Compute shaders

IMN504

Nuanceurs de calcul

Étape indépendante du pipeline graphique

- Disponible presque partout
- Shader stand-alone : GL_COMPUTE_SHADER

General Purpose GPU (GPGPU)

- Utiliser les capacités de calcul parallèle des GPUs
- Manipulation d'images, physique, visibilité, post-process
- Apprentissage machine

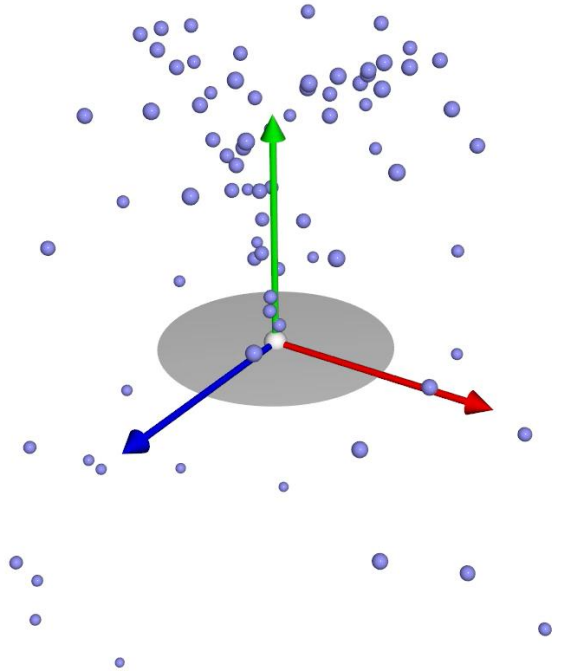
Avantages

- Les données restent sur le GPU

Cas exemple : Système de particules

Rappel : Fontaine

- Système simple
- Gravité
- Intégration semi-implicite
  - Calculer les forces
  - Mettre a jour l'accélération
  - Mettre a jour la vitesse
  - Mettre a jour la position



Exemple sur CPU

Simulation de particules

- Un ensemble de positions (sommets)
- Une méthode de mise à jour

Sur CPU

- Mettre à jour le tableau de positions (CPU)
- Envoyer les données sur le GPU
- Afficher les particules

Problème d'efficacité

- Transfert des données
- Vitesse de traitement sur le CPU

Exemple sur GPU

Simulation de particules

- Un ensemble de positions (sommets)
- Une méthode de mise à jour

En Compute Shader

- Mettre à jour le tableau de positions (GPU)
- ~~Envoyer les données sur le GPU~~
- Afficher les particules

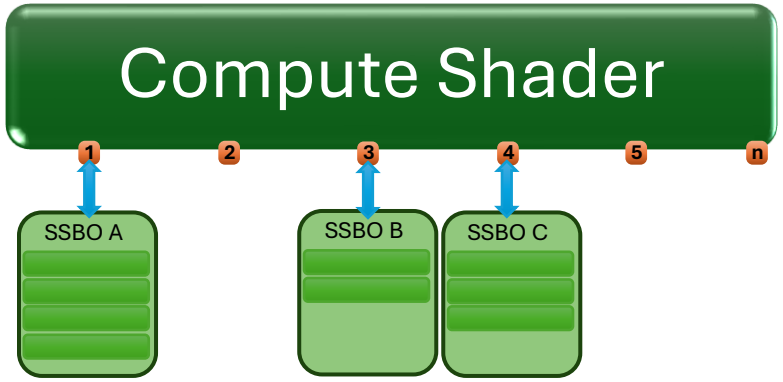
Lire et écrire des données sur le GPU

- Besoin de structures de données

Mémoire GPU

Shader Storage Buffer Object

- Tableau de données
- Types simples
- Types complexes (*struct*)
- Efficace en lecture/écriture
- Type de buffer : GL_SHADER_STORAGE_BUFFER
- Attaché à des canaux spécifiques (*layouts* et *bindings*)



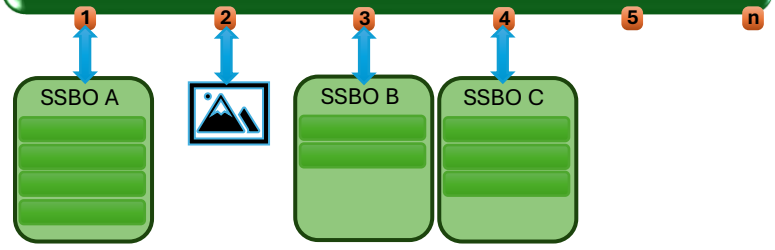

```
layout( std140, binding=2 ) buffer Pos
{
    vec4 Position[250];
    vec4 Velocity[ ];
};
```

