



VULKAN™ pour le calcul scientifique

Xavier JUVIGNY

ONERA

THE FRENCH AEROSPACE LAB

Plan de l'exposé

Présentation générale

Présentation de l'API Vulkan

Les shaders

L'API côté hôte

API utilisant Vulkan

Conclusion

Plan de l'exposé

Présentation générale

Présentation de l'API Vulkan

Les shaders

L'API côté hôte

API utilisant Vulkan

Conclusion

Historique

1992 OpenGL (Silicon Graphics)



1995 DirectX (Microsoft)



2016 Vulkan (Khronos Group)



Vulkan en quelques points

- Norme gérée par le consortium industriel Khronos Group (OpenCL, OpenGL, Sycl, slang, WebGL, etc.)
- Destiné à remplacer OpenGL et OpenGL/ES pour le rendu graphique
- Supporté par cartes graphiques récentes (**> 2016**)
- Exploite architectures modernes des GPGPUs
- Utilisation multi-cœurs CPUS pour création images
- Supporté par principaux OS (**Windows, Linux, iOS, android**)
- API C/C++ proche du hardware des GPGPUs
- Dernière version 1.3.296 (**8 Octobre 2024**)

Objectifs et réalisation

Objectifs

- Se débarrasser des limitations d'OpenGL
- Pouvoir exploiter la puissance totale de la machine

Réalisation

- Pas d'automate d'état global
 - permet multithreading sur CPUs
- Synchronisation CPU–GPU gérée par développeur
 - Optimisation pilote ignorant concurrences mémoires
- Mémoire entièrement gérée par développeur
 - Optimisation ressource mémoire GPGPU possible
- Vérification erreurs minimale
 - Pas de vérifications dans pilote (optimisation)

Langages proposant une API Vulkan



- Proposés en natif (APIs proches)
- C++ plus sûr et concis que C
- **Attention** : la plupart des tutoriaux en C et non en C++



- Plusieurs apis existent mais la plus utiliser : **vulkan** de **realitix**
- Mais pas de mise à jour depuis plus de sept mois



- **vulkano-rs** : API de référence
- Mises à jours régulières

Ecosystème Vulkan

- Le kit de développement officiel, gratuit, développé par LunarG propose de nombreux utilitaires ;
- Vulkan Configurator (license apache) permet d'outrepasser les composants optionnels définis lors de la création d'une instance dans Vulkan ;
- Permet de rajouter une couche de validation (avec divers degrés de validation), de trace d'appel, de capture de frame, de diagnostique de crash, etc.
- De voir l'environnement Vulkan installé sur la plateforme ;
- De voir les caractéristiques du GPU gérés par Vulkan ainsi que les extensions associées à ce GPU.
- Pour le débogage, voir RenderDoc qui permet de déboguer des programmes Vulkan.

