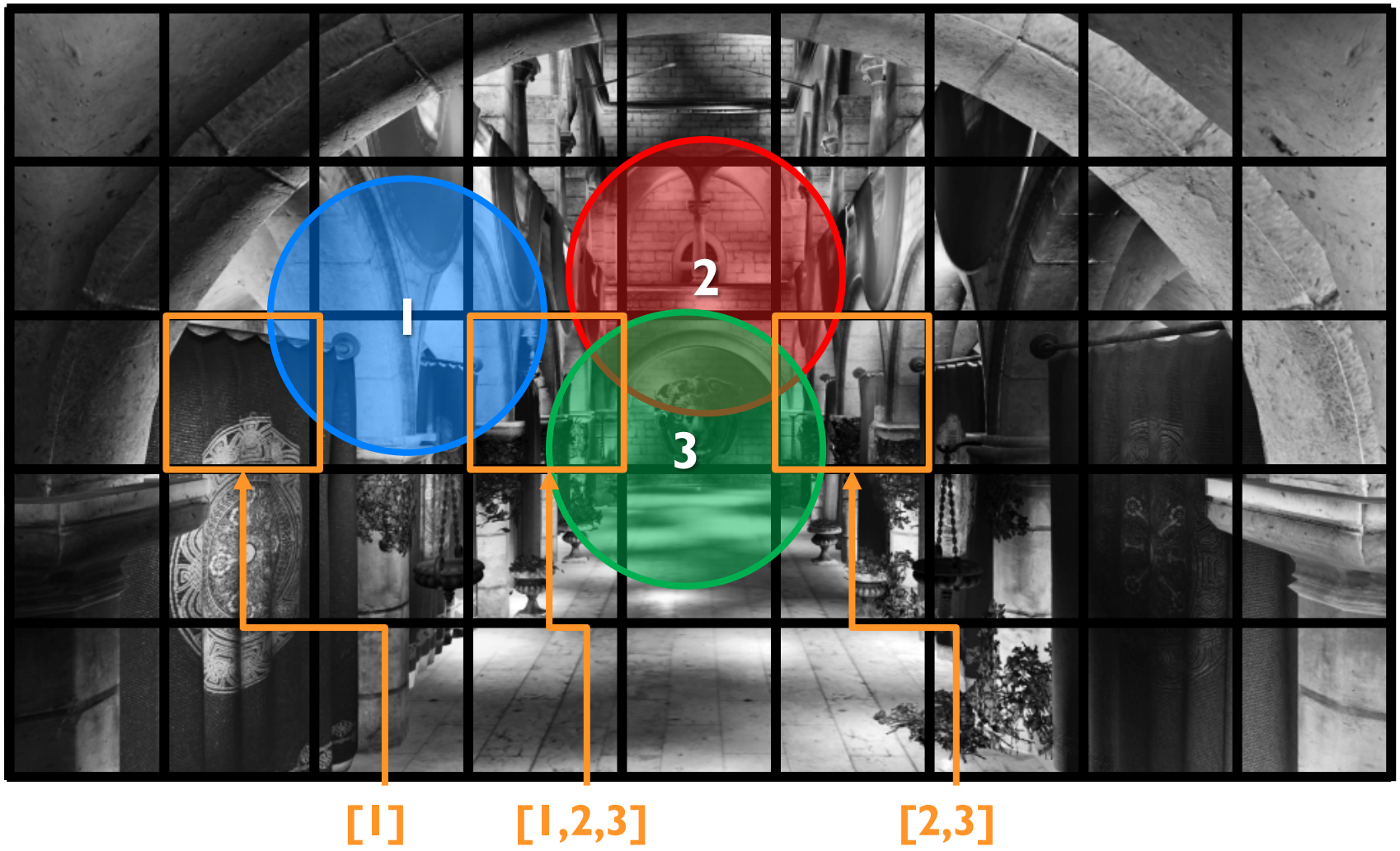
$\Longrightarrow$ évaluer plusieurs sources en même temps pour un fragment

En pratique :

grouper les sources dont les contributions se recouvrent en espace écran

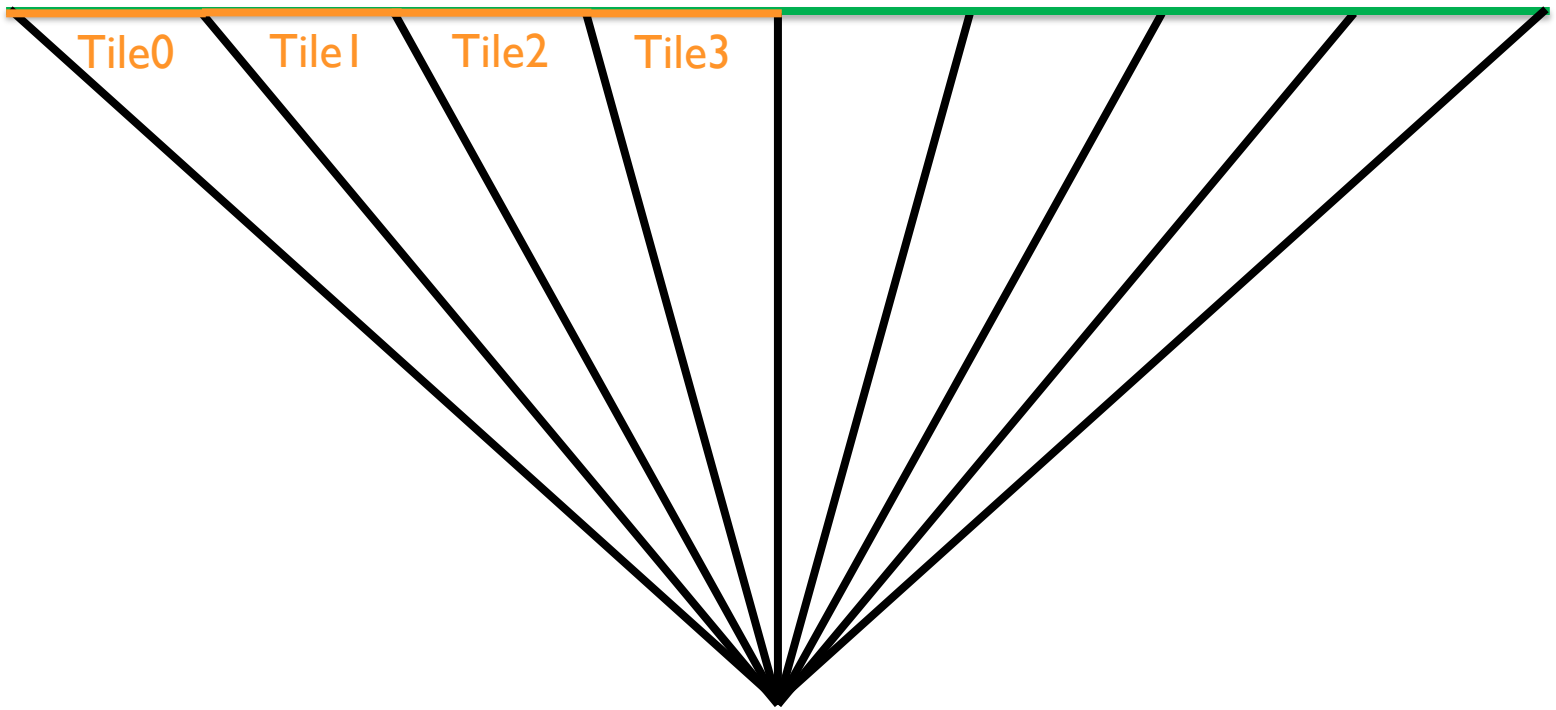
http://leifnode.com/2015/05/tiled-deferred-shading/

1. découper l'écran en tuiles



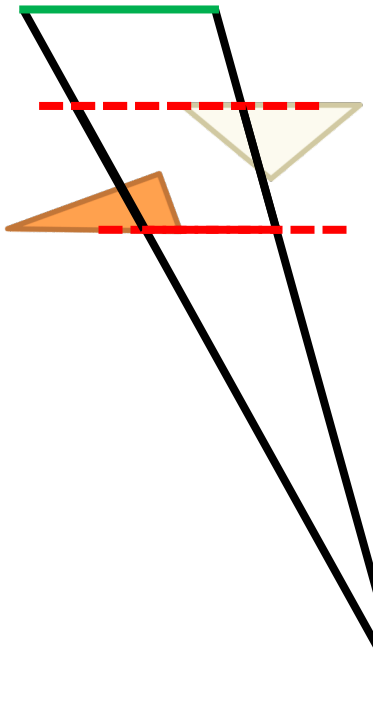
Gareth Thomas (AMD), *"Advancements in Tiled-Based Compute Rendering"*, GDC 2015

2. créer un *frustum* par tuile



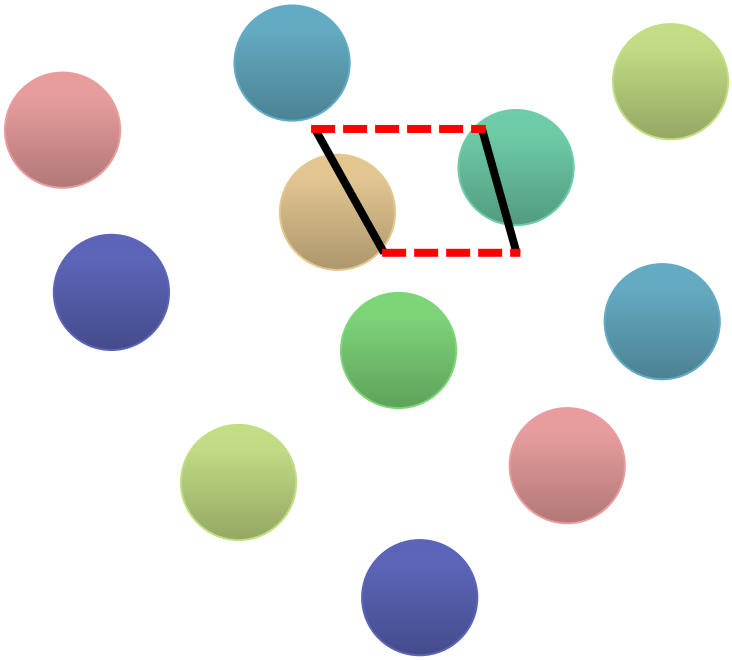
Gareth Thomas (AMD), *"Advancements in Tiled-Based Compute Rendering"*, GDC 2015

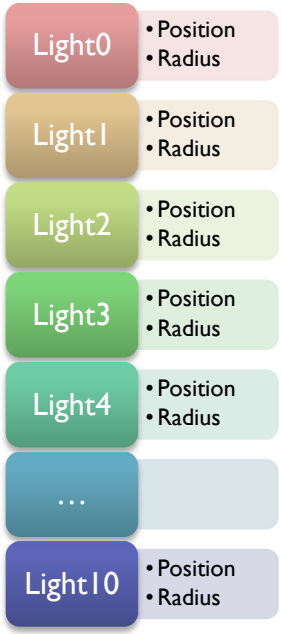
3. calcul du z_{min} et z_{max} par tuile