Mémoire GPU

Image*D

- Différent de sampler*d
  - Utiliser glBindImageTexture()
- Lecture/écriture (dans le shader)
  - *imageLoad(), imageStore()*
- Pas de *mipmap*, d'échantillonnage
- Un simple tableau *D de données
- Mécanisme de *layouts*

Compute Shader

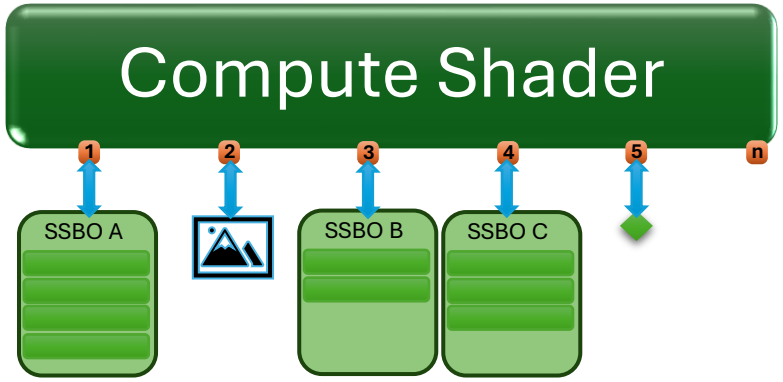



```
layout( r32f, binding = 2 ) uniform writeonly
image3D myImage;

void main()
{
  imageStore( myImage…)
}
```

Mémoire GPU

Compteurs atomiques

- Stockage de compteur 32 bits
- Lecture/écriture atomique
- Offsets automatiques
- Des fonctions complexes (add, and, max…)
  - En OpenGL 4.6

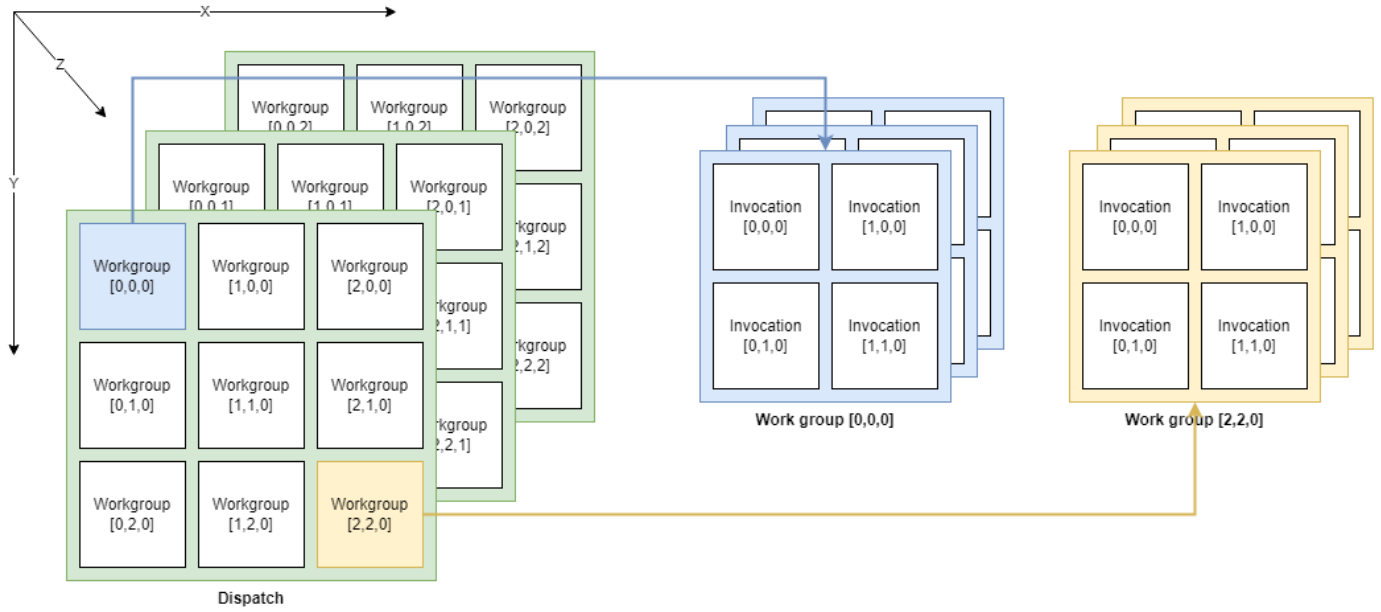



```
layout(binding = 5) uniform atomic_uint mC;
layout(binding = 5) uniform atomic_uint mC2;

void main()
{
 uint c = atomicCounter(mC);
 uint c2 = atomicCounterIncrement(mC2);

}
```