Plan de l'exposé

Présentation générale

Présentation de l'API Vulkan

Les shaders

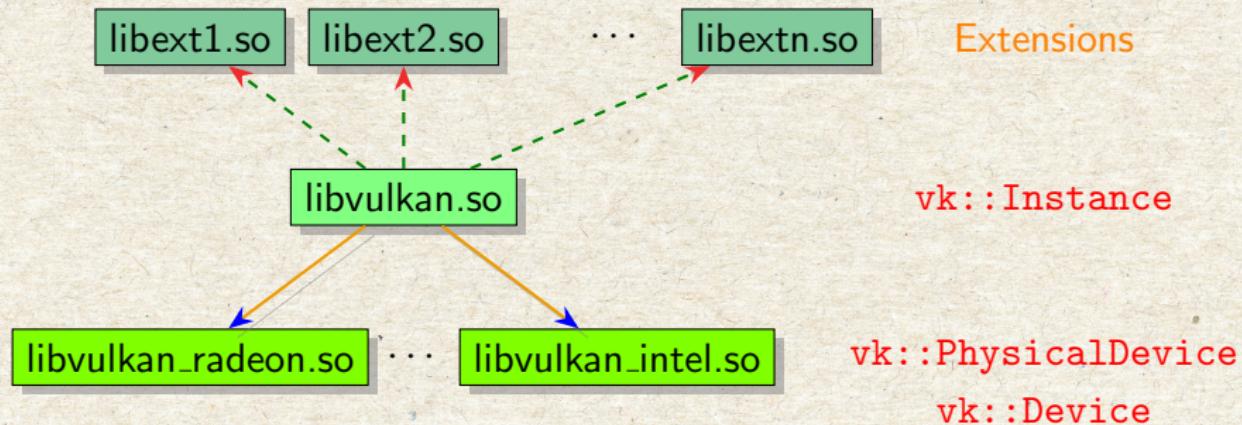
L'API côté hôte

API utilisant Vulkan

Conclusion

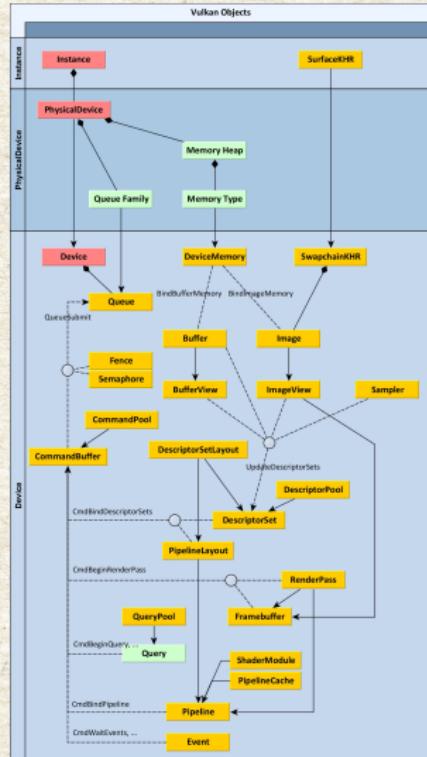
Architecture globale

Vulkan est architecturé pour être facilement extensible pour de nouveaux hardware et rajouter de nouvelles fonctionnalités.



Note : Les extensions doivent être validées et "enregistrées" par Khronos Group.

Architecture de l'API



Object : Objet principal d'une section (**instance** obligatoire)

Permet création autres objets de la section

Object : Objet non typé uniquement indexé par un entier

Appartient et indexé par un objet

→ représente l'ordre de création

← représente une composition

- - - représente une autre relation

Les shaders

- Introduits en Mai 1988 par Pixar dans logiciel **Renderman** ;
- Change comportement de certaines étapes de rendus par ses propres programmes ;
- Introduit dans OpenGL et Direct3D vers 2001 ;
- Personnaliser certaines étapes du rendu d'OpenGL/DirectX ;
- Plusieurs langages développés proches du C/C++ :
 - OpenGL shading language ou **GLSL**
 - DirectX High-Level Shader Language ou **HLSL**
- Les différents shaders existants :
 - **Tessellation** : Subdivise un élément du maillage
 - **Vertex** : Projette un sommet 3D vers écran.
 - **Geometry** : Modifie géométrie d'un polygone
 - **Fragment** : calcule couleur d'un pixel
 - **Compute** : Effectue des calculs parallèles complexes
 - **Raytracing** : calcule intersection rayon-triangle

Spécification des shaders avec Vulkan

- Avec OpenGL, compilation des shaders se fait par l'API ;
- Pour portabilité, obligé avec OpenGL de livrer application avec source des shaders
- Avec Vulkan, langage machine binaire SPIRV
- Facile à transformer en langage machine natif du GPGPU
- compilateurs externes existent indépendant de Vulkan
- Permettent de compiler shaders GLSL ou HLSL en SPIRV
- Exemple de compilateurs permettant de compiler des shaders écrits en GLSL ou en HLSL en SPIRV :
 - `glslang`
 - `glslc`
- API Vulkan permet de charger directement du SPIV pour les shaders

Le compute shader

- Permet d'effectuer des calculs complexes
- Préciser la version du langage utiliser dans votre code :

```
#version 450
```

- Organisation des threads sur grille 1D, 2D ou 3D
- Notion groupe de travail ≡ OpenCL ou bloc Cuda
- A définir en C++ mais aussi dans le shader !

```
#define WORKGROUP_SIZE 32
layout (local_size_x=WORKGROUP_SIZE,
        local_size_y=WORKGROUP_SIZE, local_size_z=1) in;
```

Passer des paramètres à un shader

- Impossible de passer directement paramètres dans `main`.
- Pour passer des paramètres, on peut passer :
 - Des constantes pour de simples valeurs en entrée :

```
layout( push_constant ) uniform Constants {
    vec2 dimensions;
} constants;
```