Gareth Thomas (AMD), *"Advancements in Tiled-Based Compute Rendering"*, GDC 2015

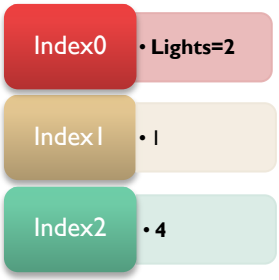
4. *light culling*

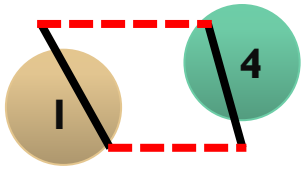


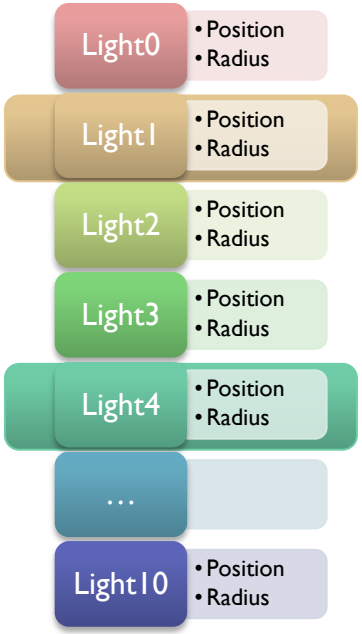


Gareth Thomas (AMD), *"Advancements in Tiled-Based Compute Rendering"*, GDC 2015

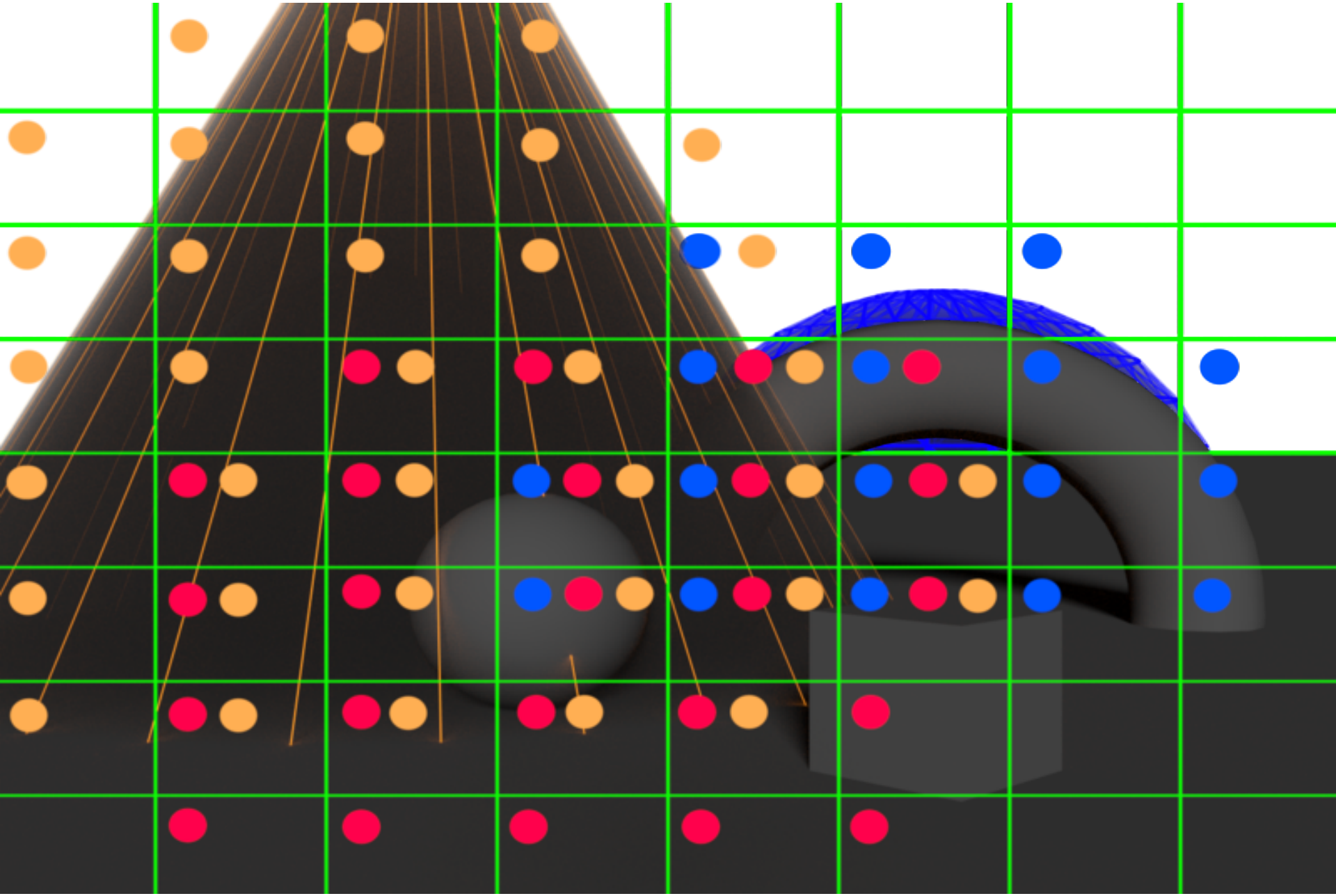
4. *light culling*



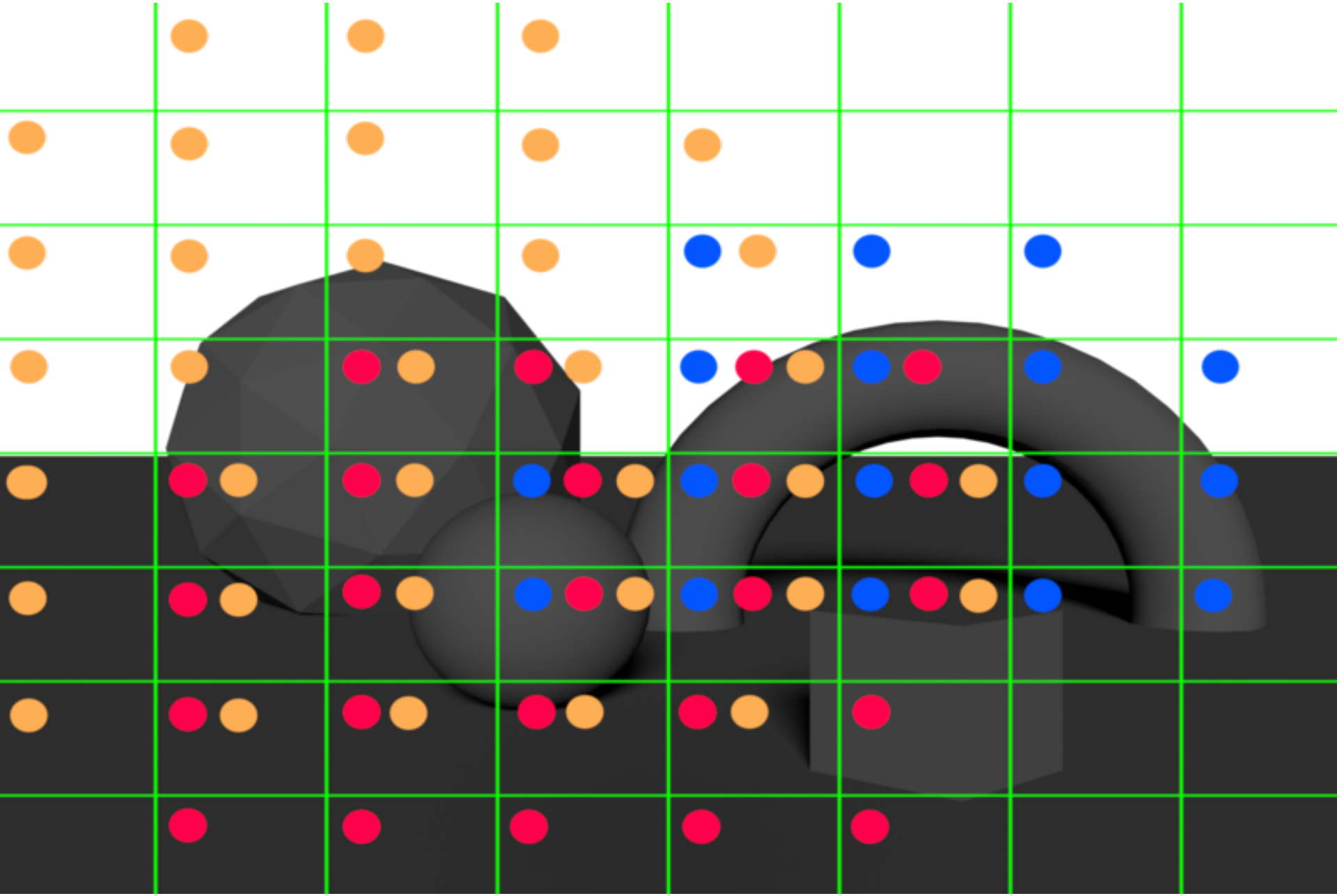




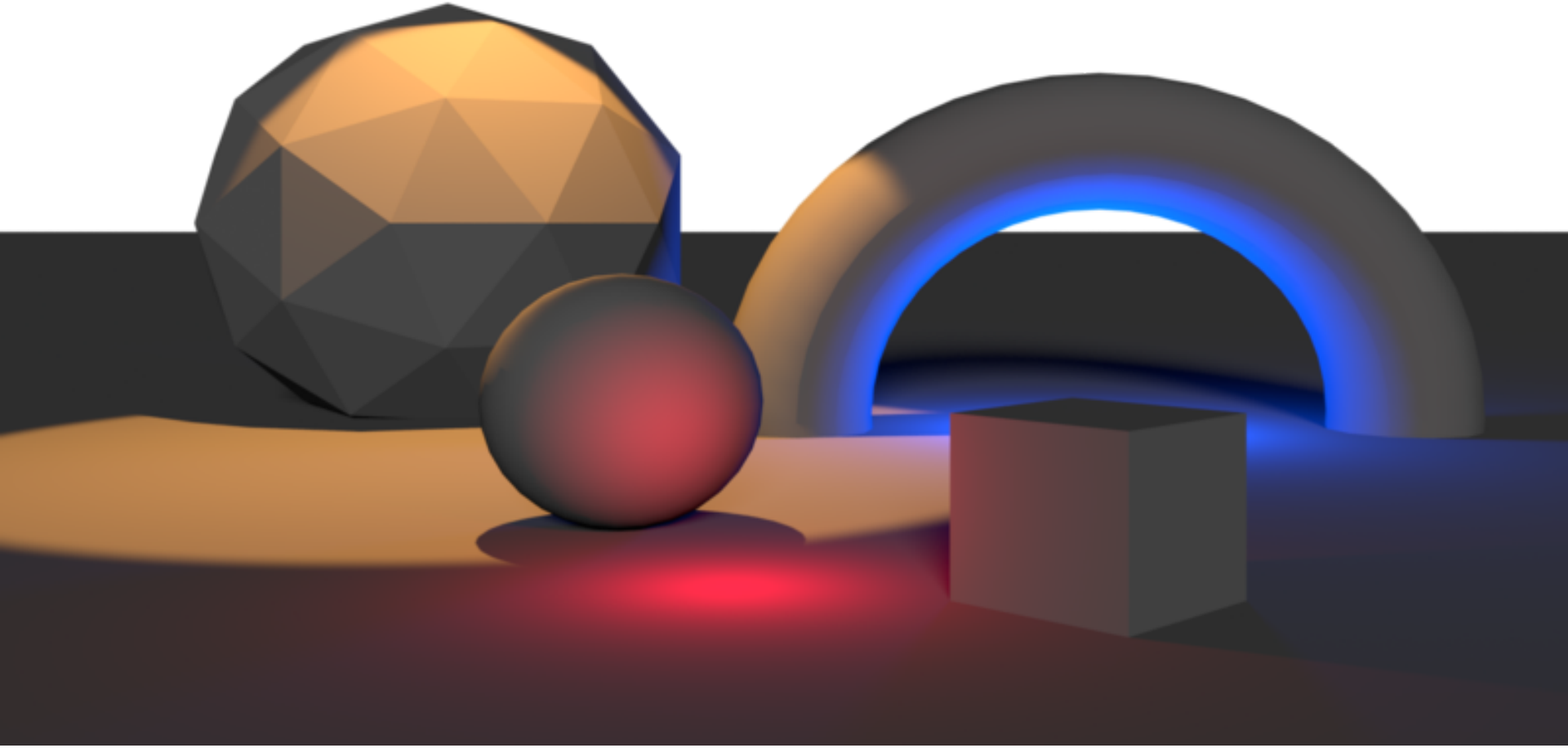
Gareth Thomas (AMD), *"Advancements in Tiled-Based Compute Rendering"*, GDC 2015



http://www.aortiz.me/2018/12/21/CG.html



http://www.aortiz.me/2018/12/21/CG.html

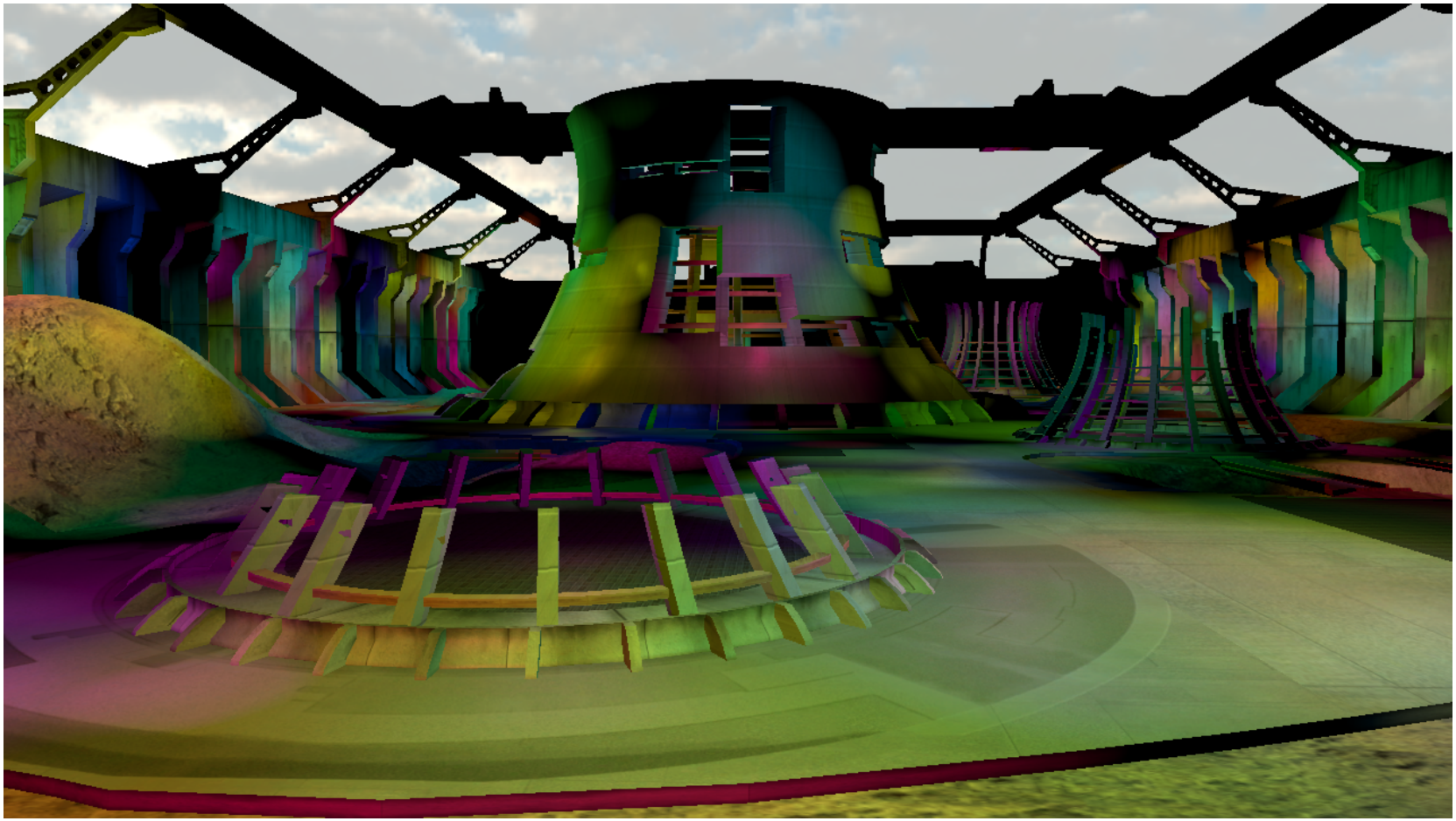


http://www.aortiz.me/2018/12/21/CG.html

Optimisation - Rendu par tuiles

[Andersson 2009]

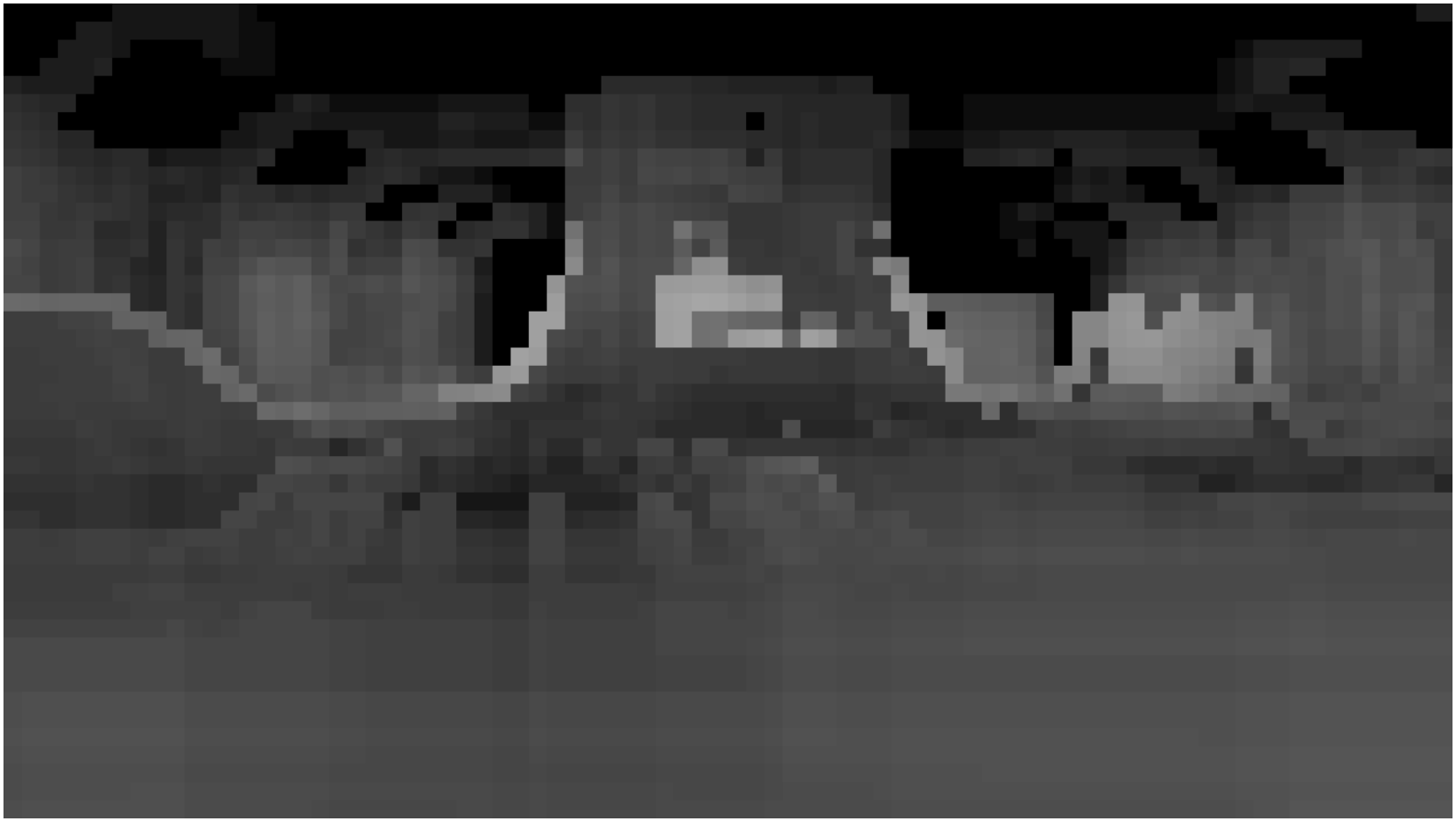
1024 sources ponctuelles



Optimisation - Rendu par tuiles

[Andersson 2009]

Nombre de sources évaluées par tuile (16x16 pixels)

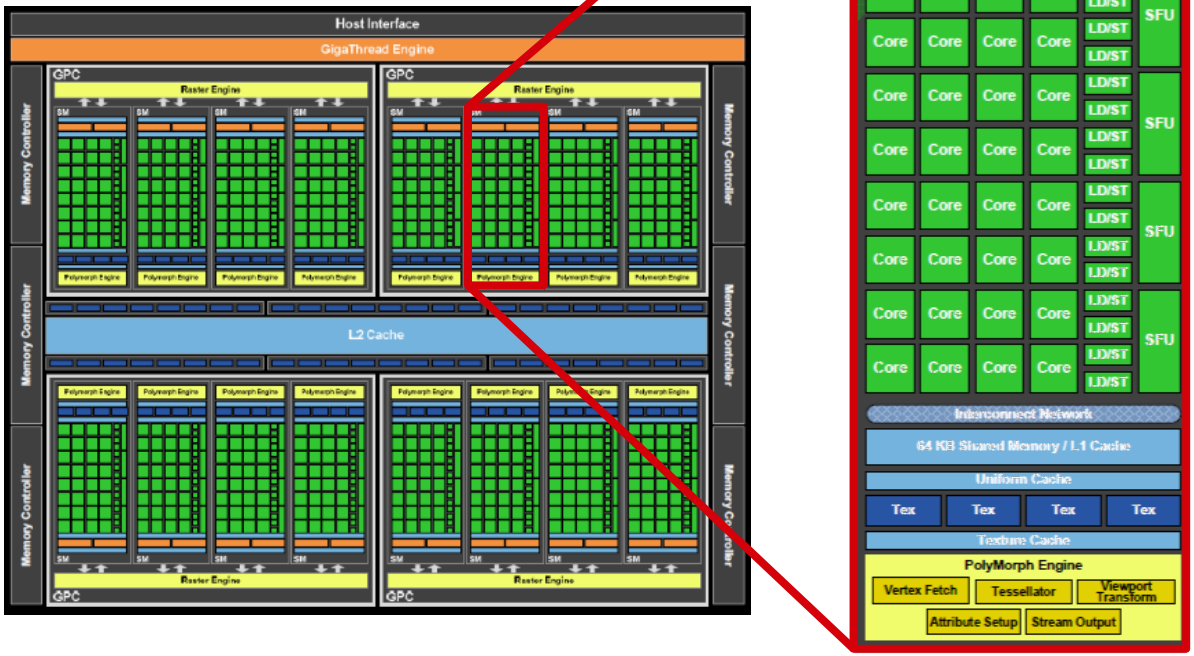


Optimisation - Rendu par tuiles

Importance de la taille des tuiles

- Si trop grandes ⟹ trop de sources dans une même tuile
- Si trop petites ⟹ *light culling* très long
  et faible cohérence du GPU

Compromis : taille d'un groupe de *threads*



Phase 2 : shading

Difficultés

- ~~Quid des sources lumineuses multiples ?~~
- **Matériaux multiples ?**
- Antialiasing ?
- Transparence ?

Matériaux multiples

Stocker tous les attributs des matériaux

- Ex : coefficients et exposants spéculaire de Blinn-Phong
- Limité à **un seul type** de matériau

Stocker l'index du matériau

- Attributs des matériaux stockés dans des tableaux (ou textures)
  `float specular_exp = specular_exponents[gbufffer.matid]`
- Shader générique quitte à réaliser des opérations inutiles
- Si **matériaux très différents,** 2 solutions :
  - branchements dynamiques dans le shader ⟹ surcoût à l'exécution
  - une passe par type de matériaux ⟹ *bandwidth*

Matériaux multiples

Stocker un ID (bitmask) **par tuile** (ex. : 16x16 pixels)

Éviter branchement si tous les pixels ont la même valeur

***Uncharted 4* -** Naughty Dog

Phase 2 : shading

Difficultés

- ~~Quid des sources lumineuses multiples ?~~
- ~~Matériaux multiples ?~~
- **Antialiasing ?**
- Transparence ?

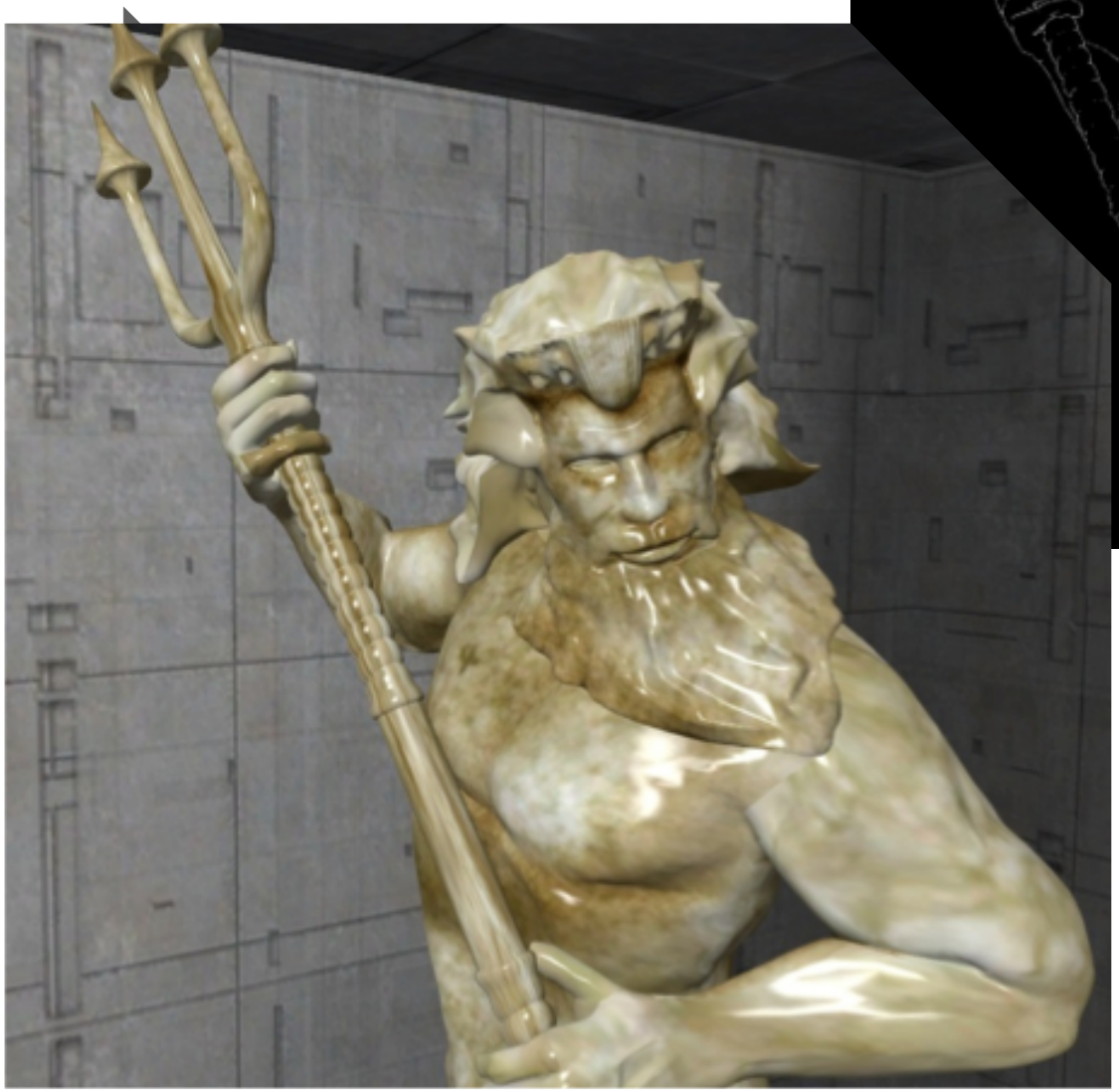
Antialiasing

Multi-Sampling Anti-Aliasing (MSAA)

- Utiliser des g-buffers avec plusieurs échantillons par pixel
  $\Longrightarrow$ coût mémoire et *bandwidth* ! (x4, x8)
- Crénelage apparaît uniquement aux **silhouettes**
  - silhouettes = discontinuité de z et de normales
  - filtre de Sobel appliqué au *z-buffer* et *normal buffer*
- Pour les pixels non détectés : calcul de l'éclairage par pixel
- Pour les pixels silhouettes : calcul de l'éclairage par échantillon
  (utilisation du *stencil buffer* possible)

$\Longrightarrow$ Deux passes de rendu

MSAA

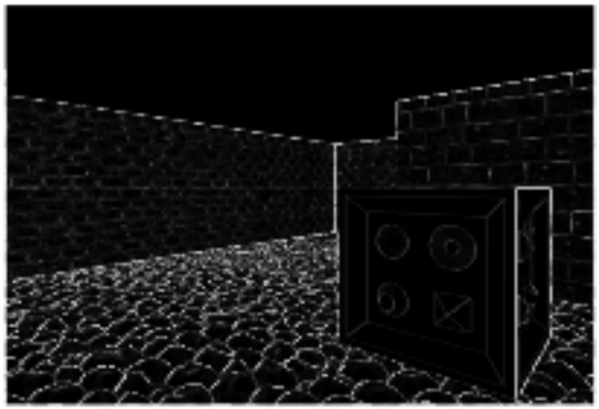


Détection des silhouettes et échantillonnage sélectif

Crytek [Sousa 2013]