Découpage du travail

Workgroup

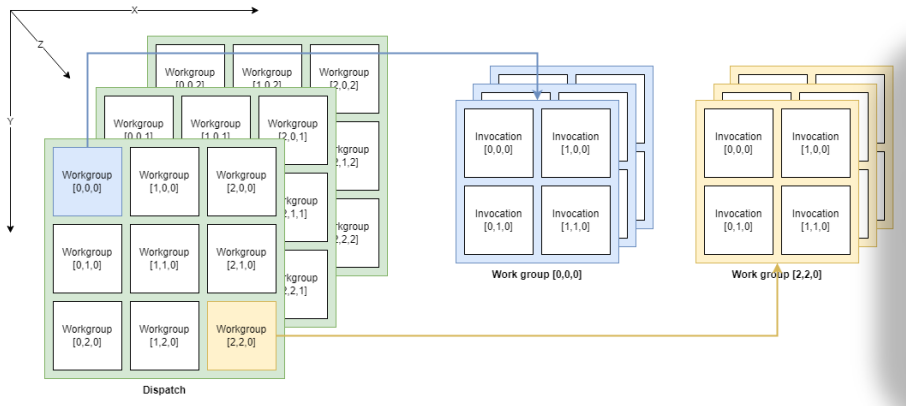
- Défini par l'appel depuis le CPU (en 3D)
- En OpenGL : glDispatchCompute(numgroupX, numgroupY, numgroupZ)
- Représente un groupe d'invocations du shader
- Un nombre maximal d'invocations au total : GL_MAX_COMPUTE_WORK_GROUP_COUNT



Découpage du travail

Invocations

- Défini par le shader (en 3D) : layout(local_size_x = X, local_size_Y =…)
- Chaque invocation exécute un shader
- Exécution en parallèle des invocations
- Affecte le nombre maximal d'invocations

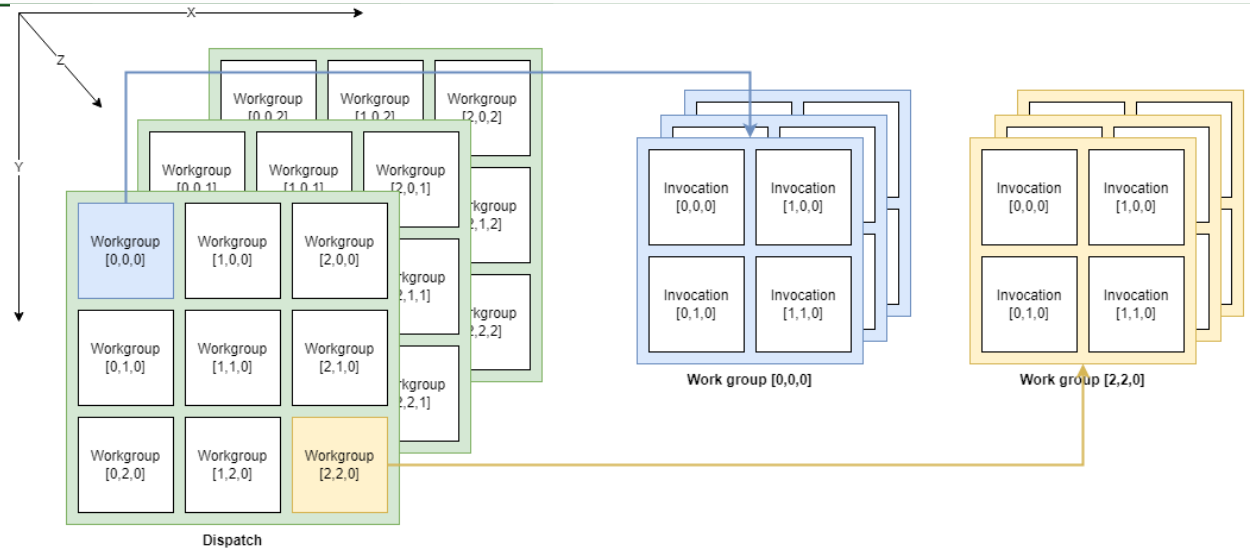



```
layout( local_size_x = 8, local_size_y = 8,
local_size_z = 8 ) in;

void main()
{


}
```