- Des buffer uniformes pour un groupe de valeurs (pouvant être structurées) en entrée :

```
layout(binding=2, set=0) uniform CameraProperties {
    mat4 viewInverse; mat4 projInverse; vec4 lightPos;
} cam;
```

- Des buffers pour des collections de valeurs :

```
layout(std140, binding = 0) buffer buf {
    vec4 imageData[];
};
```

Repérer la position d'un thread sur la grille de threads

- **Rappel** : chaque thread positionné sur grille 1D/2D ou 3D à l'aide d'un n-uple d'entiers
- La grille est subdivisé en blocs de threads (groupe en GLSL) ;
- Possibilité de connaître le positionnement global du thread exécutant notre code :

```
float x = float(gl_GlobalInvocationID.x) / float(WIDTH);  
float y = float(gl_GlobalInvocationID.y) / float(HEIGHT);
```

- Possibilité de connaître le positionnement dans un bloc du thread exécutant notre code :

```
sharedA[gl_LocalInvocationID.y][gl_LocalInvocationID.x] =  
    A[row * N + (k * 16 + gl_LocalInvocationID.x)];
```

Les étapes de création d'une application Vulkan

Structure "fixe"

- Définir un contexte Vulkan
- Choisir le(s) périphérique(s) physique(s) disponible(s)
- Définir un périphérique logique représentant le(s) périphérique(s) physique(s)
- Choisir une ou plusieurs queues d'exécution selon le(s) type(s) de traitement
- Décrire la structure sous-jacente en mémoire représentée par les buffers utilisées dans les shaders
- Créer des réservoirs pour la création de buffers de commandes et de pipeline
- Créer le(s) buffer(s) de commande
- Créer le(s) pipeline(s) associés au(x) buffer(s) de commande.

Les étapes de création d'une application Vulkan (suite)

Structure "dynamique"

- Créer des buffers en choisissant le type de mémoire dans lequel réserver ;
- Transférer éventuellement des données de l'hôte vers le périphérique ;
- Associer des buffers avec un numéro de binding (et éventuellement de set) utilisés dans les shaders ;
- Exécuter un buffer de commande (ou plusieurs en parallèle).

Les couches logicielles (Layers)

Création d'un contexte : on peut rajouter des couches logicielles.

Couche logicielle (Layer)

- Certaines étapes de l'exécution ont des points d'interruption
- On peut y brancher nos propres couche(s) logicielle(s)
- Possibilité d'enchaîner plusieurs fonctions d'interruption
- **Exemples :**
 - Débogueur pour intercepter et afficher les problèmes détectés par Vulkan ;
 - Sauvegarder des frames à des moments spécifiques
- Possible de demander la liste des couches logicielles préexistantes à l'exécution de l'application.
- Possibilité de programmer propres couches (voir [la documentation fournie par renderdoc](#))

Les extensions

Création d'un contexte : on peut rajouter des extensions

Extensions

- Des fonctionnalités peuvent être rajoutées dynamiquement :
- Deux types de fonctionnalités : celles liées à l'environnement et l'autre à chaque périphérique
- **Exemple :**
 - Ecrire sur un type de surface donnée (Metal, Wayland, X11, Windows)
 - Des fonctionnalités pour supporter l'accélération RTX
- Pour chaque fonctionnalités disponibles, possibilité d'avoir des pointeurs sur les fonctions associées à l'extension
- **Note 1** : Contrairement aux layers, rajout d'une extension doit être approuvée par Khronos Group
- **Note 2** : Certains layers sont associés avec une extension donnée (par exemple pour le déboggage d'applications Vulkan)

Exemple : Rajout d'un outil de validation/débogage

```
std::vector<const char*> enabled_layers{}, enabled_ext{};

auto layer_properties=vk::enumerateInstanceLayerProperties();
bool has_validation_layer_type = false;
for (auto prop : layer_properties)
{
    if (strcmp("VK_LAYER_KHRONOS_validation",prop.layerName)==0)
        has_validation_layer_type = true;
}// for (auto prop : ....)
if (has_validation_layer_type)
    enabled_layers.push_back("VK_LAYER_KHRONOS_validation");

auto extensions=vk::enumerateInstanceExtensionProperties();
for (auto prop : extensions)
{
    if (strcmp(VK_EXT_DEBUG_UTILS_EXTENSION_NAME,prop.extensionName) == 0)
    {
        enabled_ext.push_back(VK_EXT_DEBUG_UTILS_EXTENSION_NAME);
    }
}
```