MSAA

Détection des silhouettes



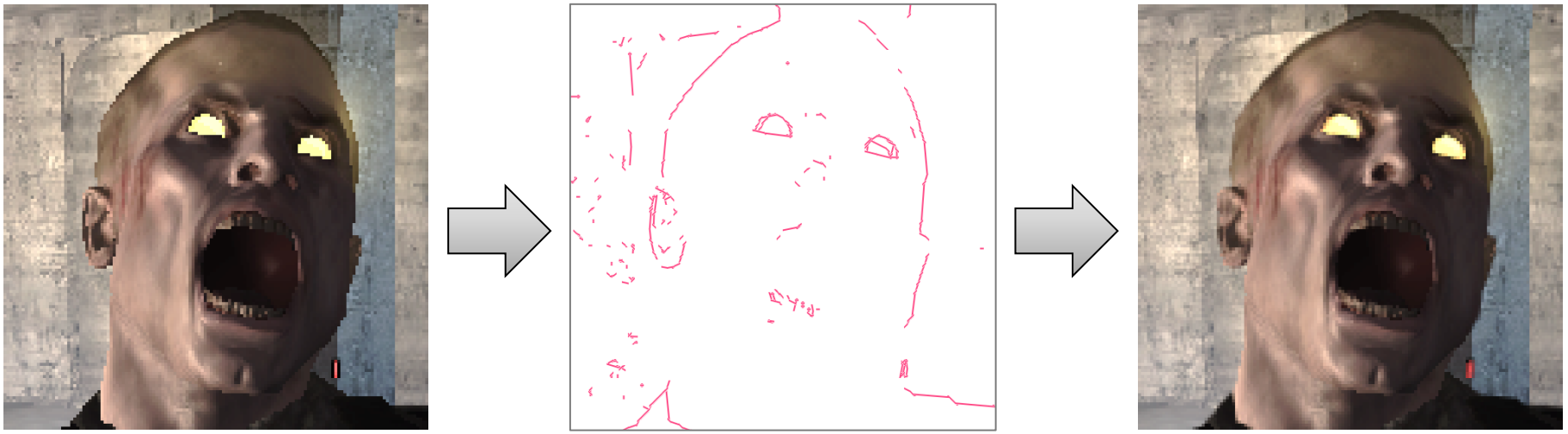
Échantillonnage sélectif

Antialiasing

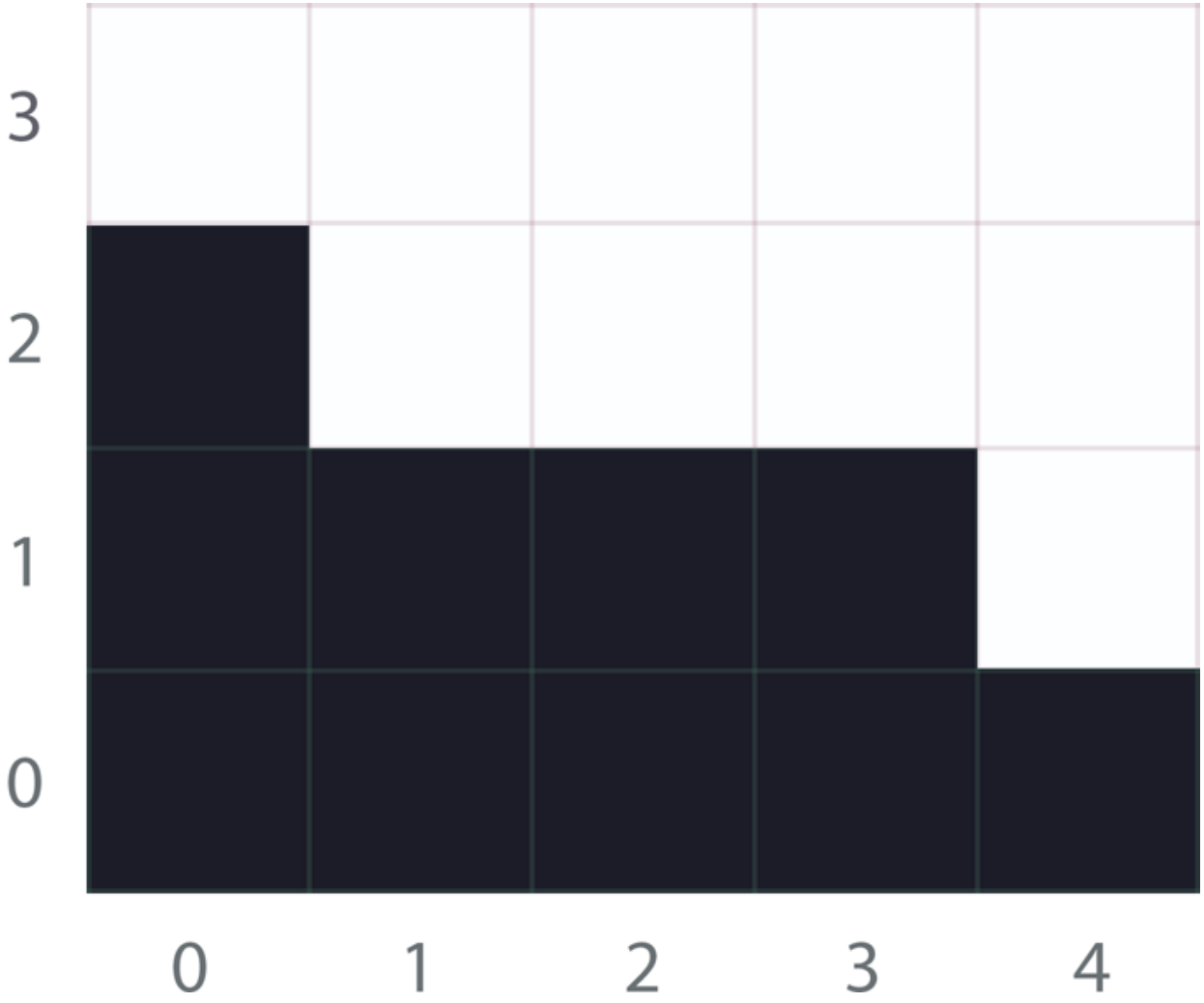
MorphoLogical AntiAliasing (MLAA) – AMD

Fast Approximate Anti-Aliasing (FXAA) - Nvidia

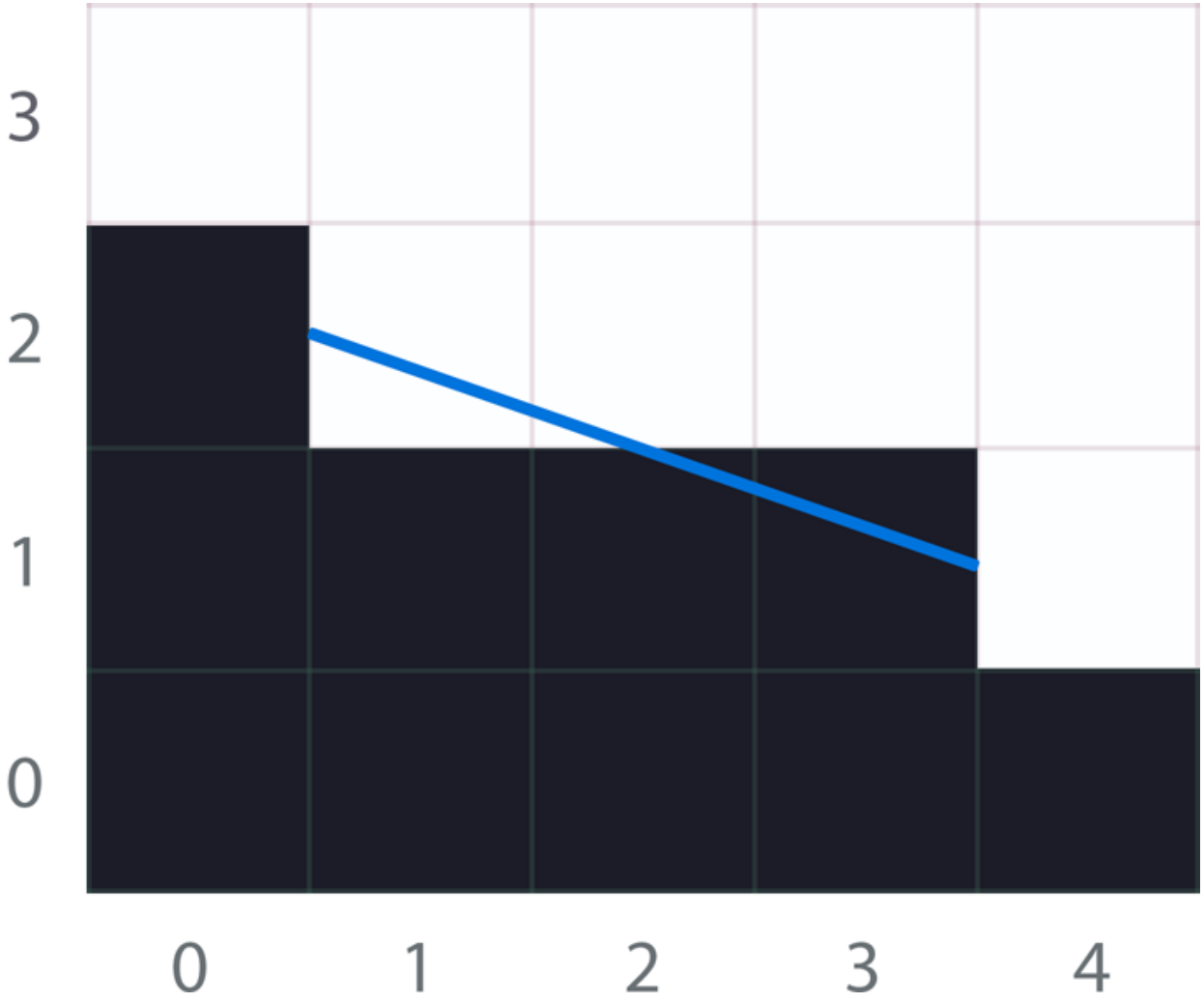
1. Détecter les silhouettes
2. Filtrer les couleurs près des silhouettes sans sur-échantillonner



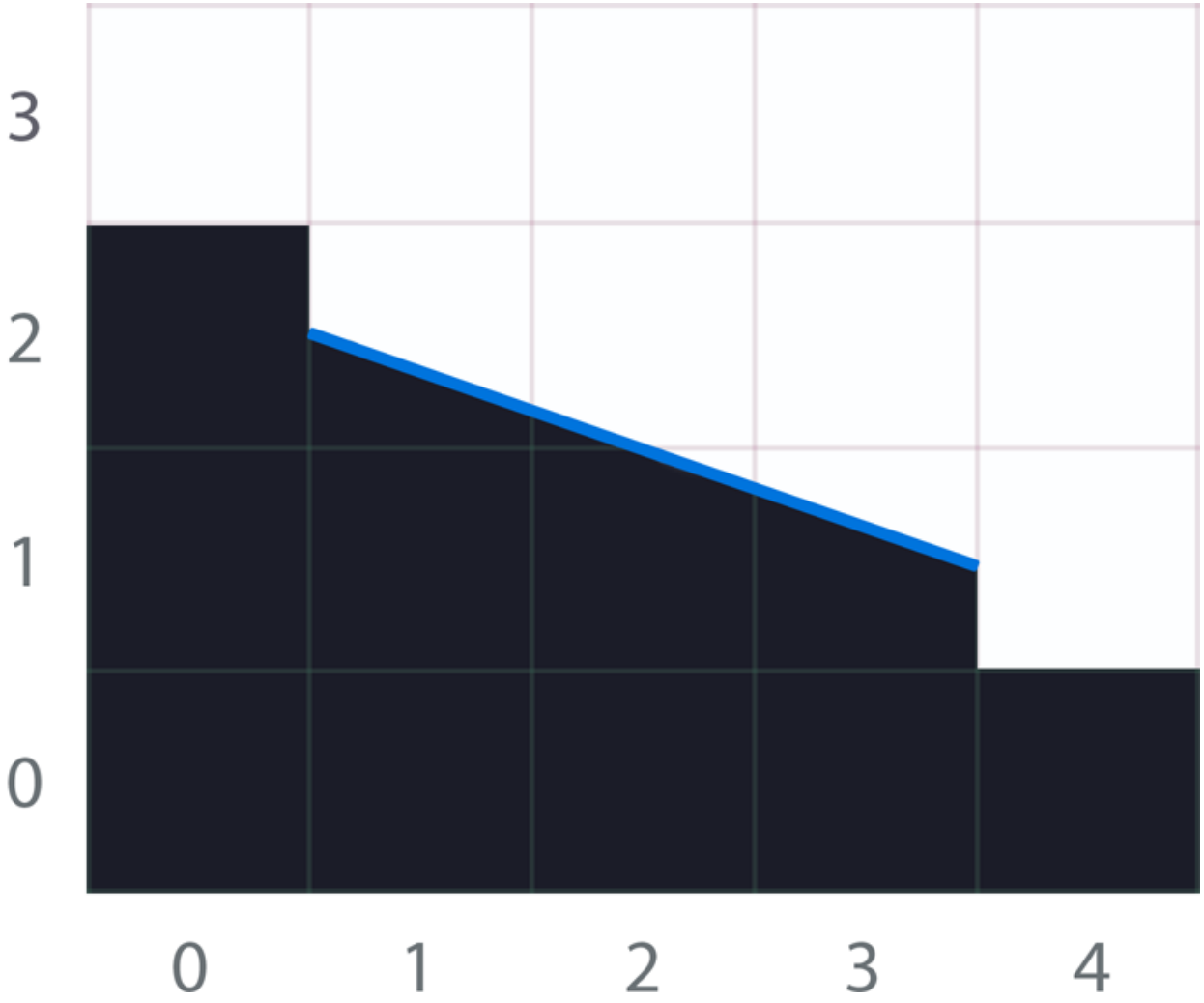
MLAA [Reshetov 2009]



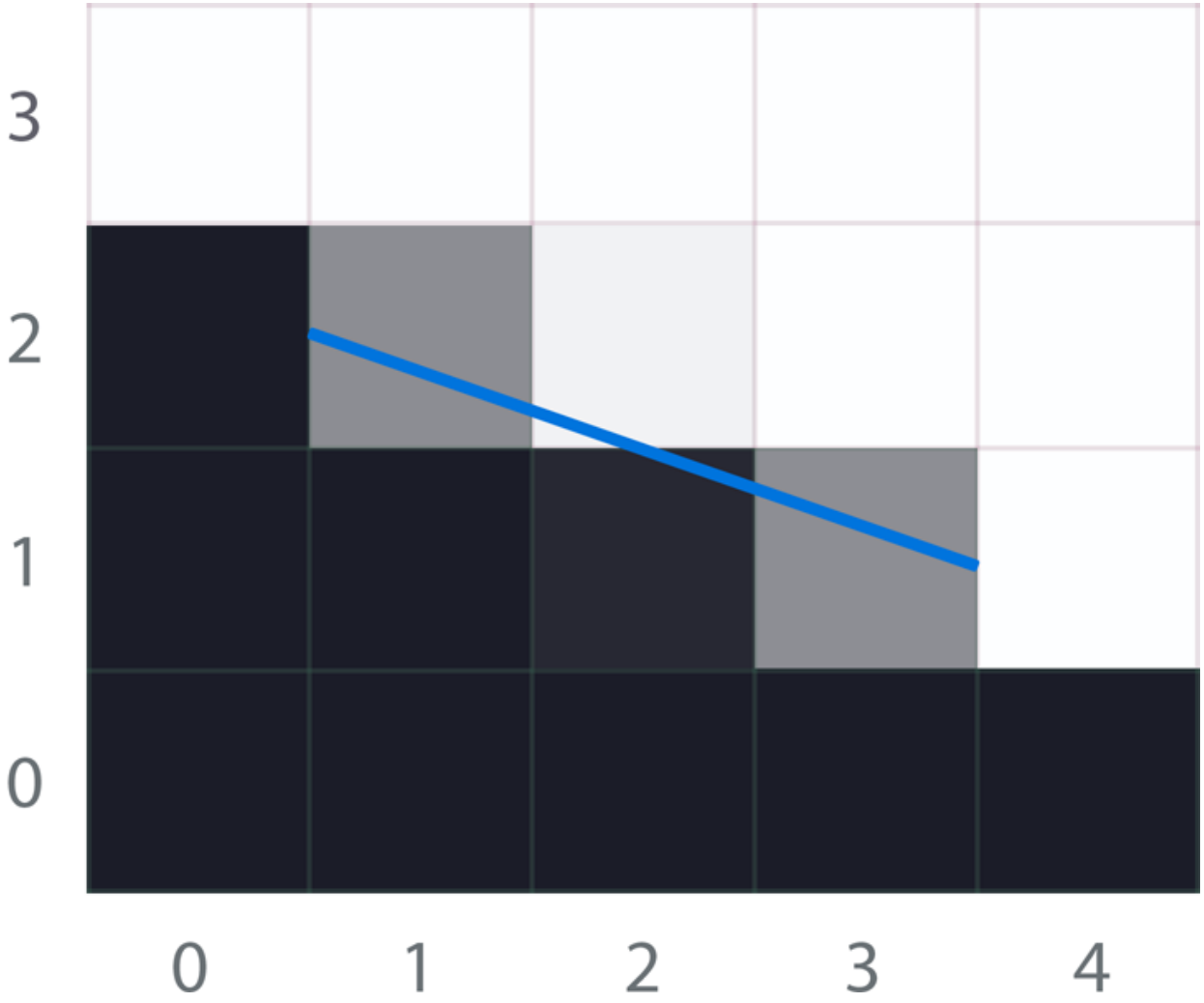
MLAA [Reshetov 2009]



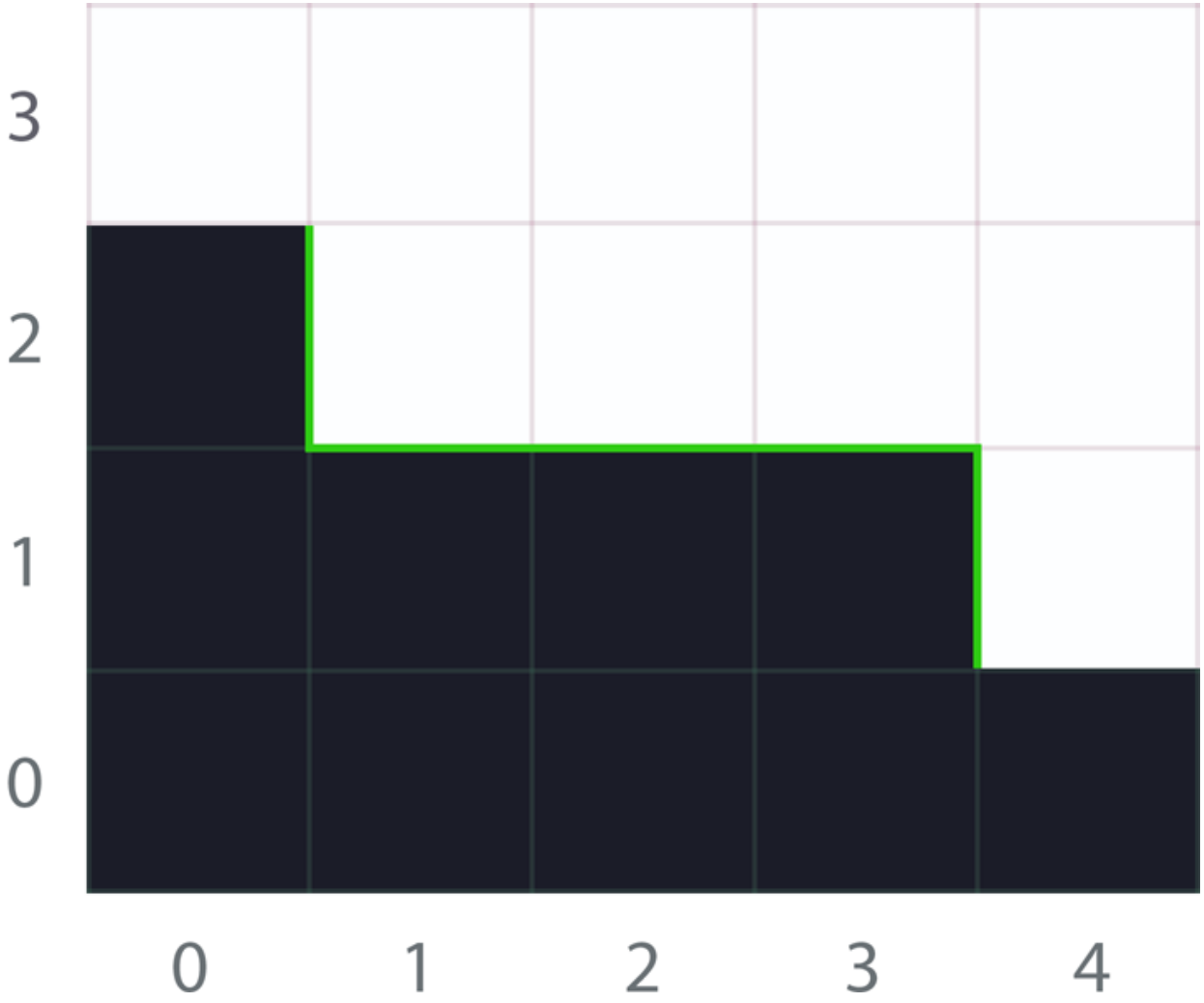
MLAA [Reshetov 2009]