Identifier les invocations

Un ensemble de variables uniques

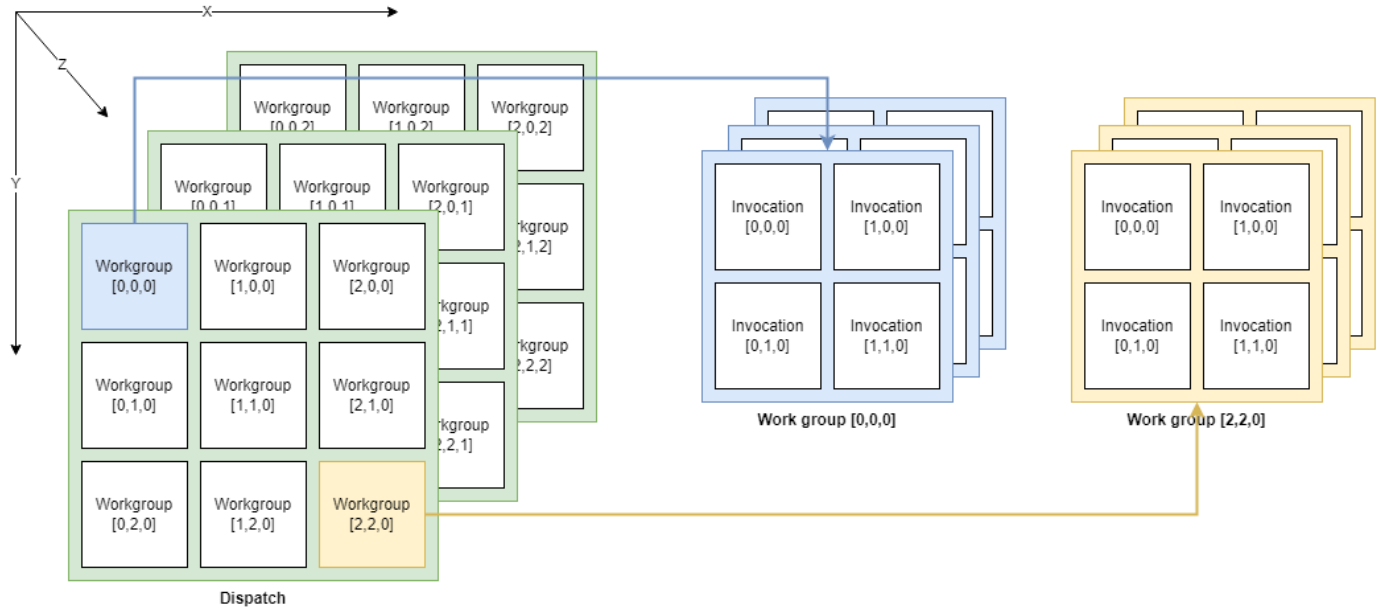
- **in** uvec3 gl_NumWorkGroups; — Nombre de workgroups
- **in** uvec3 gl_WorkGroupID; — Numéro(XYZ) du workgroup
- **in** uvec3 gl_LocalInvocationID; — Numéro(XYZ) local de l'invocation dans le workgroup
- **in** uvec3 gl_GlobalInvocationID; — Numéro(XYZ) global de l'invocation
- **in** uint gl_LocalInvocationIndex; — Numéro(XYZ) local de l'invocation