Création d'un contexte Vulkan

Lors de la création d'un contexte Vulkan, il faut :

- Scanner les couches logicielles disponibles pour le driver Vulkan disponible sur la machine
- Sélectionner les couches logicielles désirées (chaînes de caractère)
- Scanner les extensions disponibles pour le driver Vulkan
- Sélectionner les extensions logicielles désirées (chaînes de caractère)
- Décrire votre application
- Créer le contexte à partir des données précédentes.

Création du contexte

```
vk::ApplicationInfo application_info;
application_info.setApiVersion(vk::ApiVersion13)
    .setPApplicationName("Computing shader")
    .setApplicationVersion(0)
    .setPEngineName("Computing engine")
    .setEngineVersion(0);

vk::InstanceCreateInfo create_info;
create_info.setPApplicationInfo(&application_info)
    .setPEnabledLayerNames(enabled_layers)
    .setPEnabledExtensionNames(enabled_ext);

vk::Instance instance = vk::createInstance(create_info);
```

Choisir son périphérique

- Plusieurs périphériques compatibles avec Vulkan :
 - le CPU (via une couche llvm)
 - le GPU interne au CPU
 - le GPU dédié
 - etc.
- Vulkan chargera un driver spécifique selon le périphérique choisi ;
- On doit donc choisir un (ou des) périphérique(s) ;
- Pour cela, il faut interroger chaque périphérique géré par Vulkan pour sélectionner selon le besoin de l'application.

Choisir son périphérique (suite)

- ➊ Récupérer tous les périphériques compatibles Vulkan :

```
auto devices = instance.enumeratePhysicalDevices();  
for (auto device : devices) { ... }
```

- ➋ Pour chaque périphérique :

- ➌ `dev.getFeatures()` décrit fonctionnalités fines disponibles sur périphérique : test débordements de buffer ? Tracer droites d'épaisseur > 1 ? etc.
- ➌ `dev.getProperties()` décrit propriétés hardware : dédié ou intégré ? taille limite des entités (buffer, constantes, textures, etc.), nom, versions vulkan supporté, etc.
- ➌ `dev.getMemoryProperties()` décrit les types mémoires supportées et leurs capacités : mémoire partagée avec CPU, mémoire dédiée, texture, constante, etc.
- ➌ `dev.enumerateDeviceExtensionProperties()` : les extensions disponibles (\neq extensions de l'instance)
- ➌ `dev.enumerateDeviceLayerProperties()` : les layers disponibles (\subset layers de l'instance)

Création d'un périphérique logique

- Pour contrôler le(s) périphérique(s) réel(s), on doit créer un périphérique **logique** qui nous servira d'API ;
- Le contrôle des GPGPUs se fait par un système de queue de commande ;
- Il existe plusieurs types de queues de commande :
 - Une queue dédiée au transfert de mémoire CPU ⇔ GPU ?
 - Une queue dédiée au calcul uniquement ?
 - Une queue dédiée au graphisme ?
 - Une queue pour le calcul et le graphisme ?
 - etc.
- On doit donc spécifier le(s) type(s) de queues de commande qu'on veut utiliser
- Mais aussi les extensions du périphérique qu'on veut utiliser ;
- Et enfin les fonctionnalités fines qu'on souhaite utiliser.

Création des buffers

Deux types de buffers :

- **Uniform buffer** : Pour stocker un ensemble de données en entrée (lecture seule) pour les shaders
- **Storage buffer** : Buffer stockant un ensemble de valeurs en lecture/écriture modifiable par les shaders

Mêmes protocoles pour les gérer (mais fonctions parfois ≠) :