MLAA [Reshetov 2009]

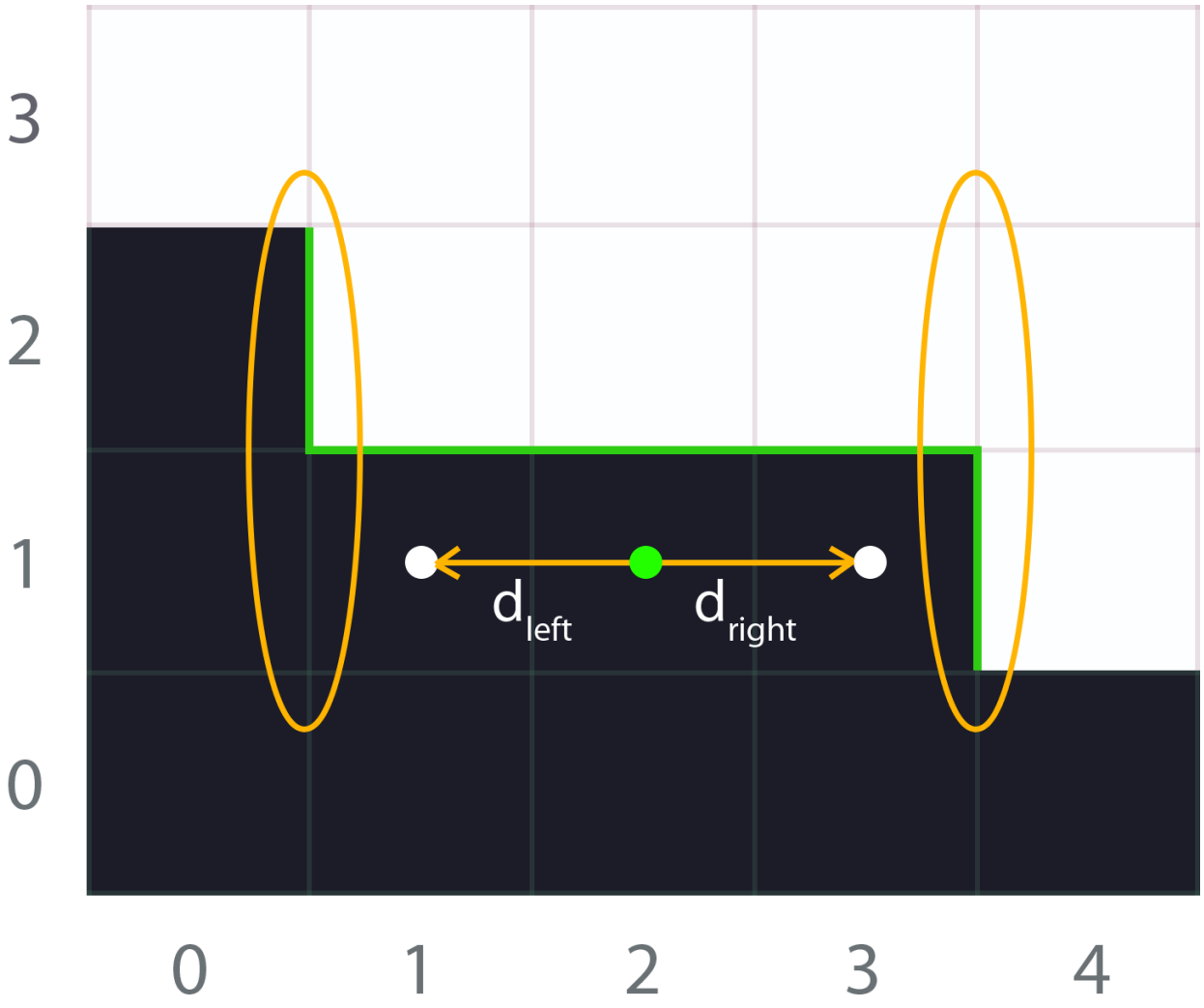


MLAA [Reshetov 2009]

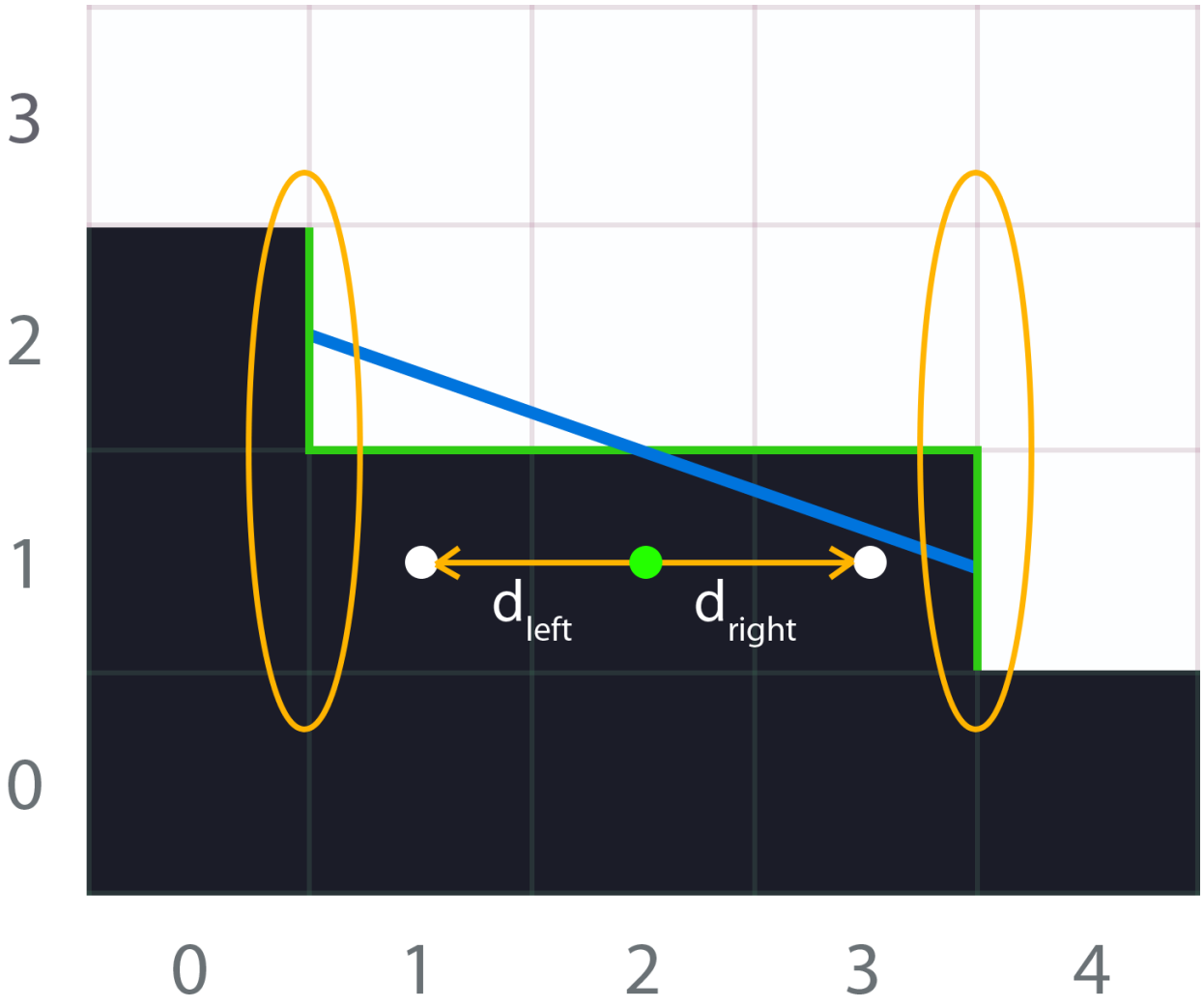


MLAA [Reshetov 2009]

MLAA [Reshetov 2009]



MLAA [Reshetov 2009]



MLAA [Reshetov 2009]

MLAA

Très rapide mais approximations et artefacts

$\Longrightarrow$ de nombreuses extensions :

(Filmic) SMAA [Jimenez 2011-16]

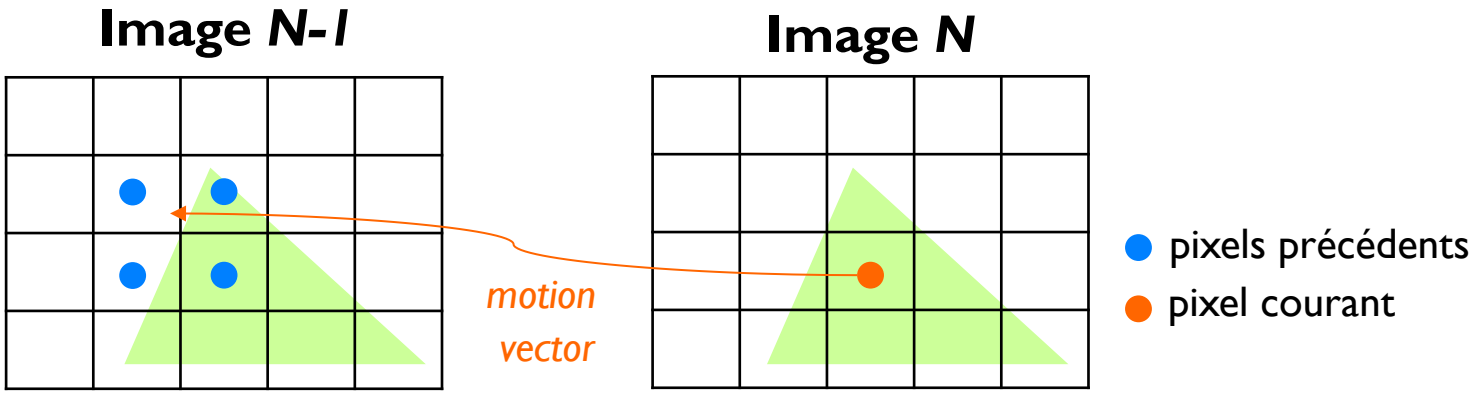
- Suppression des silhouettes de faible contraste $\Longrightarrow$ plus net
- Amélioration des performances
- Filtrage temporel $\Longrightarrow$ moins de scintillements

http://www.iryoku.com/aacourse/

Antialiasing

Temporal Anti-Aliasing (TAA)

- Utilisation d'échantillons d'**images précédentes** (*history buffer*) pour sur-échantillonner les pixels de l'image courante
- Nécessite le **champ de mouvement** de la scène entre deux images (*motion vectors*)
- Détecter les occlusions entre deux images
- Standard dans les moteurs de jeux (UE4, Untiy)



http://behindthepixels.io/assets/files/TemporalAA.pdf

Sans antialisaing

***Uncharted 4* - Naughty Dog**

FXAA

***Uncharted 4* - Naughty Dog**

TAA

***Uncharted 4* - Naughty Dog**

Phase 2 : shading

Difficultés

- ~~Quid sources lumineuses multiples ?~~
- ~~Matériaux multiples ?~~
- ~~Antialiasing ?~~
- **Transparence ?**

  ⟹ passe de rendu direct (*forward rendering*) des objets transparents

Deferred Shading - Conclusion

Implémenté dans tous les moteurs de rendu

(en standard dans *UE4*, *Frosbite*, optionnel dans *Unity*)

- Permet de mettre en œuvre des effets complexes **à moindre coût**
  - calculs uniquement sur les pixels visibles
  - effets en post-traitement simples
- Découplage de la géométrie et du calcul de l'éclairage
  ⟹ anti-aliasing plus complexe

Rendu direct

- Indispensable pour les effets de transparence
- Peut être plus efficace si peu de sources lumineuses

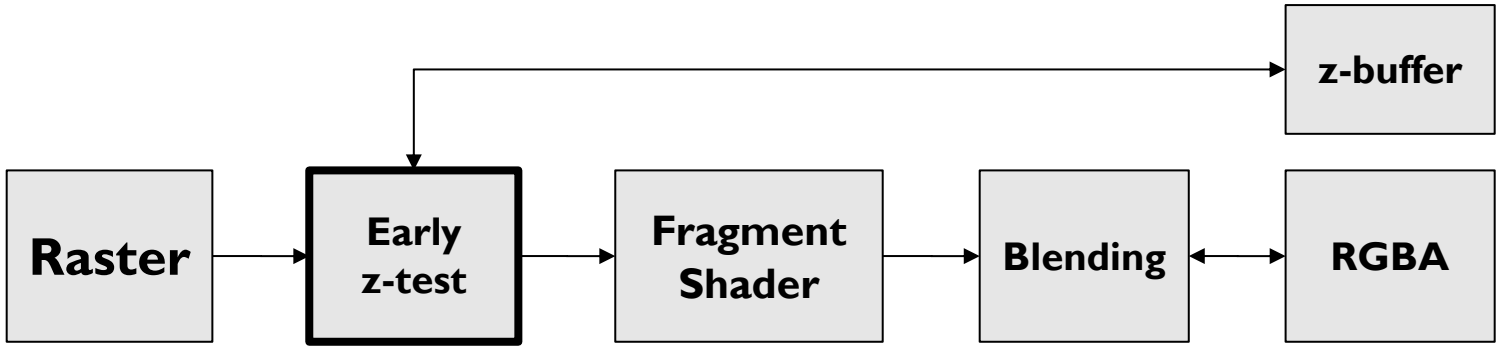
Early-Z rejection [Mitchel et al. 2004]

GPU

Élimine les fragments non visibles avant qu'ils soient traités par le fragment shader

Activé dans certain cas :

- si le fragment shader **ne modifie pas** la profondeur
- si le test de stencil est désactivé
- si le test de profondeur est LESS



Forward+ / *Tiled forward shading*

[Harada et al. 2012]

Extension du rendu direct

1. *Depth pre-pass* : remplissage du *z-buffer*
2. *Light culling* : calcul d'une liste de lampes visibles par tuile
3. *Shading* : calcul de l'éclairage

10000 **petites** lampes ponctuelles, 60fps

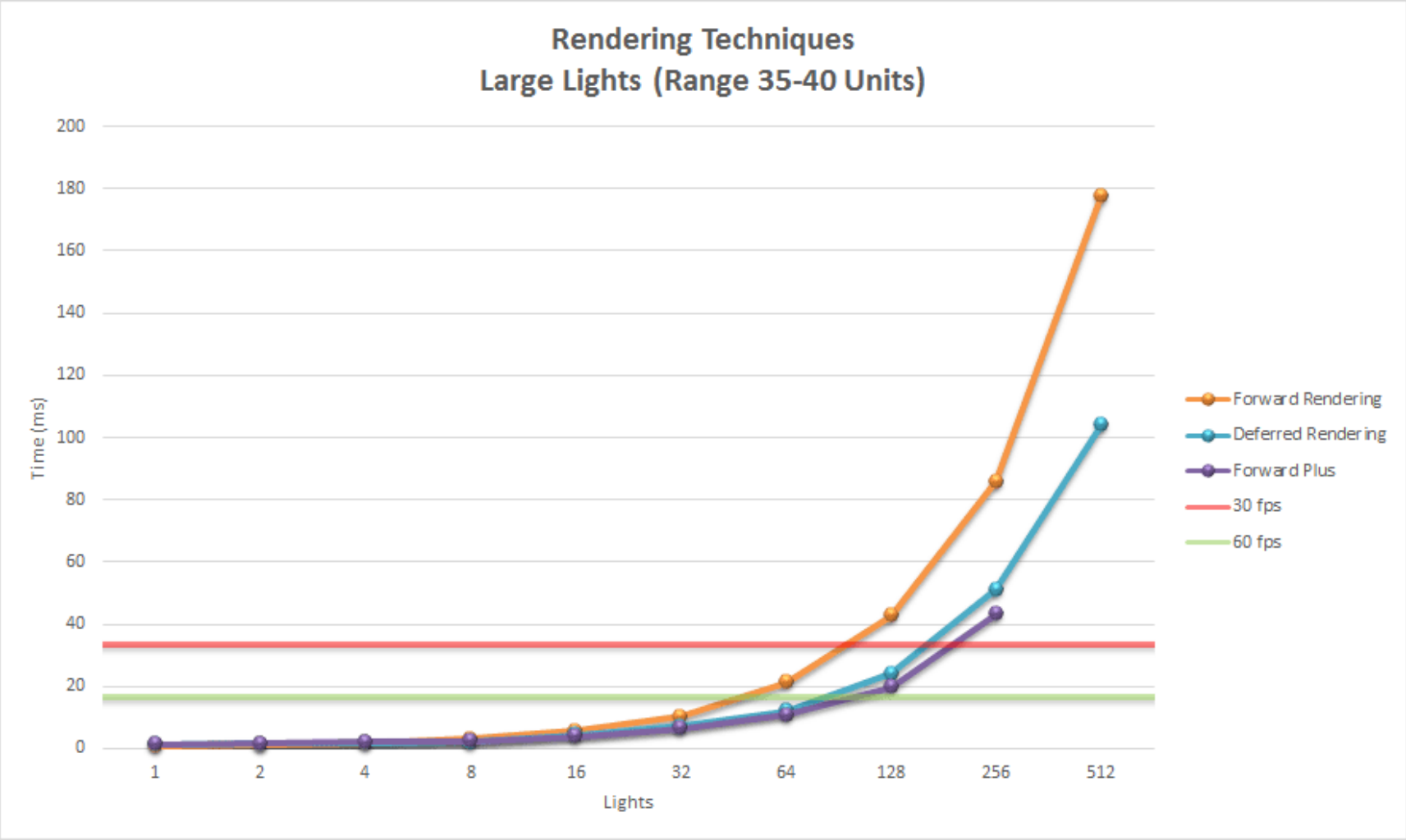
Utilisé dans :