Découpage du travail

Principe de base

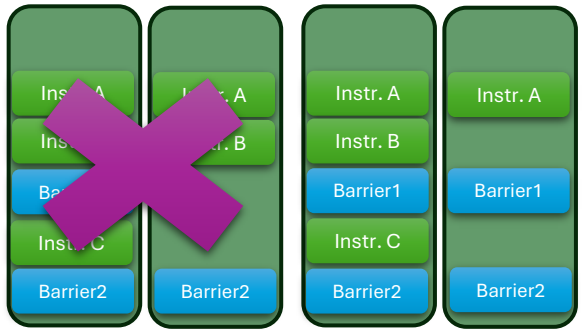
- Découper le travail en workgroup et invocation
- Pas de mémoire partagée entre les workgroup
- Des accès mémoires partagés entre les **invocations**



Découpage du travail

Synchronisation des invocations

- Des variables partagées : *shared*
- Une **seule** invocation doit l'initialiser
- Besoin de synchroniser les exécutions : barrier()
  - Stoppe l'exécution jusqu'à ce que **toutes** les invocations passent la barrière
  - L'ordre des barrières doit être identique
  - Rend visible les variables partagées (après)
- Besoin de synchroniser la mémoire : memoryBarrierShared()
  - S'assure que **toutes** les écritures sont effectuées




```
shared float var;

void main()
{
    A();
    B();
    barrier();
    C();
    barrier();
}
```

Retour aux particules

Simulation de particules

- Un ensemble de positions (sommets)
- Une méthode de mise à jour

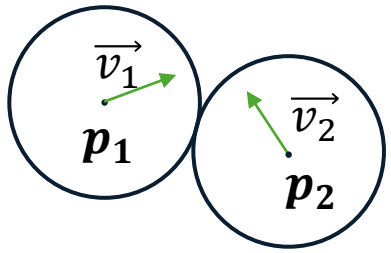
Méthode

- Mettre à jour le tableau de positions (CS)
- Afficher les particules

Questions de synchronisation

Problématique des accès parallèles

- Chaque instance traite et affecte une unique particule
  - Test des collisions
  - Réponse
    - Mise a jour position, vitesse



Deux particules (p_1, v_1) et (p_2, v_2)

(p_1, v_1)

- Tester collision avec (p_2, v_2)
  - Collision
- Calculer (p'_1, v'_1)
- $(p_1, v_1) = (p'_1, v'_1)$

Questions de synchronisation

Problématique des accès parallèles

- Chaque instance traite et affecte une unique particule
  - Test des collisions
  - Réponse
    - Mise a jour position, vitesse

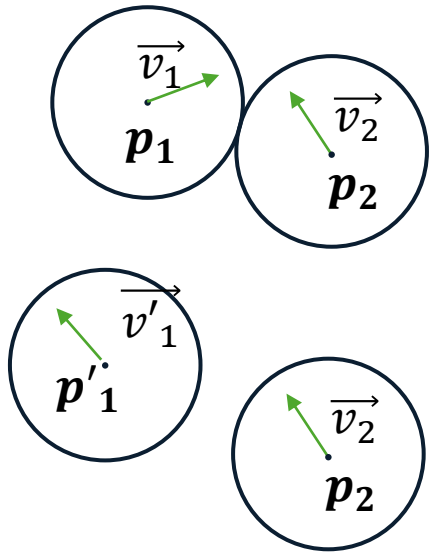
Deux particules (p_1, v_1) et (p_2, v_2)

(p_1, v_1)

- Tester collision avec (p_2, v_2)
  - Collision
- Calculer (p'_1, v'_1)
- $(p_1, v_1) = (p'_1, v'_1)$

(p_2, v_2)

- Tester collision avec (p_1, v_1)
- Pas de collision
- Calculer (p'_2, v'_2)
- $(p_2, v_2) = (p'_2, v'_2)$



Questions de synchronisation

Problématique des accès parallèles

- Chaque instance traite et affecte une unique particule
  - Test des collisions
  - Réponse
    - Mise a jour position, vitesse

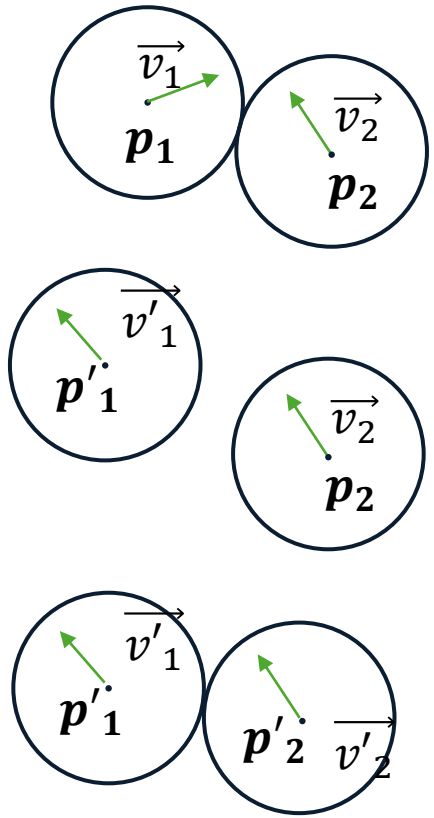
Deux particules (p_1, v_1) et (p_2, v_2)