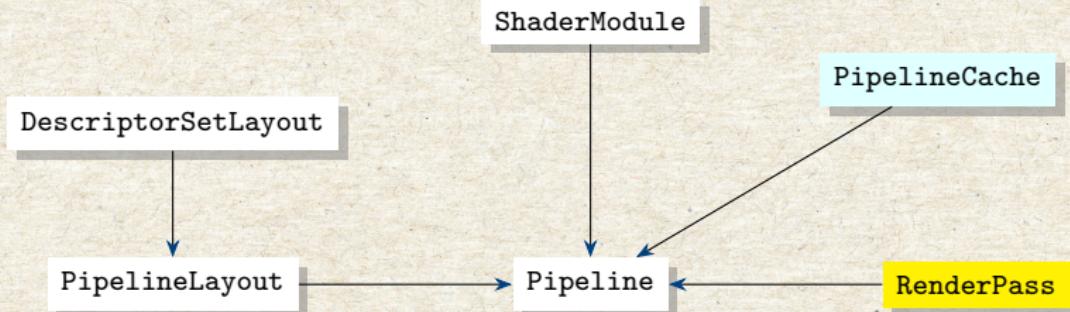
- Création du buffer (descriptif) :
 - Nombre d'octets nécessaire qu'on va devoir réservé ?
 - Pour quel usage ? Uniform ? Storage ? Encore/Décodage vidéo ? etc.
 - Mode d'accès : exclusif (à une queue) ou en concurrence
- Rechercher la mémoire adéquate avec la description du buffer et les caractéristiques voulues par le programmeur
- Allouer la mémoire
- Relier la mémoire avec le buffer (bind)
- Copier données mémoire hôte vers mémoire GPU (optionnel)

Association des buffers avec les shaders

Mécanisme complexe mais souple !

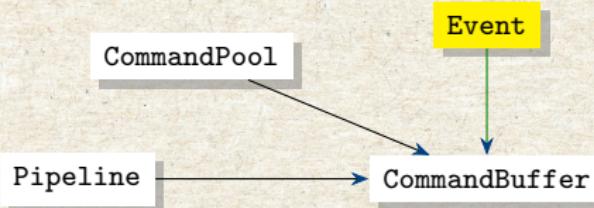
- Pour chaque ensemble (set) dans les shaders, décrire l'organisation mémoire attendue pour chaque shader dans un DescriptorSetLayout.
- Préparer un réservoir de descripteur avec max descripteurs pouvant être alloués
- Allouer les descripteurs à partir de l'organisation mémoire qu'on a décrit : prépare des objets de réceptions des buffers qu'on va utiliser dans chaque shader
- Puis autant de fois qu'on veut, affecter des buffers aux différents objets créés par un descripteur

Déclaration du Pipeline



- Les GPUs calculent une image en plusieurs étapes dont l'ordonnancement est fixe : **Pipeline** graphique.
- Certaines étapes optionnelles et personnalisables (shaders, transferts)
- Vulkan demande de définir nos pipelines graphiques en :
 - Précisant les types de shader devant être exécutés
 - Chargeant et associant les shaders utilisés
 - Précisant leurs points d'entrée (en général **main**)
 - Précisant la description des ensembles de descripteurs utilisés

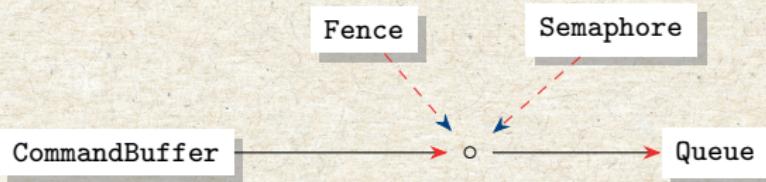
Déclaration du buffer de commande



Pour déclarer un buffer de commande, il faut :

- Déclarer la queue d'exécution qu'on veut utiliser
- Créer un réservoir de buffer de commande utilisant cette queue ;
- Allouer à partir de ce réservoir un ou plusieurs buffers de commande ;
- Pour chaque buffer de commande :
 - Débuter l'enregistrement des commandes qu'on devra exécuter
 - Au moins relier un pipeline et des descripteurs au buffer de commande ;
 - Puis préciser la répartition des threads pour le shader de calcul ;
 - Enfin terminer l'enregistrement.