- *DiRT & GRID Series*
- *The Order: 1886*
- *Ryse*

http://www.3dgep.com/forward-plus/#Forward

Comparaisons

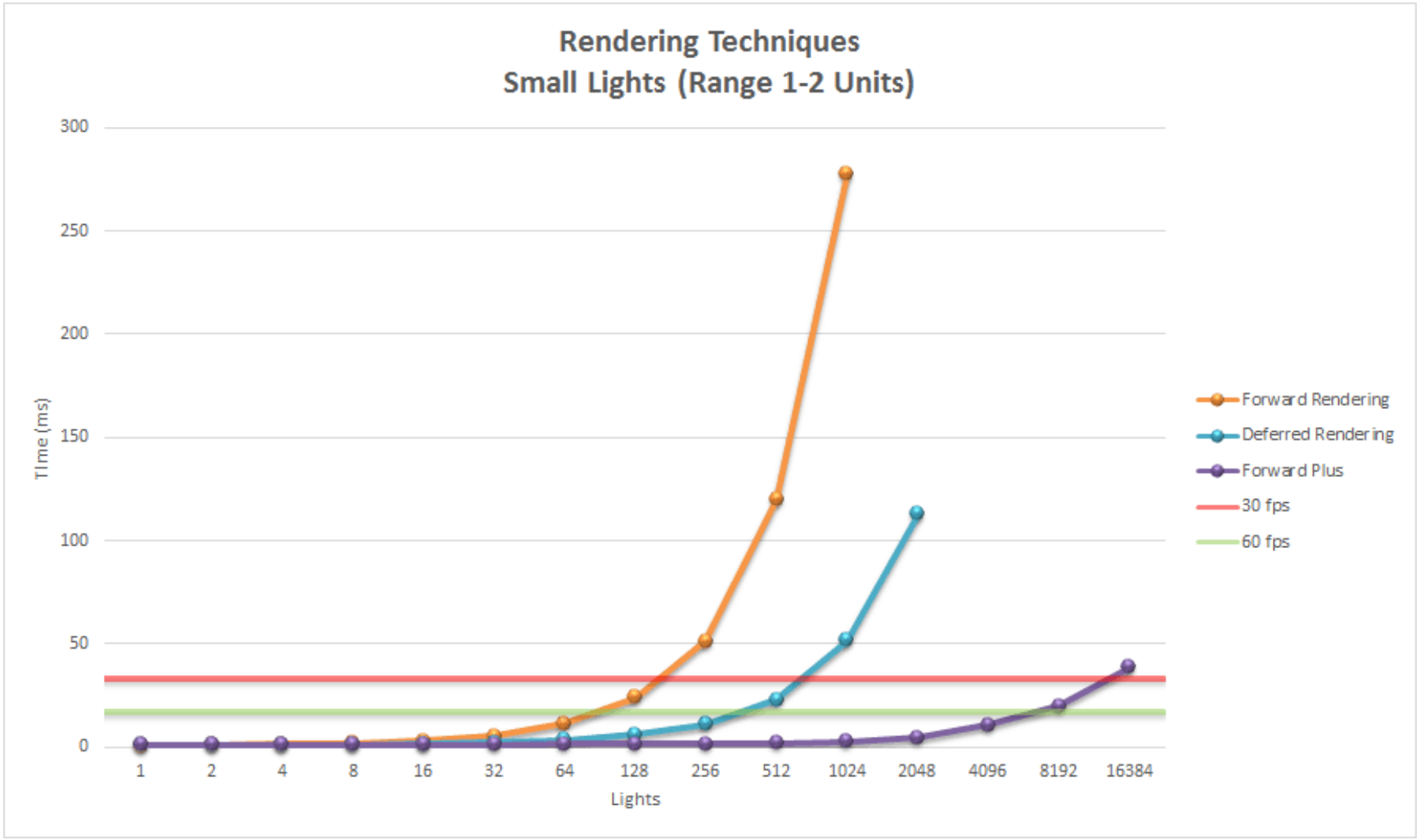
Sponza - NVIDIA GeForce GTX 680 – résolution 1280x720



http://www.3dgep.com/forward-plus/#Experiment_Setup_and_Performance_Results

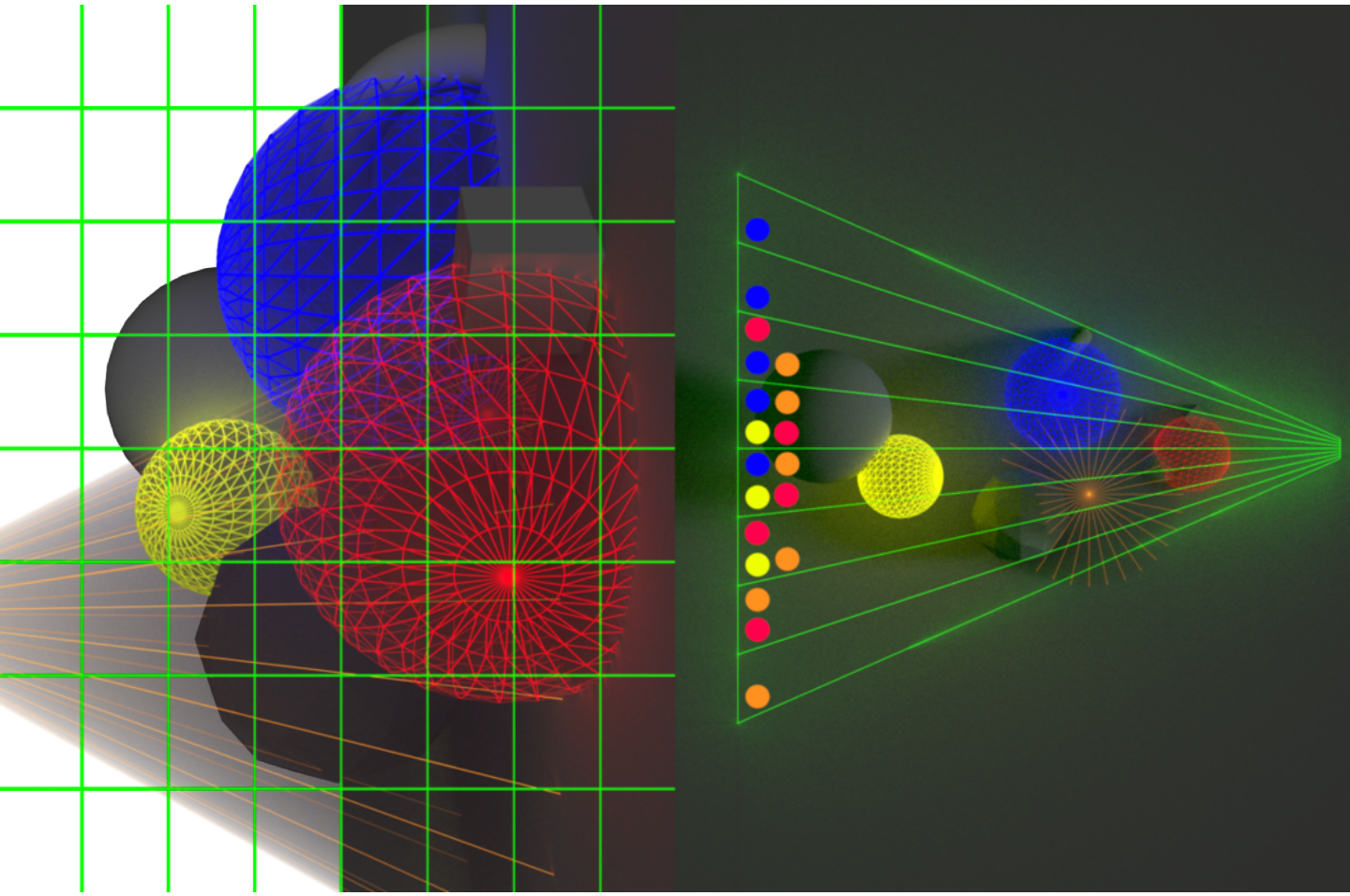
Comparaisons

Sponza - NVIDIA GeForce GTX 680 – résolution 1280x720

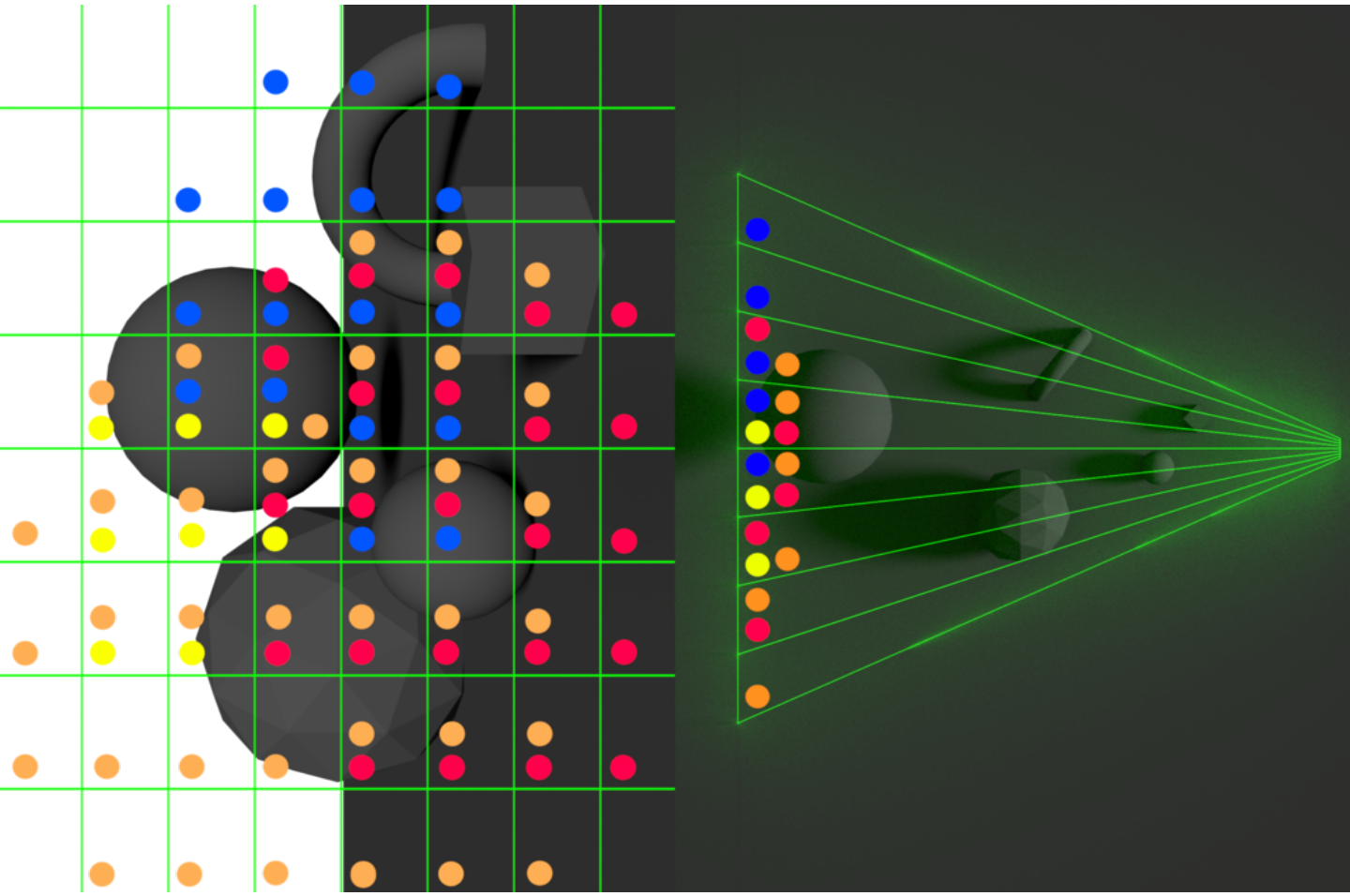


http://www.3dgep.com/forward-plus/#Experiment_Setup_and_Performance_Results

http://www.aortiz.me/2018/12/21/CG.html



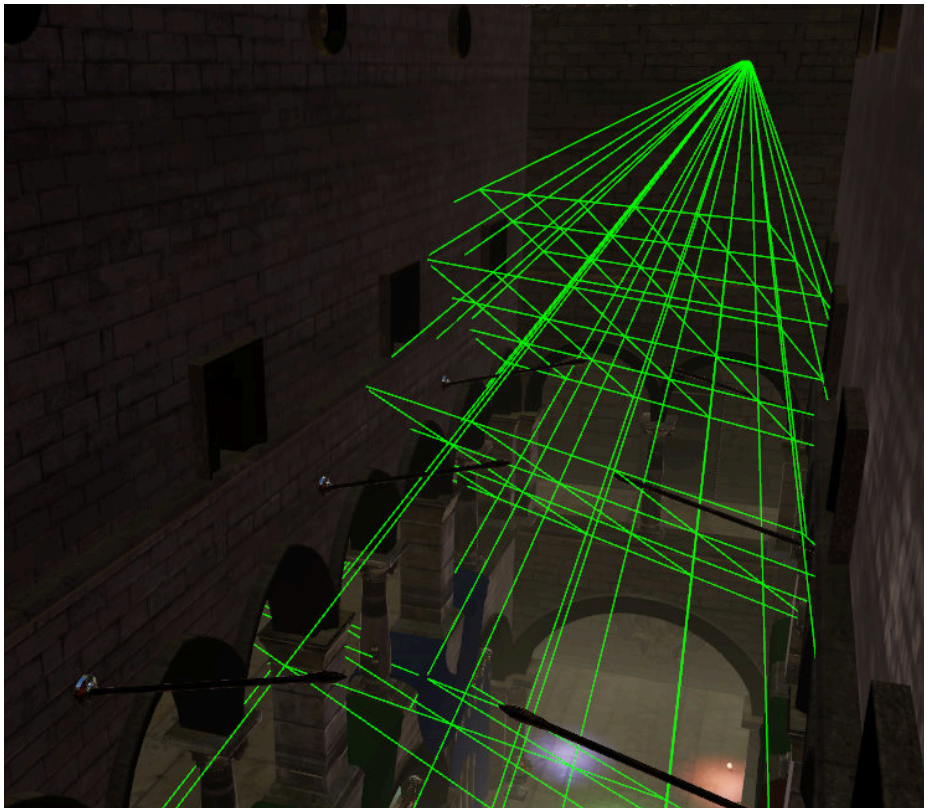
http://www.aortiz.me/2018/12/21/CG.html



Clustered shading [Olsson 2012]

Applicable au rendu direct ou *deferred*

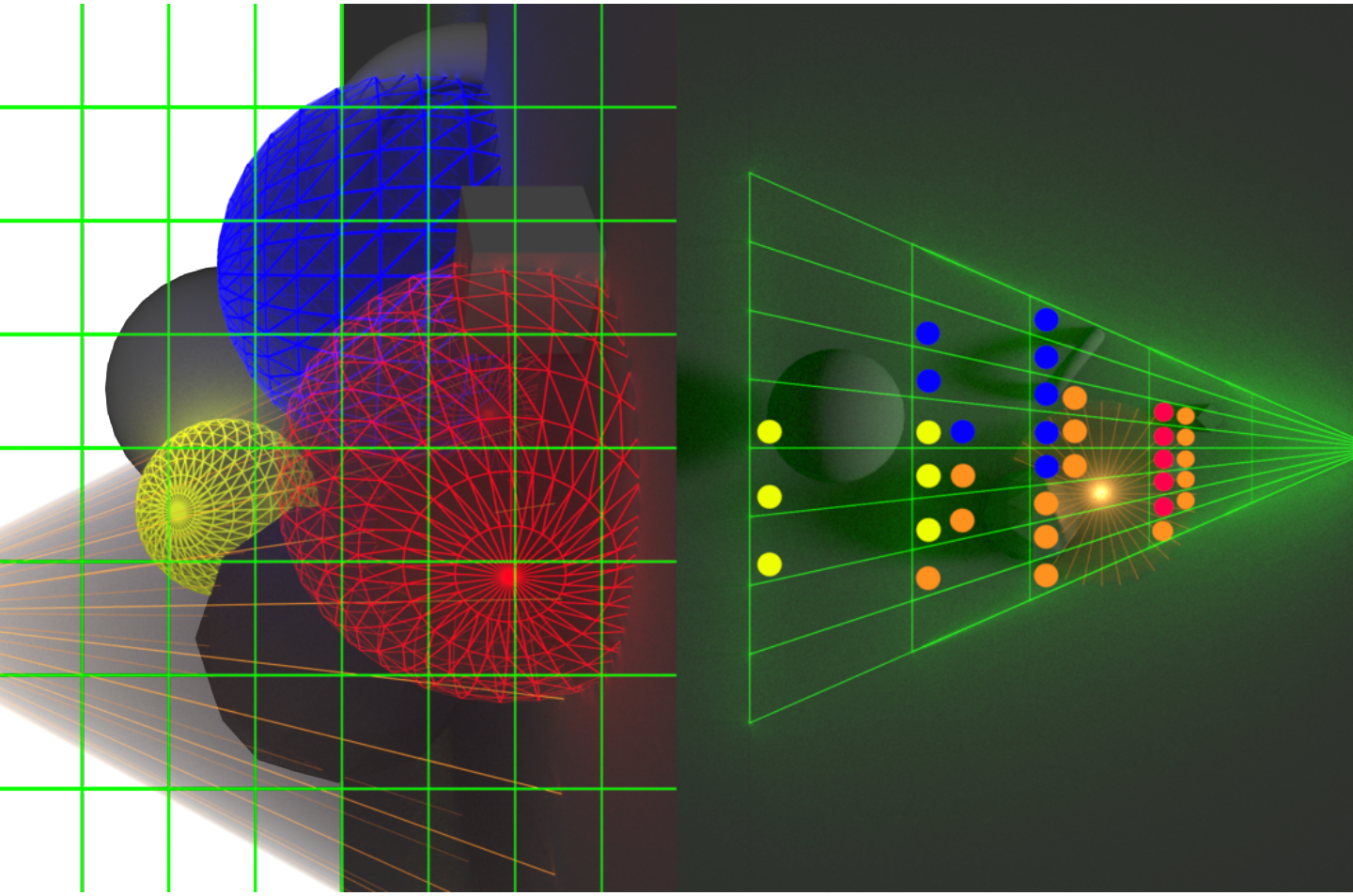
- *Depth pre-pass* optionnelle
- Indépendant de la complexité en profondeur



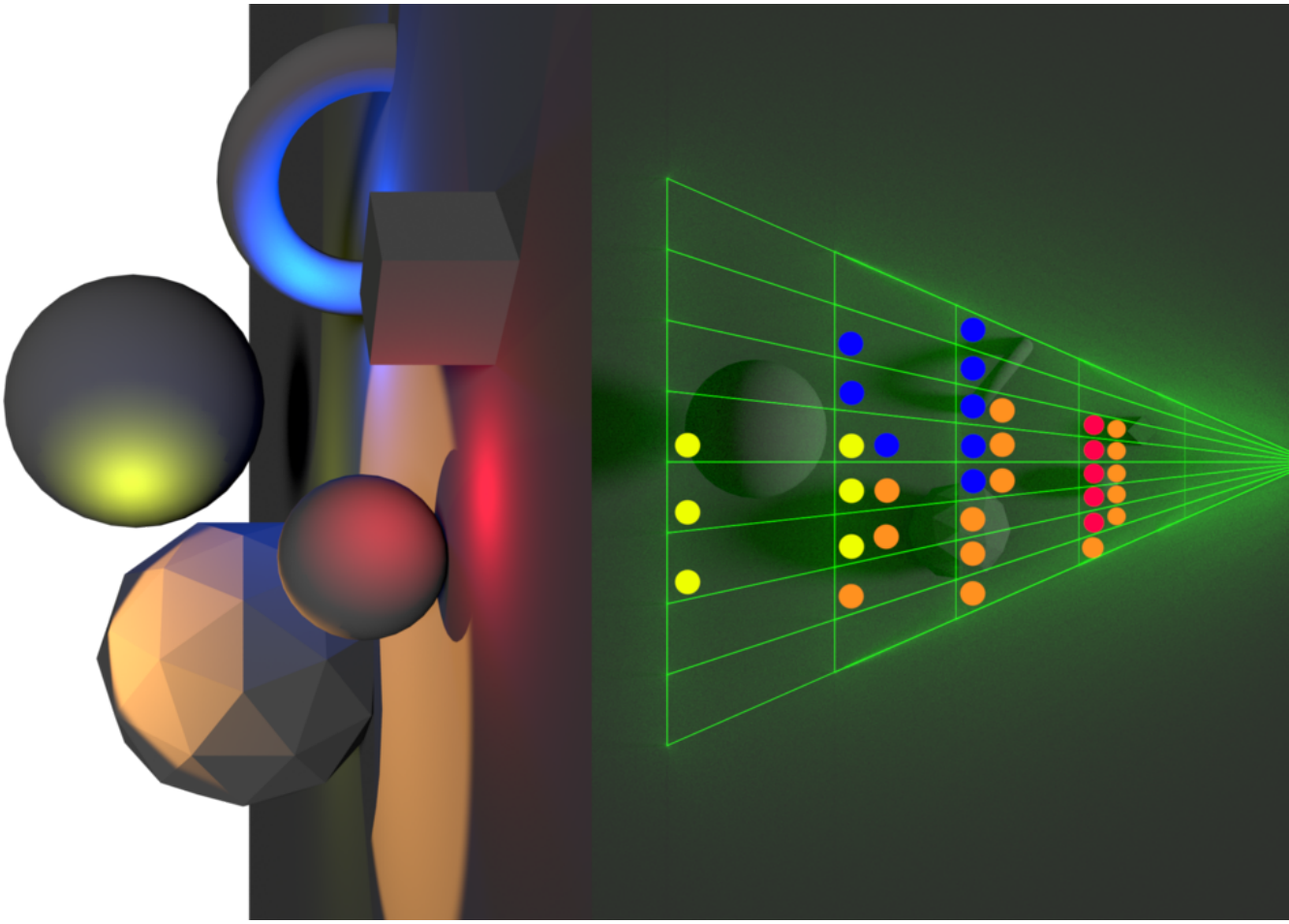
Utilisé dans :