(p_1, v_1)

- Tester collision avec (p_2, v_2)
  - Collision
- Calculer (p'_1, v'_1)
- $(p_1, v_1) = (p'_1, v'_1)$

(p_2, v_2)

- Tester collision avec (p_1, v_1)
- Pas de collision
- Calculer (p'_2, v'_2)
- $(p_2, v_2) = (p'_2, v'_2)$



Questions de synchronisation

Raisonnement au niveau des collisions

- Pour une collision entre deux particules
  - Modifier (p_1, v_1) et (p_2, v_2)
  - Comment synchroniser ?
  - Dans ce cas :
    - Une instance du shader = une collision
    - Raisonner au niveau des collisions
      - Combien ? Comment les détecter ?
    - Que faire en cas de multiples collisions
      - En parallèle ?

Synchroniser les étapes temporelles

- Ordre d'évaluation des particules inconnu
- Un tableau d'entrée et un tableau de sortie
- Mise a jour de l'entrée synchronisée
- En pratique, échange des tableaux
  - Entrée(n+1)=Sortie(n)

Fonctionnement général

Initialisation

- Préparer les buffers (SSBO) (entrée et sortie)
- Remplir les conditions initiales

Boucle de rendu

- Exécuter le compute shader
  - Mise a jour des variables
- Mettre une barrière pour s'assurer que le CS a fini
  - glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT)
    - S'assure que les opérations d'écriture sont terminées
- Effectuer le rendu
  - Comment ?

Rendu a partir du CS

Besoin d'instanciation

- Un unique modèle géométrique de sphère
- Lancer le rendu de **N** instances (N fixe ET connu à l'avance)
  - `glDrawElementsInstanced(…,N,…)`
- Aller chercher les positions dans le SSBO
  - Utiliser gl_InstanceID

Dans le shader

- On a en entrée :
  - La position d'un sommet de la géométrie
  - Une matrice Model de l'objet global *Particules*
  - Un numéro d'instance
    - Identique pour tous les sommets de la même instance
- Effectuer une translation pour chaque sommet
  - De la valeur du centre de l'instance (position de la particule)
  - A récupérer dans le tableau des positions
    - Avec le numéro d'instance

Optimiser les calculs

Complexité

- Pour chaque particule
  - Tester les collisions entre n particules
- Complexité totale de $\theta(n^2)$
- Besoin de structures accélératrices