Exécution du buffer de commande



- L'exécution d'un buffer de commande est asynchrone.
- Il faut prévoir au moins une barrière pour la synchronisation
- Les étapes à suivre sont donc :
 - Préparer une soumission d'un ou plusieurs buffers de commande ;
 - Créer au moins une barrière de synchronisation (Fence)
 - Soumettre notre soumission à la queue avec la barrière de Synchronisation
 - Puis attendre la fin de la synchronisation
 - Enfin détruire la barrière de synchronisation
 - On récupère les données sur l'hôte en mappant un buffer mémoire de sortie

Test performance sur un ensemble de mandelbrot

Ensemble de mandelbrot

- On considère la suite récursive complexe :
$$\begin{cases} z_0 &= 0 \\ z_{n+1} &= z_n^2 + c \end{cases}$$
 où $c \in \mathbb{C}$ choisie.
- On étudie la convergence/divergence de cette suite selon c ;
- Si $\exists N$ tel que $|z_N| \geq 2$, la suite diverge ;
- Dans certaines zones dans disque $\rho < 2$, la suite converge ;
- Sinon impossible montrer convergence/divergence de la suite ;
- Image représente partie plan complexe : à chaque pixel p correspond un c ;
- Si suite "converge", p devient noir, sinon couleur dépend du premier N tel que $|z_N| \geq 2$.

Test de performance sur le calcul de l'ensemble de mandelbrot

But : Calculer en simple précision l'ensemble de mandelbrot sur une image 3200×2400 pixels avec 512 itérations max.

- Test "idéal" car très peu d'accès à la mémoire par rapport à la charge de calcul ;
- Permet de tester le maximum qu'on peut tirer d'un GPU intégré sans mémoire dédiée.

Processeur	cœurs	Tps	GPGPU	cœurs	Tps	rapport
Intel i5 11th gen.	4(8)	795	Intel IRIS X ^e	80	43	18.5
Intel i7 11th gen.	8(16)	617	GeForce RTX 3060	3584	8	77

Temps exprimé en millisecondes.

Plan de l'exposé

Présentation générale

Présentation de l'API Vulkan

Les shaders

L'API côté hôte

API utilisant Vulkan

Conclusion

Comment éviter toute cette complexité ?

- Vulkan très puissant mais complexe à mettre en œuvre ;
- Existe des librairies dédiées au calcul avec Vulkan
- Exemple Kompute
- **Mais**
 - Petit projet...
 - Encore en version beta
 - Manque encore quelques fonctionnalités
 - Mais permet de tester un produit matrice-matrice en Vulkan !
- Autres projets simplifiant Vulkan (encore non testé !) :
 - VUDA : coder Vulkan en Cuda API !
 - NCNN : Neural network interface
 - Site centralisant les projets vulkan

Produit matrice-matrice

- Produit plein : $C = A \cdot B$ avec $A, B, C \in \mathbb{R}^{N \times N}$
- Pour faciliter la vérification du résultat :
 - $A = U_A \cdot V_A^t$ avec $U_A, V_A \in \mathbb{R}^N$;
 - $B = U_B \cdot V_B^t$ avec $U_B, V_B \in \mathbb{R}^N$;
 - $C = U_A \cdot V_A^t \cdot U_B \cdot V_B^t = (V_A | U_B) U_A \cdot V_B^t$

dimension 1024

Processeur	cœurs	GFlops	GPGPU	cœurs	GFlops
Intel i5 11th gen.	4(8)	180	Intel IRIS X ^e	80	55
Intel i7 11th gen.	8(16)	250	GeForce RTX 3060	3584	330

dimension 4096

Processeur	cœurs	GFlops	GPGPU	cœurs	GFlops
Intel i5 11th gen.	4(8)	195	Intel IRIS X ^e	80	60
Intel i7 11th gen.	8(16)	255	GeForce RTX 3060	3584	330

- Des optimisations encore possible dans le shader !
- Remarque 1 : Sur RTX, Cublas affiche 638 GFlops !
- Remarque 2 : C++ : produit par bloc multithreadé 27GFlops !

Plan de l'exposé

Présentation générale

Présentation de l'API Vulkan

Les shaders

L'API côté hôte

API utilisant Vulkan

Conclusion

Conclusions

Vulkan

- permet d'écrire des codes puissants
- Mais plus adapté au graphisme qu'au calcul
- Préférable d'utiliser OpenCL que Vulkan
- Mais Vulkan permet aussi de faire du raytracing hardware
- Interaction entre Vulkan et OpenCL possible
- Interaction OpenCL-Vulkan pour raytracing acoustique

KCompute

- Grande simplification du code ;
- Mais encore en développement ;
- Continuer à tester...