- *Just Cause 3*
- *Forza Horizon 2*
- *DOOM (2016)*
- *Detroit: Become Human*
- *Unreal Engine (deferred)*
- *Unity HDRP*
- *Google Filament*

http://www.aortiz.me/2018/12/21/CG.html



http://www.aortiz.me/2018/12/21/CG.html



Bilan

	Simple Forward	Simple Deferred	Tile/Clust. Deferred	Tile/Clust. Forward
Geometry passes	1	1	1	1-2
Light passes	0	1 / light	1	1
Light culling	per mesh	per pixel	per volume	per volume
Transparent	easy	no	with fwd.	easy
MSAA	built-in	hard	hard	built-in
Bandwidth	low	high	medium	low
Small triangles	slow	fast	fast	slow
Vary shading	simple	hard	medium	simple

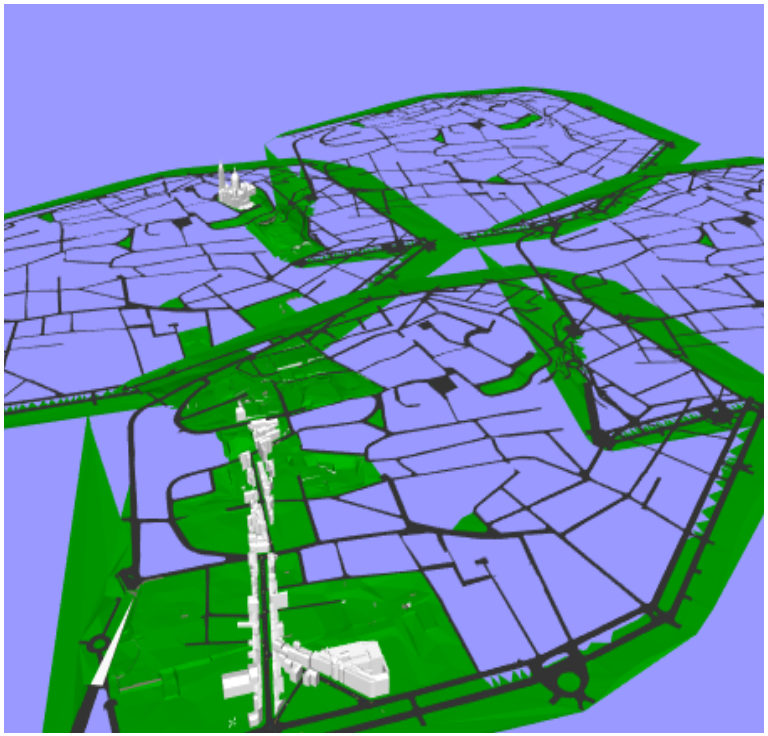
Pour un GPU de **bureau**, contraintes différentes sur mobile.

« *Real-Time Rendering*, *4th edition* », AK Peters

Visibilité

Pour un point de vue donnée que voit-on ?

Ou plutôt **que ne voit-on pas ?**



Visibilité

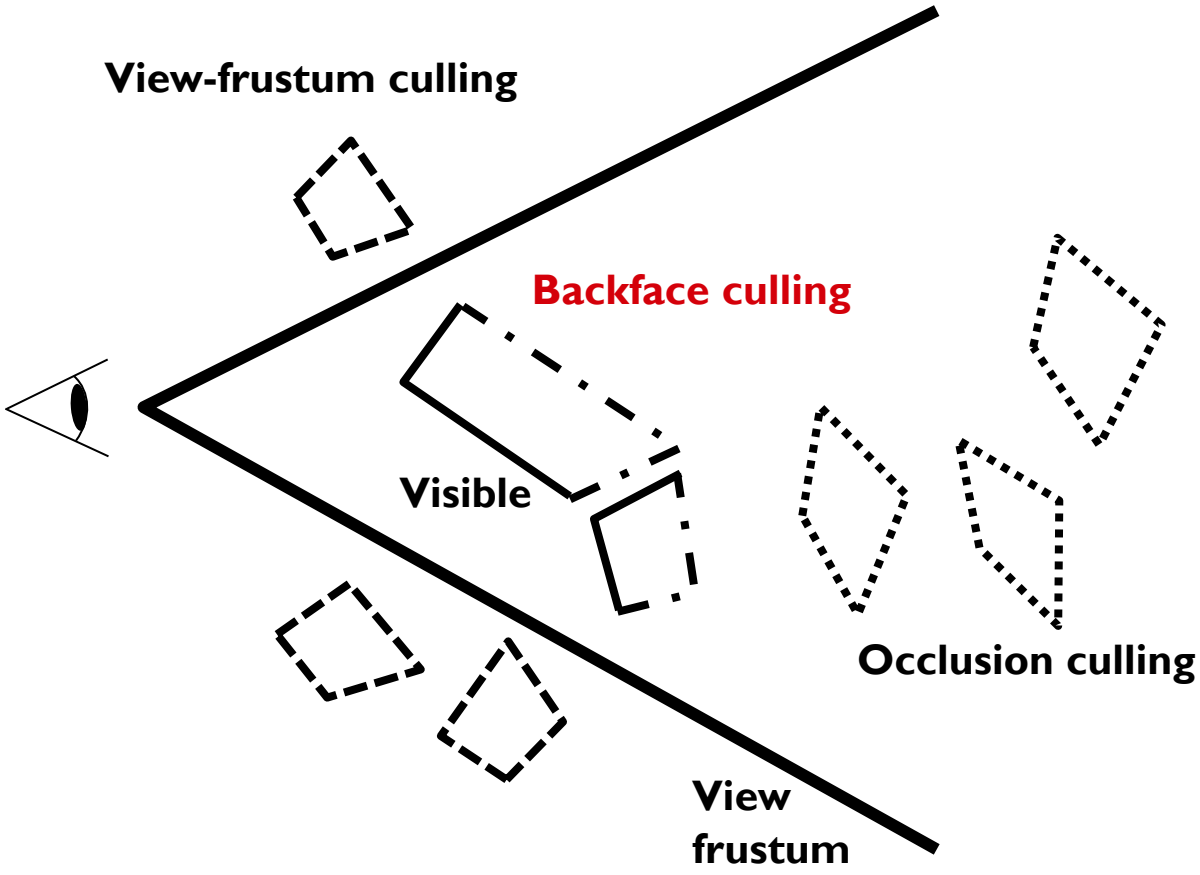
Que ne voit-on pas ?

- Ce qui est caché par un objet
- Ce qui est plus petit qu'un pixel
- Ce qui est en dehors du volume de vue
- Ce qui est complètement dans l'ombre

⟹ ***culling*** et ***clipping***

- Éliminer les objets **le plus tôt possible**
- Avant le *z-buffer* pour ne pas avoir à traiter tous les polygones

Culling



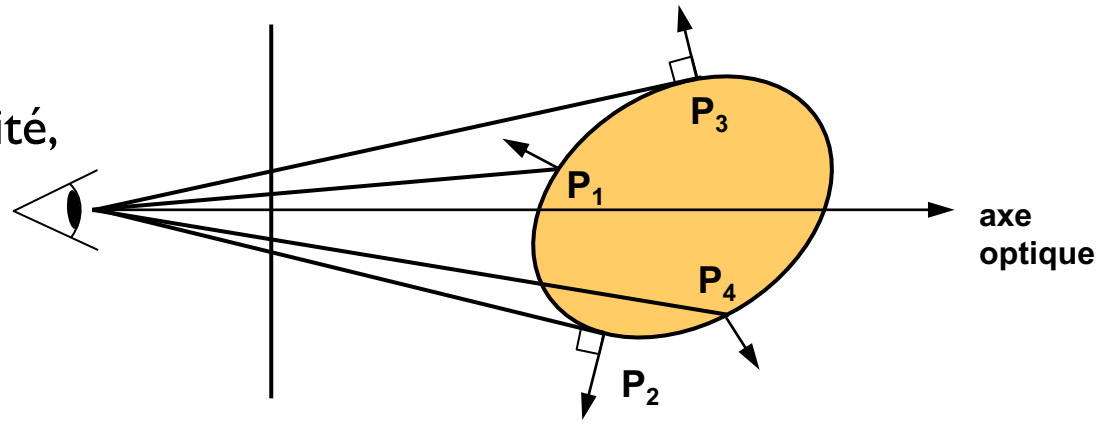
A Survey of Visibility for Walkthrough Applications,
Cohen-Or, Chrysanthou, Silva 2003

Backface culling

GPU

Éliminer les parties de la surface dont la **normale** pointe dans la **direction opposée au point de vue.**

P_1 est visible, P_2 et P_3 sont à la limite de visibilité, P_4 n'est pas visible (si l'objet est opaque !)



La suppression des faces arrières suffit :

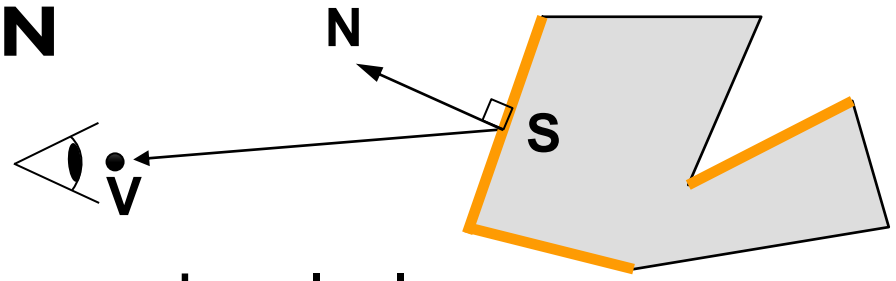
1. si l'objet est seul dans la scène
2. si l'objet est convexe ; les faces avant d'un objet concave peuvent être masquées par d'autres faces avants

Backface culling

GPU

Produit scalaire (V – S) • N

- > 0 : on garde le polygone
- < 0 : on l'élimine



Économise jusqu'à **50%** du temps de calcul

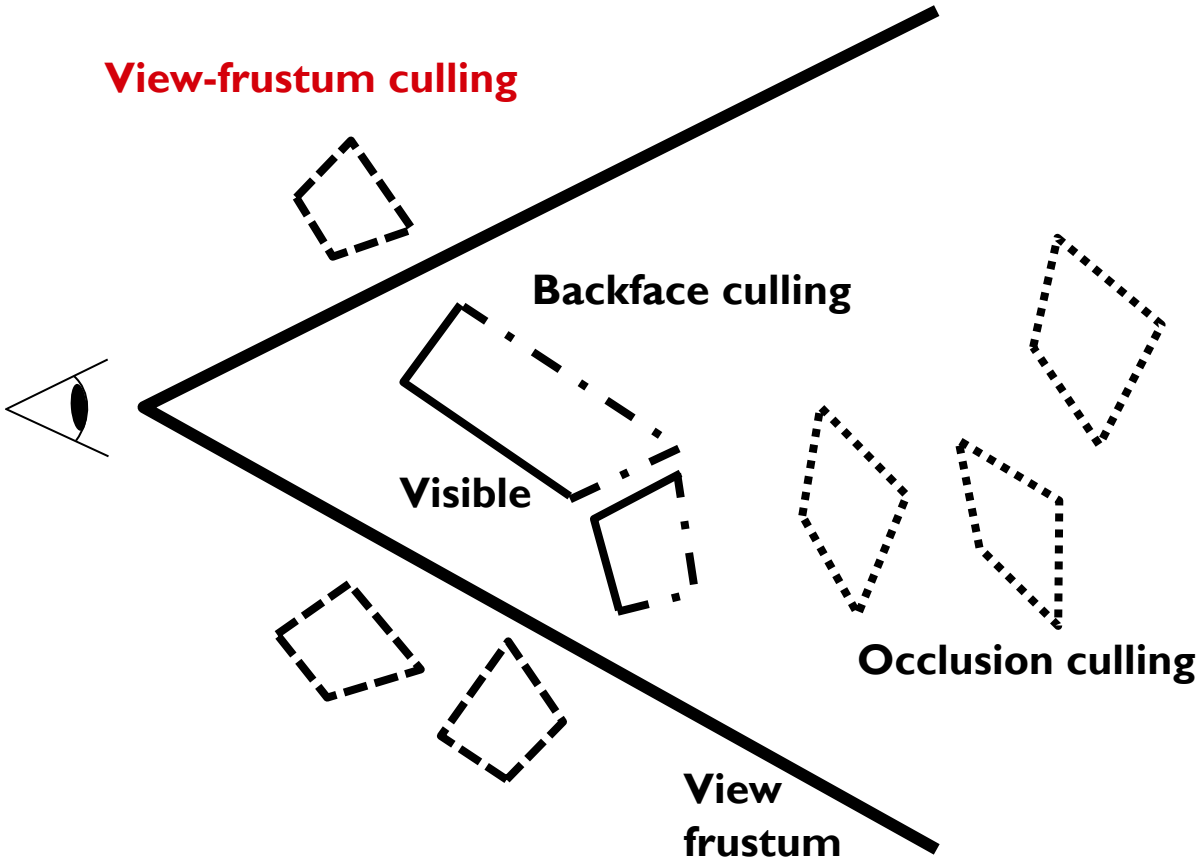
- beaucoup moins pour certaines scènes :
  les murs, sols, terrains font essentiellement faces à la caméra

Préalable aux autres algorithmes

En OpenGL :

```
glCullFace(GL_BACK)
glEnable/Disable(GL_CULL_FACE)
```

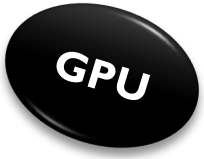
Culling

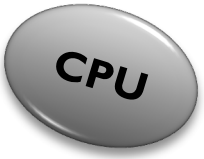


A Survey of Visibility for Walkthrough Applications,
Cohen-Or, Chrysanthou, Silva 2003

View-frustum culling

Découpage (*clipping*) des primitives intersectant le *frustum*





Utiliser une **enveloppe englobante par objet**

- Un objet est **rejeté** si son enveloppe englobante est totalement à l'**extérieur** d'un des 6 plans du *frustum*
- Complexité : nombres d'objets dans la scène

Volume englobant :

- Sphère englobante
- Boite englobante alignée par rapport aux axes
- Boite englobante orientée
- Cylindre englobant

Exemple : sphère englobante

CPU

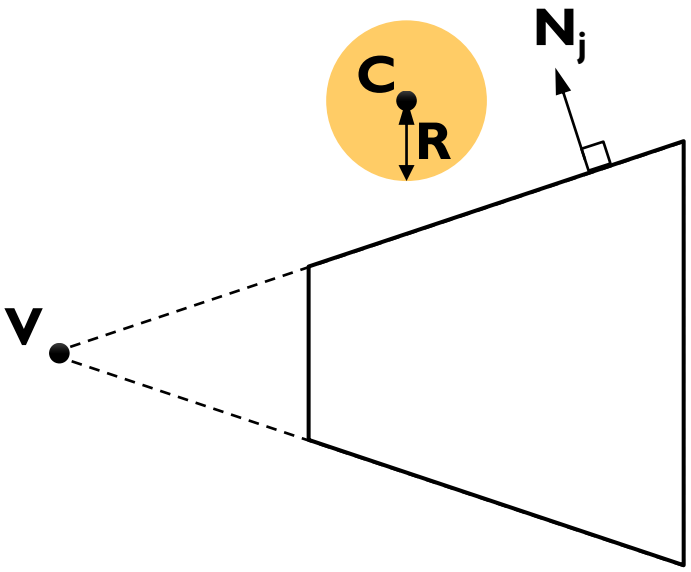
***Frustum* = 6 plans** + position de la caméra **V**

Chaque plan =

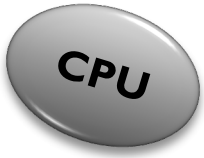
- une **normale N_j** qui pointe vers l'extérieur du volume
- un **scalaire $d_j = -N_j \bullet V$**

Test de rejet :

```
for (j=0 ; j<6 ; ++j)
{
      // distance du centre au plan
      float dist = C.dot( N[j] ) + d[j];
      if( dist > R )
            return true;
}
return false;
```



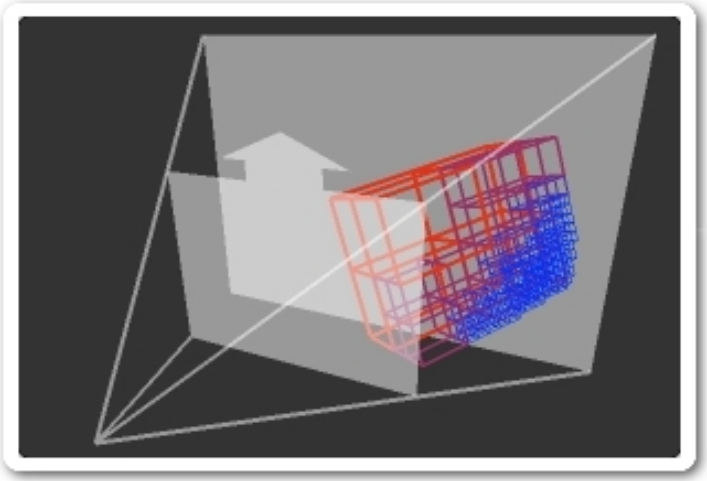
View-frustum culling avancé



Utilisation de **structures de données spatiales adaptatives** (*Octree*, *Kd-tree*, *BSP Tree*)

- Découpage de l'espace en cellules
- Chaque cellule contient la liste des objets qui l'intersectent
- Test d'intersection des cellules de la structure avec le volume de vision

Culling hiérarchique
(graphe de scène)



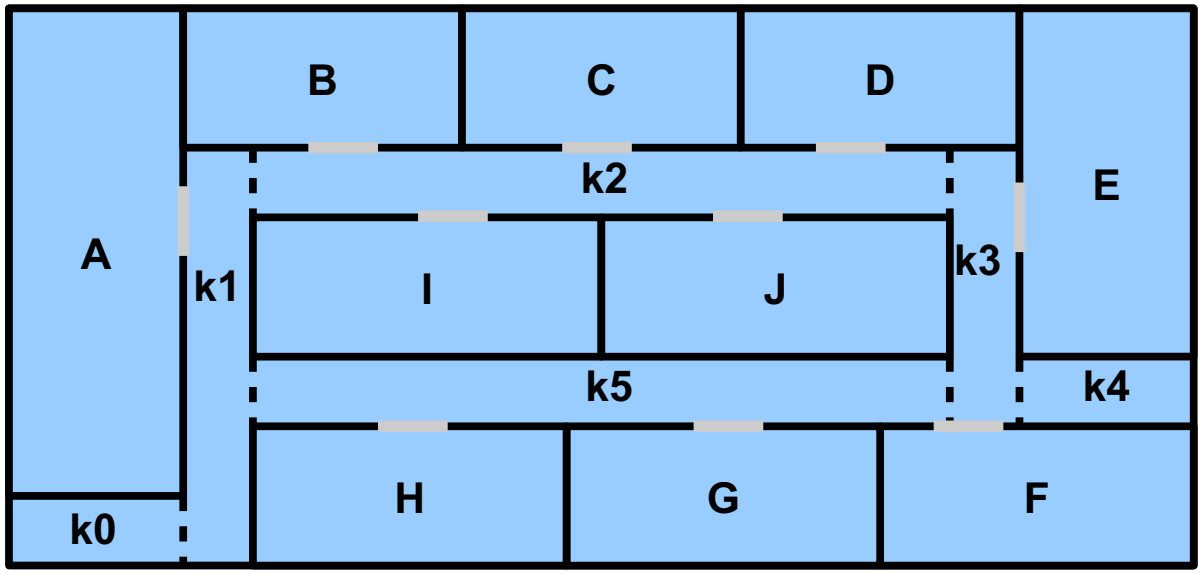
http://www.libqglviewer.com/examples/frustumCulling.html

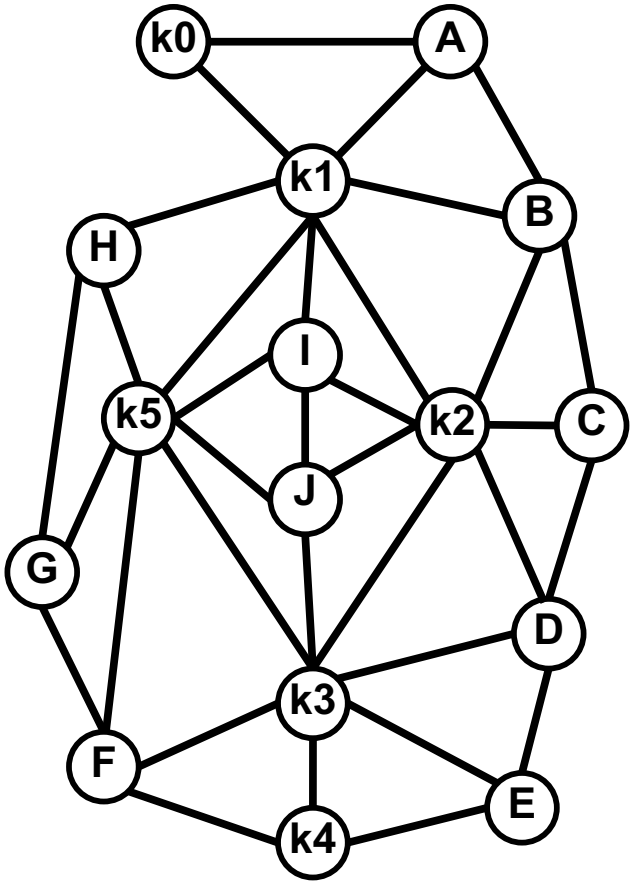
Cellules et portails

CPU

En intérieur, on ne voit jamais la totalité de la scène.