Structures accélératrices

Subdivisions de l'espace

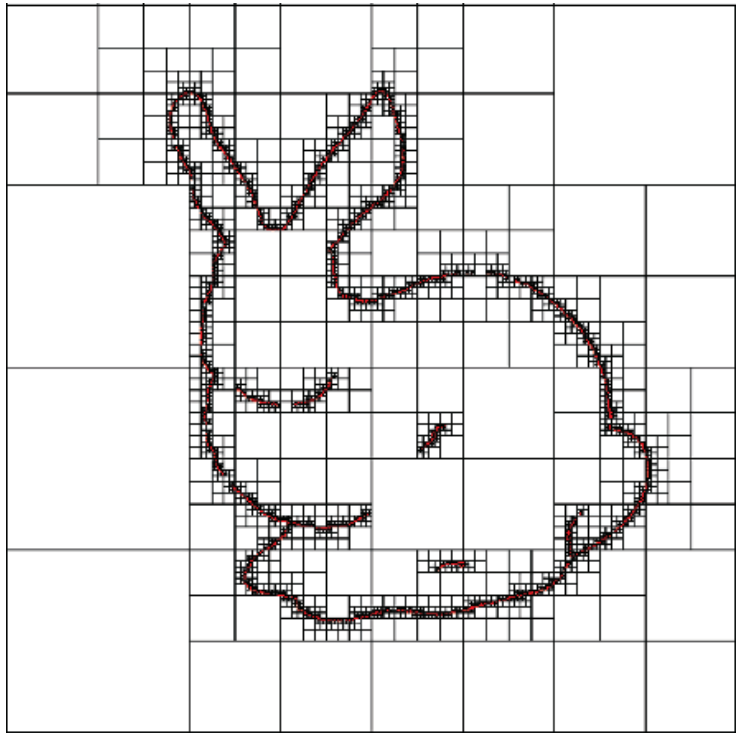
Structures régulières

QuadTree,OcTree

- Subdiviser l'espace
- Division régulière récursive
- Subdivision selon un critère
  - Ex : Cellule non-vide
  - Ex : Profondeur max k

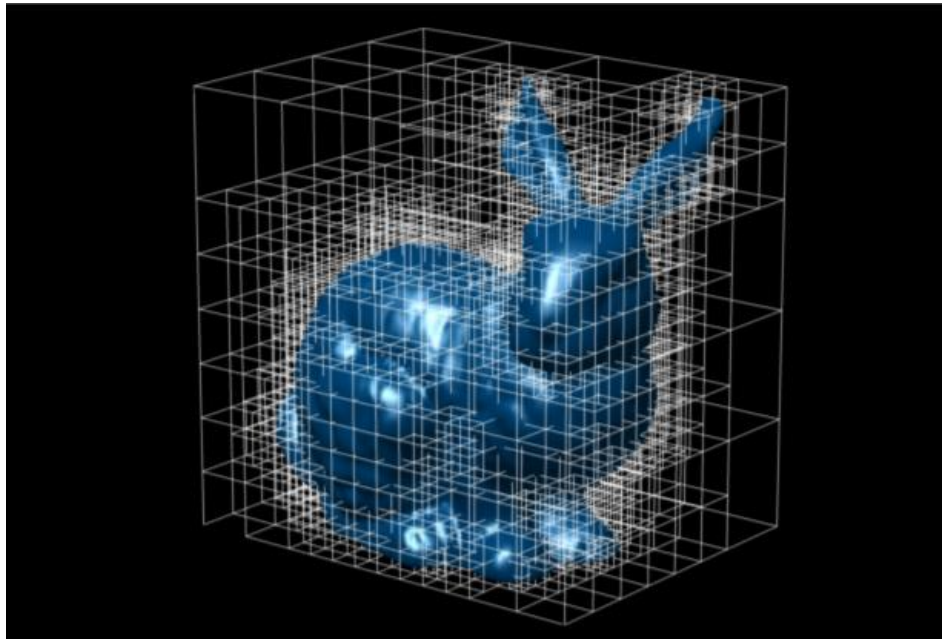
Pseudo-algorithme QuadTree

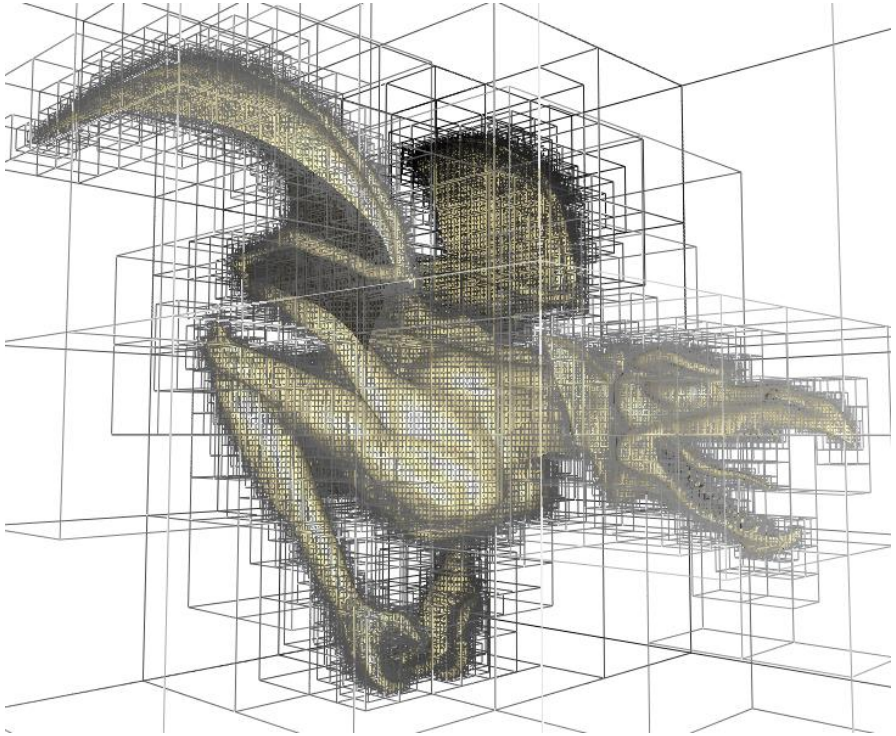
- Tant que ma liste L est non vide
  - Prendre une cellule C^i
  - Si (non-vide(C^i) et $i < k$)
    - Subdiviser C^i en 4 cellules C^{i+1}
    - Ajouter tous les C^{i+1} a L