Construction du **graphe d'adjacence** des cellules.





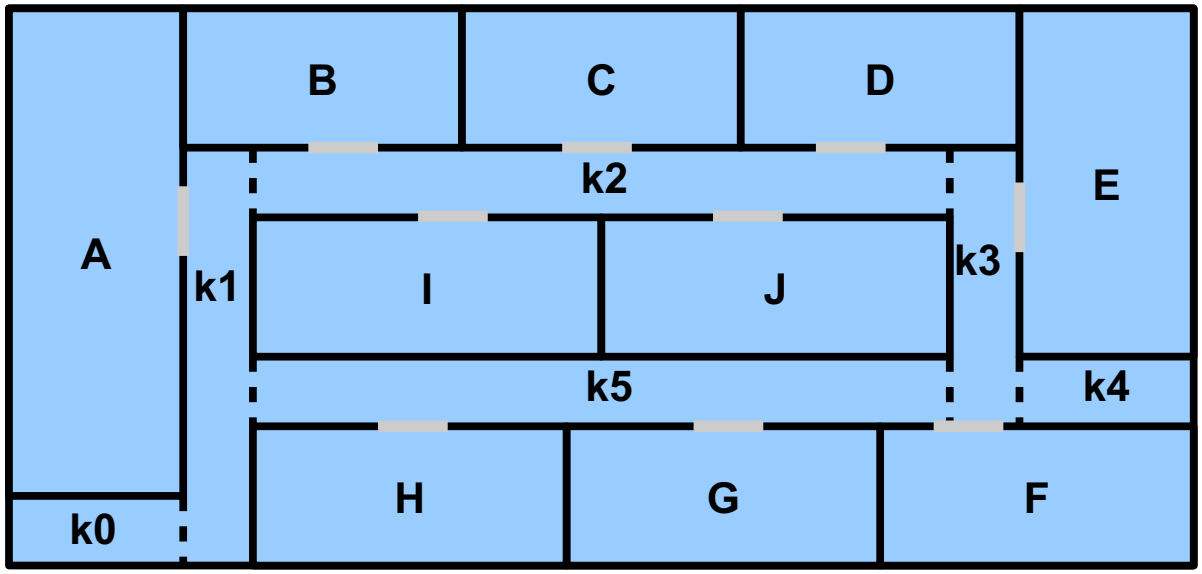
Cellules et portails

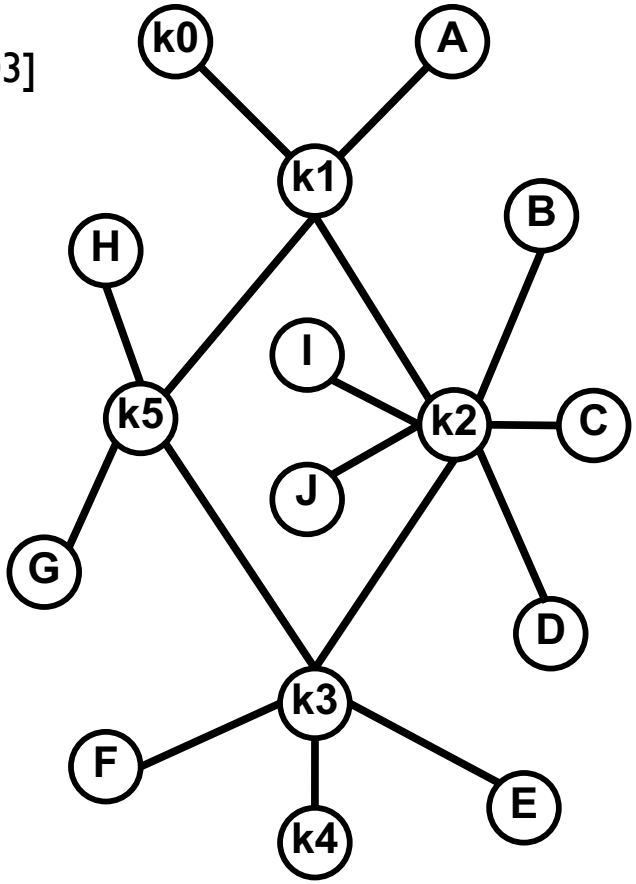
En intérieur, on ne voit jamais la totalité de la scène.

Construction du **graphe d'adjacence** des cellules.

Extension au **graphe de visibilité**

- automatiquement [Teller 1992, Haumont et al. 2003]
- en pratique, souvent manuellement



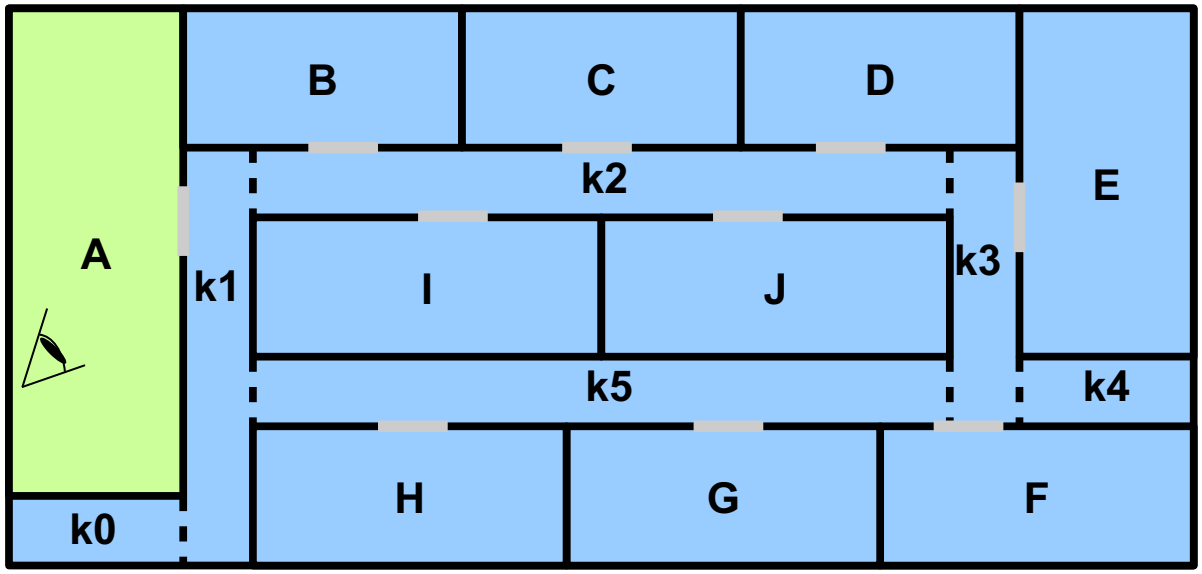


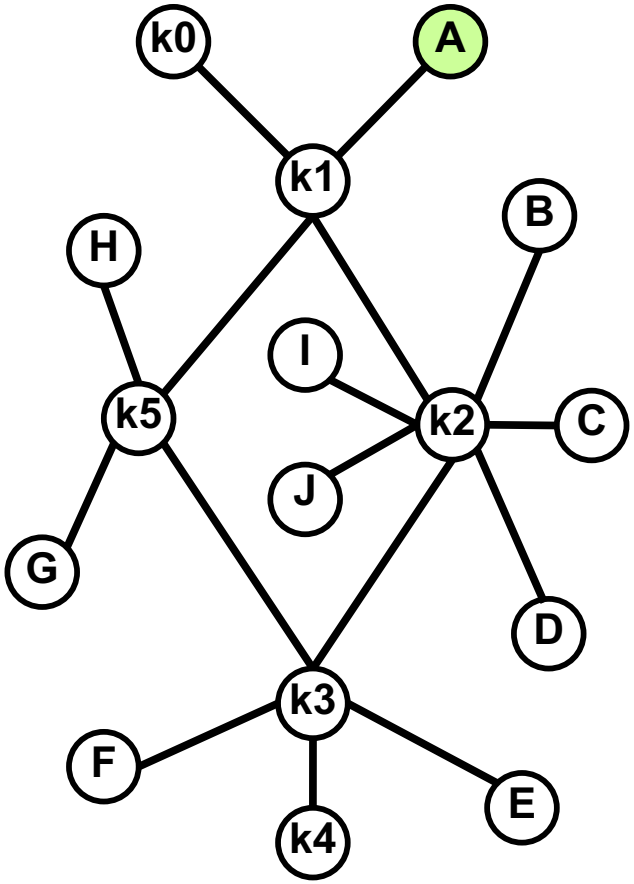
Cellules et portails

CPU

Exploitation du graphe pour le rendu

1. localisation de l'observateur



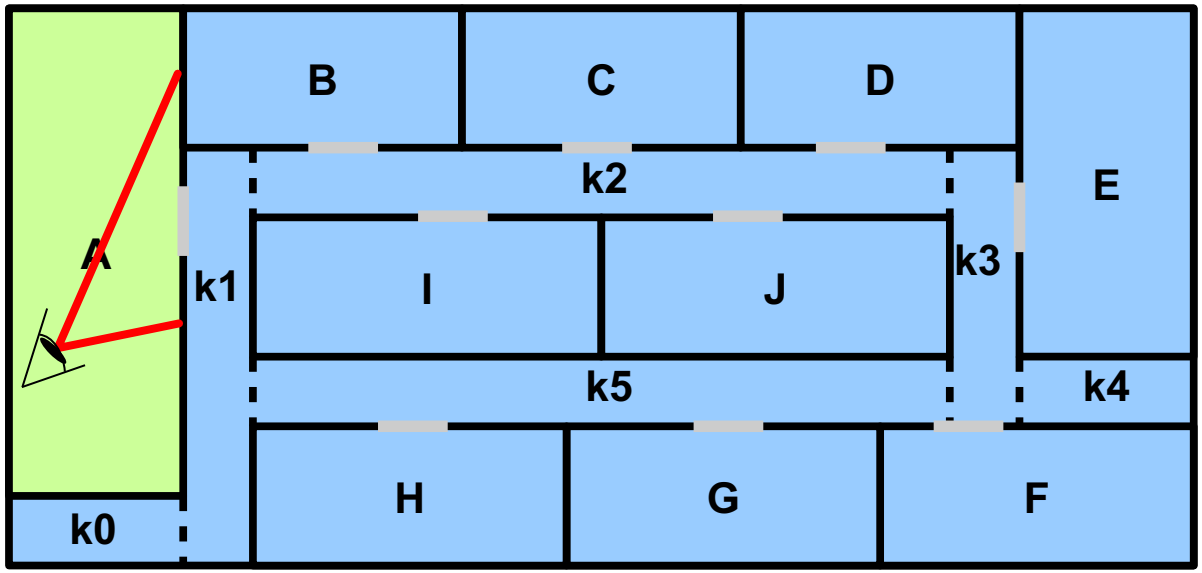


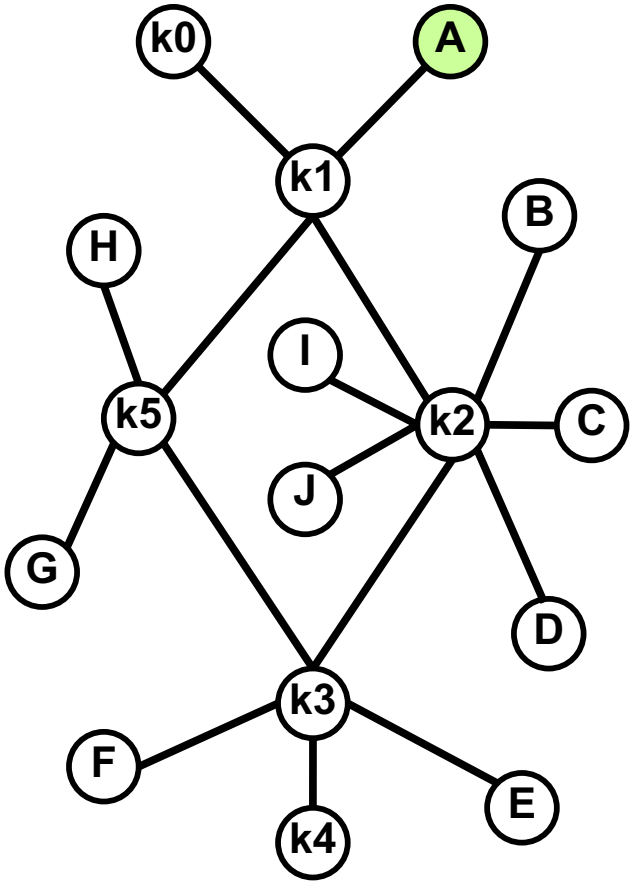
Cellules et portails

CPU + GPU

Exploitation du graphe pour le rendu

1. localisation de l'observateur
2. rendu de la cellule courante avec *view-frustum culling*



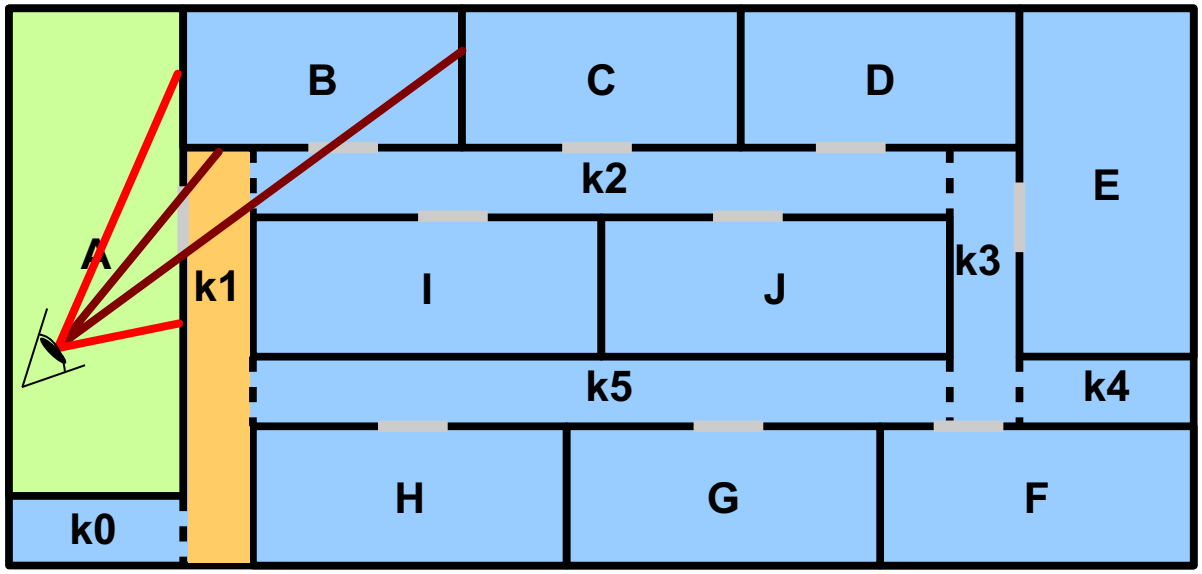


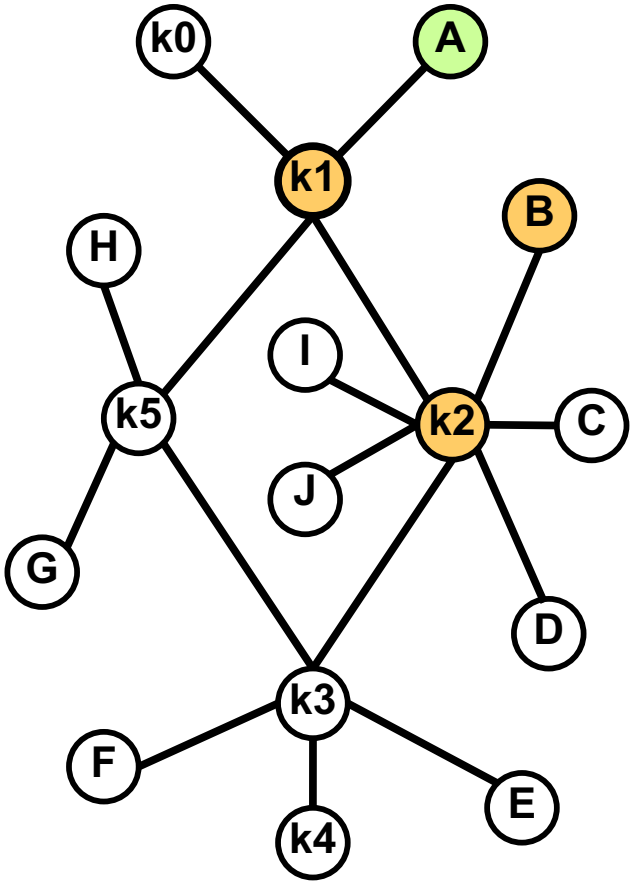
Cellules et portails

CPU + GPU

Exploitation du graphe pour le rendu

1. localisation de l'observateur
2. rendu de la cellule courante avec *view-frustum culling*
3. propagation récursive par adjacence