Structures accélératrices

- Subdivisions de l’espace
  - Octree





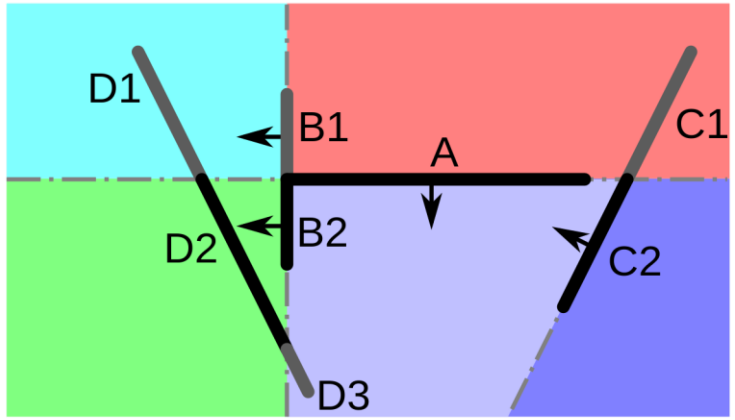
Structures accélératrices

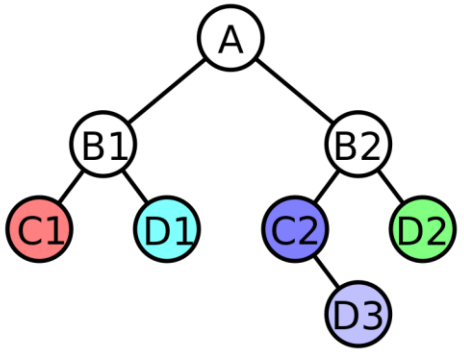
Subdivisions de l'espace

Binary Space Partition

BSP

- Subdiviser l'espace
- Diviser l'espace en deux
  - Avant/Arrière
  - Critère d'arrêt
- Arbre BSP (Doom,Quake…)
- Algorithme récursif





Structures accélératrices

Indexer facilement

- Table de hachage
- Retrouver une cellule facilement avec les positions XYZ
  - $c = hash(pos)$
- Un tableau **C** de cellules
  - Chaque cellule **c** contient une liste de particules

Structures accélératrices

Indexer facilement

- Table de hachage
- Retrouver une cellule facilement avec les coordonnées XYZ
- Pour une particule **p**
  - Retrouver la cellule **c**
  - Pour chaque particule **p'** dans **c**
    - Tester la collision
    - Éventuellement les particules des cellules voisines
  - Mettre a jour position/vitesse
    - $p_{new} = collision(p, p')$

Structures accélératrices

Mettre a jour la table **C**

- Problème : p_{new} n'est peut être plus dans **c**
  - $p_{new} \notin c$
- Il faut supprimer p dans **c** et insérer p_{new} au bon endroit
- Quand faire cette opération ?

Structures accélératrices

Mettre a jour la table **C**

- Dans le compute shader traitant la particule p ?
  - Si p n'est plus dans c, il ne peut plus entrer en collision avec les autres p' de c
  - Problème de parallélisme
- Il faut synchroniser au bon moment
  - Un autre compute shader
    - Une passe de mise a jour de la table **C**

Structures accélératrices

Table de hachage en parallèle

- Problématique
  - Plusieurs cellules peuvent insérer au même moment au même endroit
  - Synchroniser les accès
  - Utiliser les opérations atomiques
  - Mise à jour paresseuse
- Beaucoup de littérature
- A faire en exercice