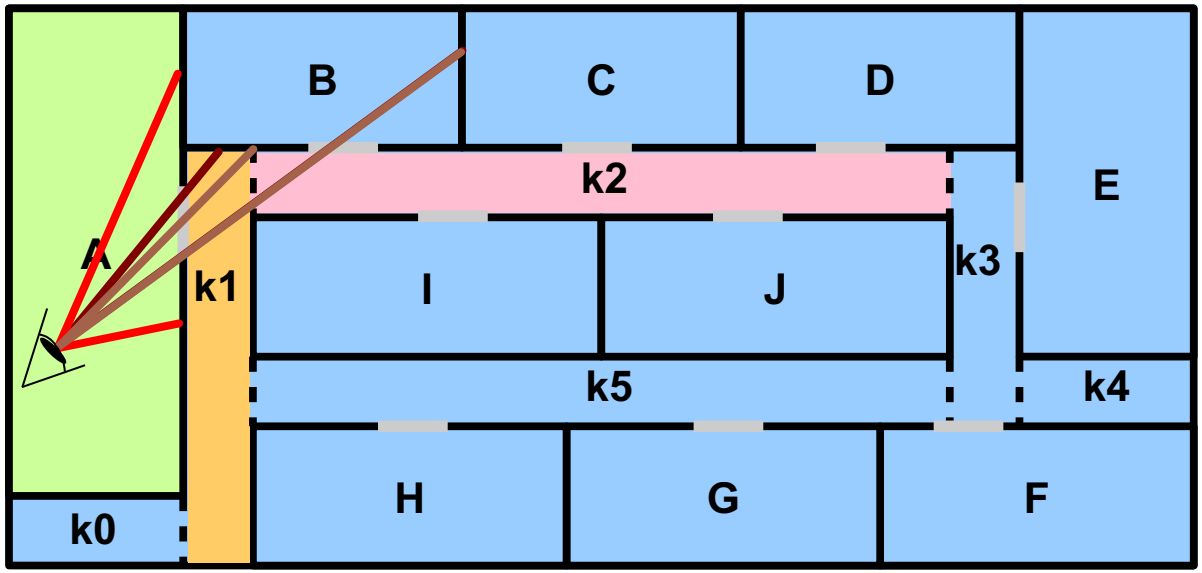


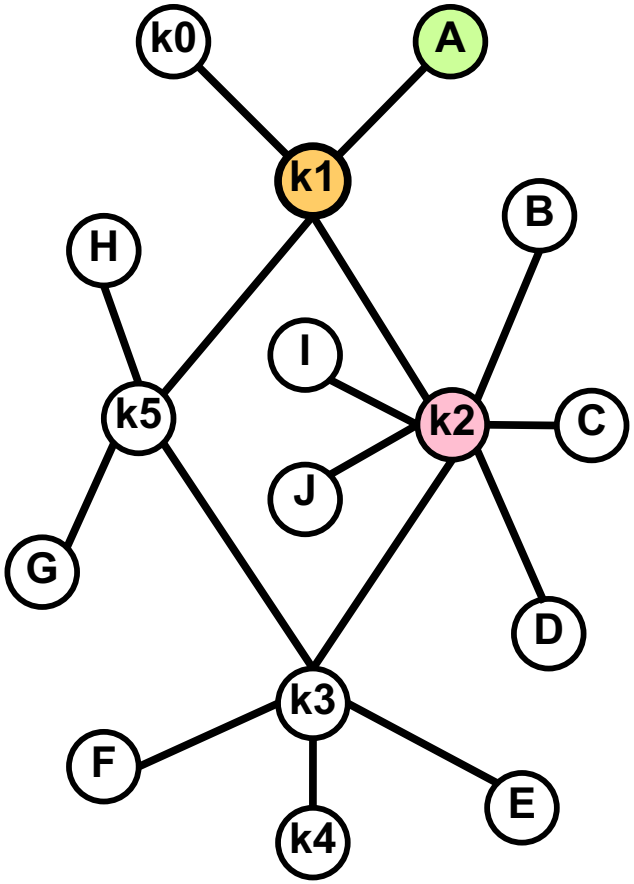
Cellules et portails

CPU + GPU

Exploitation du graphe pour le rendu

1. localisation de l'observateur
2. rendu de la cellule courante avec *view-frustum culling*
3. propagation récursive par adjacence



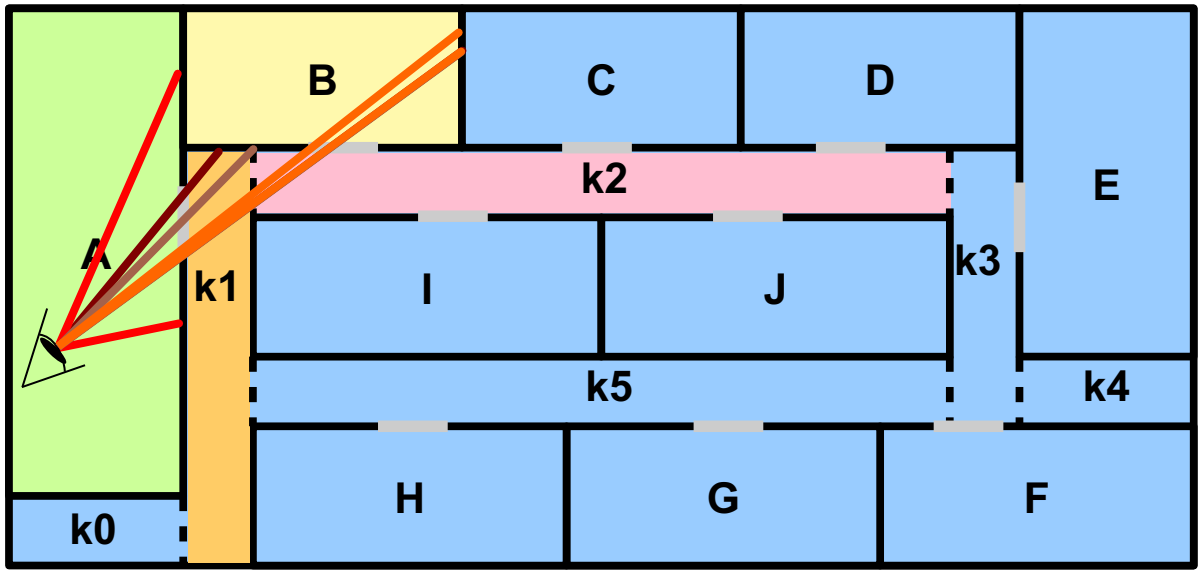


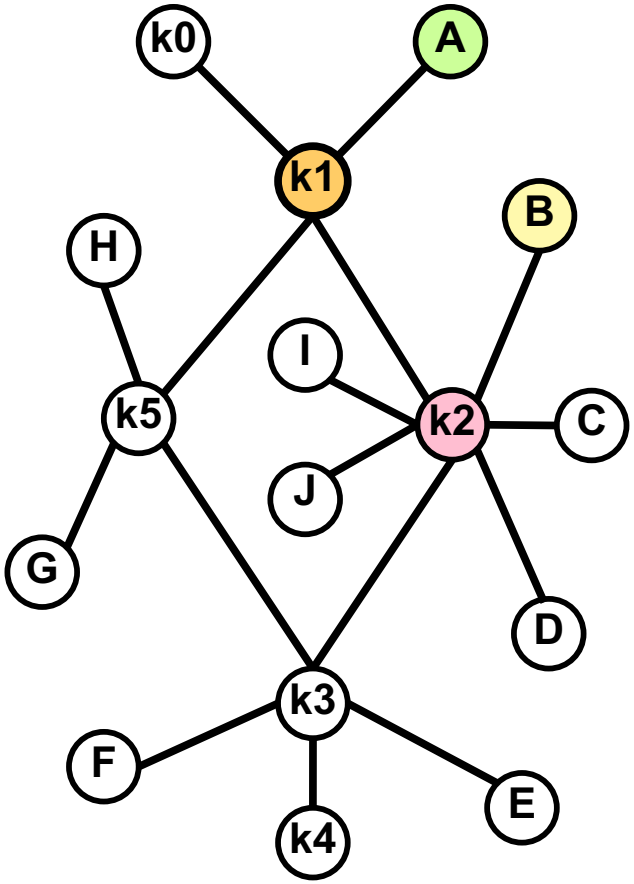
Cellules et portails

CPU + GPU

Exploitation du graphe pour le rendu

1. localisation de l'observateur
2. rendu de la cellule courante avec *view-frustum culling*
3. propagation récursive par adjacence



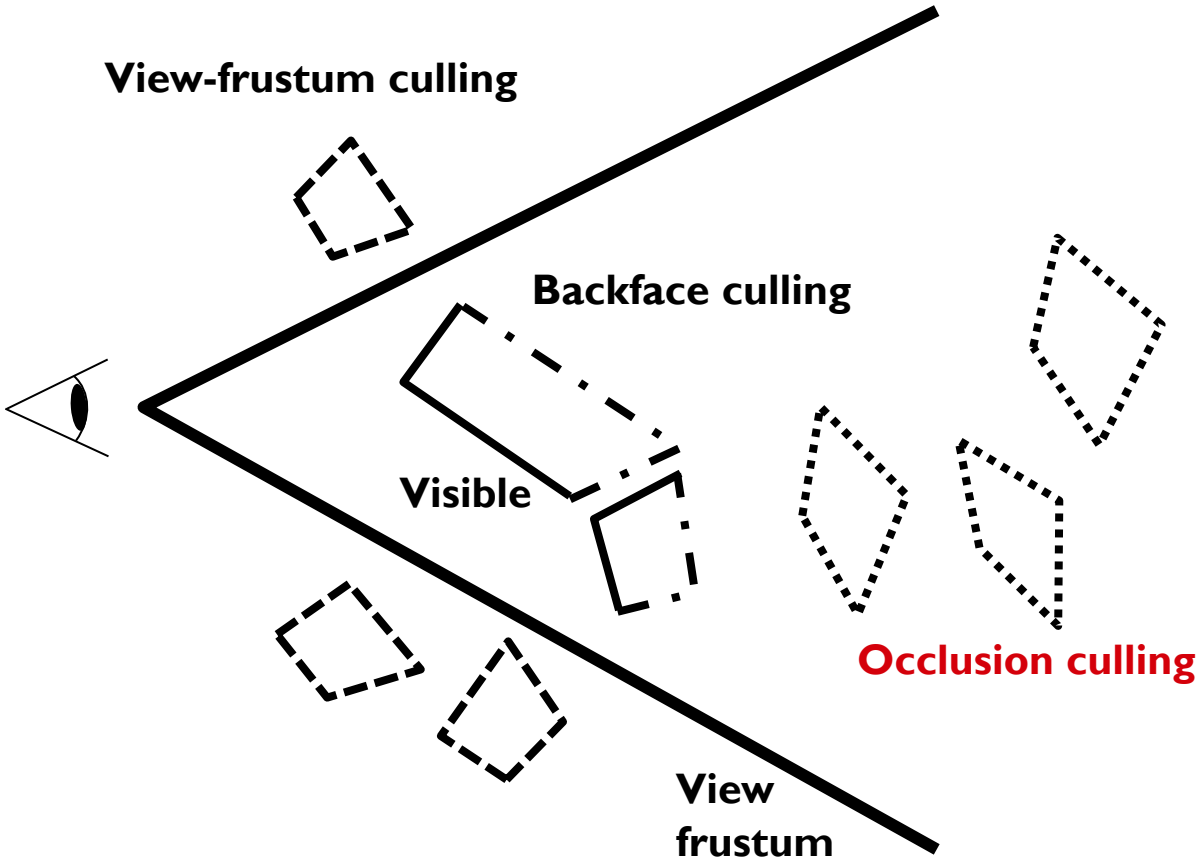


Cellules et portails

Attention aux miroirs !

David Luebke & Chris Georges, *UNC-Chapel Hill*

Culling



A Survey of Visibility for Walkthrough Applications,
Cohen-Or, Chrysanthou, Silva 2003

Occlusion culling

Répondre à la question : **« Qui cache qui ? »**

Approche globale car concerne toutes les primitives de la scène

Dans l'idéal, on voudrait l'EVS (*Exact Visible Set*)

- Ensemble des primitives (partiellement) visibles
- Vision par ordinateur : *« Aspect graph »*
- Grande complexité : $O(n^9)$

En pratique, on construit un PVS (*Potentially Visible Set*)

- Prédiction **conservative** ou **approximative** de l'EVS

Occlusion culling

Algo naïf : O_R liste des *occluders* potentiels

Pour chaque objet g de la scène :

1. Tester si la boite englobante de g est occultée selon O_R
   $\Longrightarrow$ si « vrai », ignorer g
2. Si « faux », tracer g
3. Ajouter g à la liste O_R des *occluders* potentiels

Ordre de parcours des objets ?

Occlusion culling

Algo naïf : O_R liste des *occluders* potentiels

Pour chaque objet g de la scène :

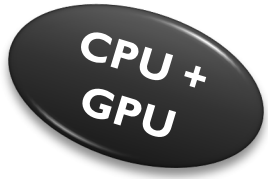
1. Tester si la boite englobante de g est occultée selon O_R
   $\Longrightarrow$ si « vrai », ignorer g
2. Si « faux », tracer g
3. Ajouter g à la liste O_R des *occluders* potentiels

Ordre de parcours des objets ?

$\Longrightarrow$ **beaucoup plus efficace d'avant vers arrière**

$\Longrightarrow$ **tri grossier**

Occlusion culling



Nom de code OpenGL : *occlusion queries*

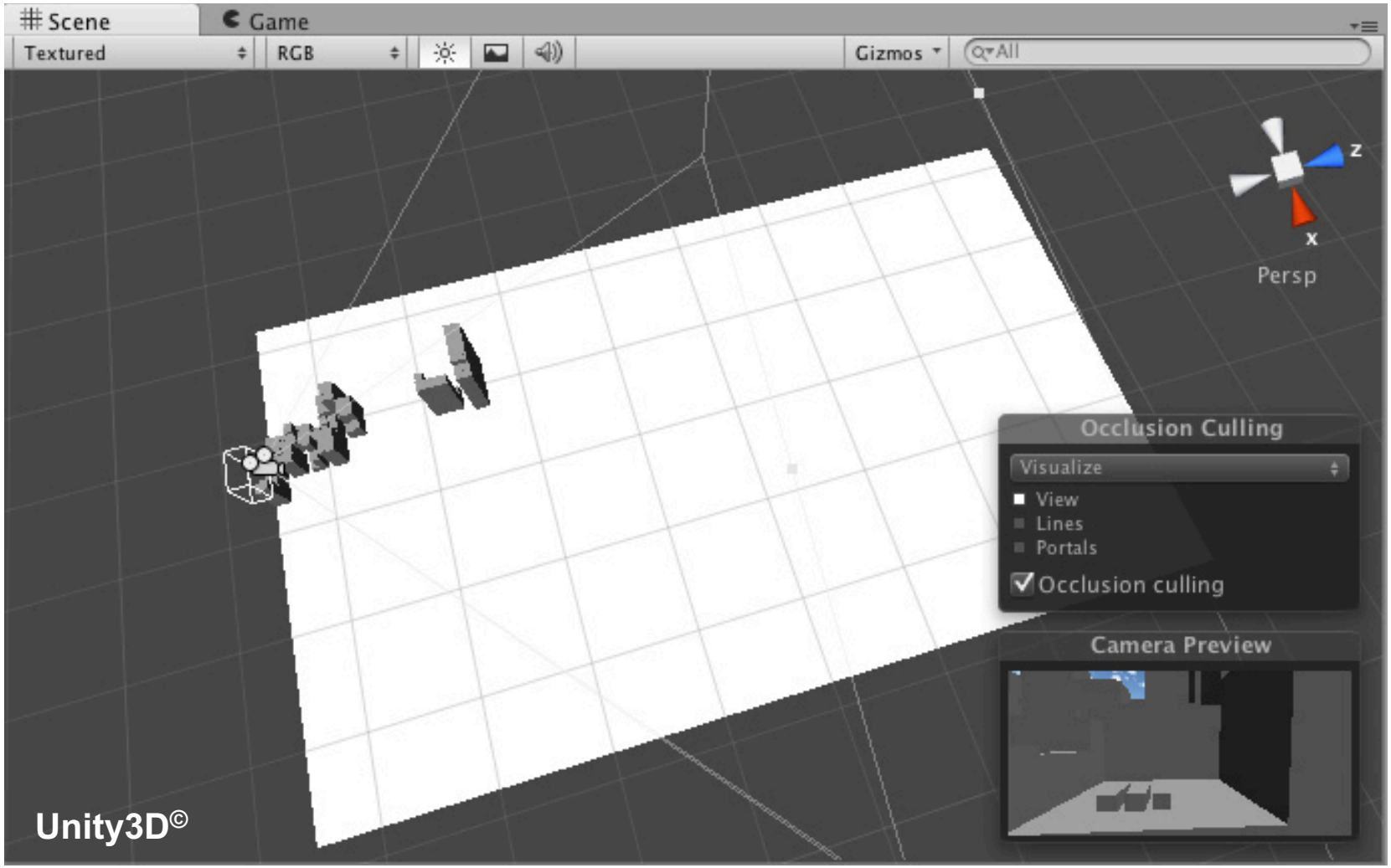
- Requête = tracé d'un groupe de primitives (ou de leur volume englobant) sans modifier les buffers de destinations
- Retourne « vrai » si **au moins un** test de profondeur a réussi ⟹ le groupe de primitives **n'est pas complètement caché**
- Algorithme en espace image :
  - rastérisation du volume englobant
  - test avec le *depth buffer* courant (sans le modifier)
  - comptage du nombre *n* de fragments visibles
  - si *n* = 0, primitives invisibles

http://www.opengl.org/wiki/Query_Object

Occlusion culling

CPU + GPU

Efficace pour les scènes complexes avec beaucoup d'occlusions

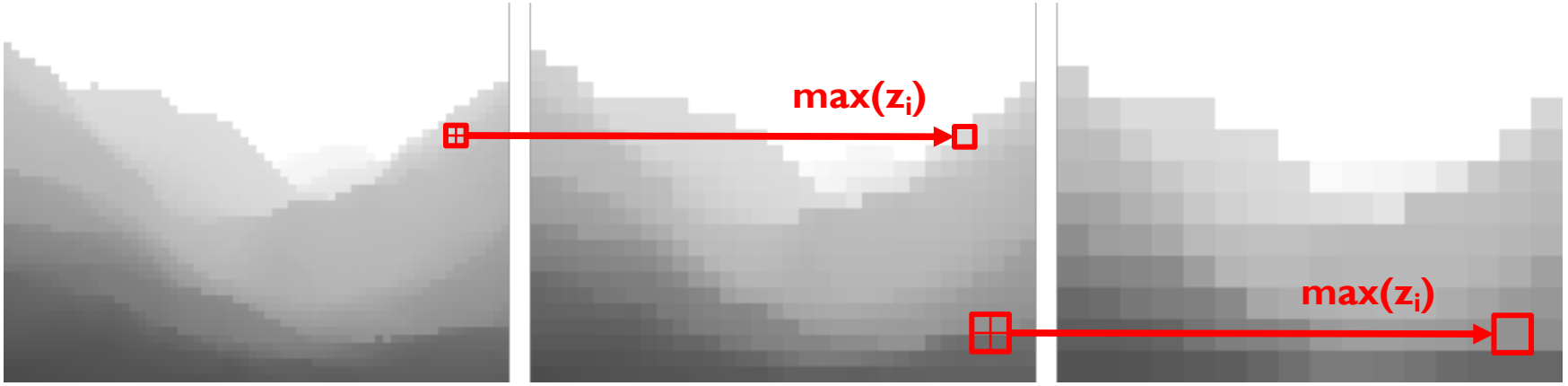


Hierarchical-Z (Hi-Z) culling

GPU

Pyramide / MIP-map de *depth buffers* [Greene et al. 93]

1. Rendu des *occluders* pour remplir le *depth buffer*
2. Création de niveaux de MIP-map en conservant le z_{max} par texel (FBO + Fragment Shader)



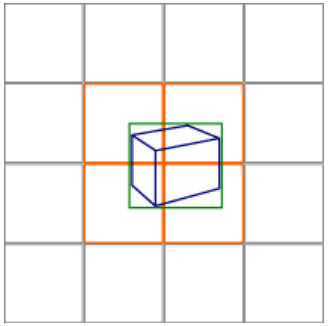
http://rastergrid.com/blog/2010/10/hierarchical-z-map-based-occlusion-culling/

Hierarchical-Z (Hi-Z) culling

GPU

Pyramide / MIP-map de *depth buffers* [Greene et al. 93]

1. Rendu des *occluders* pour remplir le *depth buffer*
2. Création de niveaux de MIP-map en conservant le z_{max} par texel (FBO + Fragment Shader)
3. Test de visibilité de primitives
   a. Sélection du niveau de MIP-map en fonction de la taille de l'AABB projetée (recouvrant 2x2 texels)
   b. Estimation du z_{min} de la primitive, comparaison au z_{max} $\Longrightarrow$ **rejet conservatif**

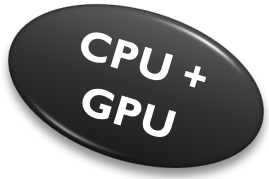


- *world-space AABB*
- *screen extents*
- *overlapping HZB texels*

http://blog.selfshadow.com/publications/practical-visibility/

https://arm-software.github.io/opengl-es-sdk-for-android/occlusion_culling.html

Occlusion culling



Identification des objets « occultants »

- Objets ayant une très forte probabilité d'être visibles a priori
  - les objets les plus proches du point de vue
  - les objets visibles à l'image précédente
    $\Longrightarrow$ cohérence spatio-temporelle
    un objet étiqueté « visible » le sera pendant quelques images
- Ces objets sont tracés sans test d'occlusion