

GMIN 317 – Moteur de Jeux

Game Engine 2 – Optimisations

Université Montpellier 2

Rémi Ronfard

remi.ronfard@inria.fr

<https://team.inria.fr/imagine/remi-ronfard/>

9 octobre 2015

Le but de cette présentation est de fournir tous les moyens mis à votre disposition afin de produire un moteur de jeu le plus efficace possible. Il s'agit d'un cours compliqué, avec de nombreuses nouvelles notions. Des notions que vous ne verrez jamais dans d'autres cours ..

Comment optimiser votre mémoire pour accélérer vos calculs ?

Que faire calculer à votre moteur de jeux ?

Comment sauvegarder les données ?

Comment communiquer entre différents composants

Comment valider son travail ?

- La gestion de la mémoire est un crucial pour obtenir un moteur de jeu efficace. Pour cela il faut contrôler différents points:
 - La quantité d'information produite
 - Le type de données utilisé
 - La localisation de la donnée.

Mémoire

- Il faut mettre en place plusieurs stratégies:
 - Allouer des pools mémoires de tailles maîtrisé
 - Eviter l'allocation multiple de petits éléments.
 - Favoriser la réduction du nombre d'allocation
 - Mettre en place une gestion efficace de la mémoire
 - Eviter la fragmentation de l'information
 - Evaluer les capacités de la mémoire de la machine cible
 - Surveiller les fuites mémoire

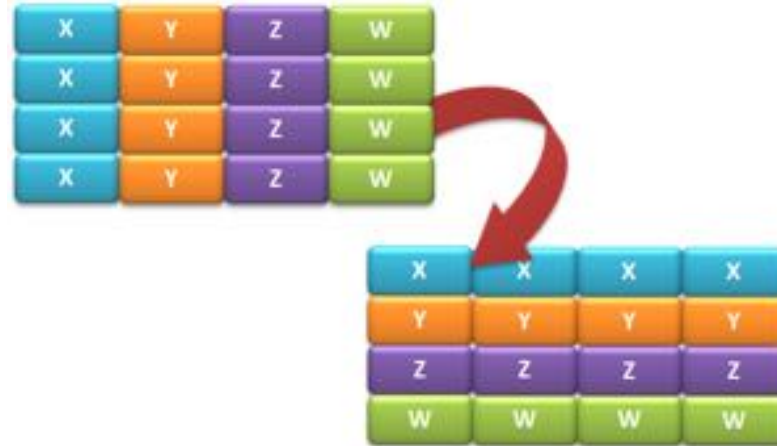
- Les opérations d'écriture et lectures en mémoire ont un coup.
 - Une évolution des architectures (32bits, 64 bits, ...)
 - Evaluer la meilleur solution entre :
 - Entier
 - Float
 - Double
 - ...
 - Ne pas surconsommer en mémoire
 - Réutiliser les bits non utile
 - Réutiliser la mémoire déjà alloué
 - Virgule fixe vs Virgule flottante
 - Précision des calculs (options compilateur, choix de la donnée)

- La localisation de la donnée est un point critique.
- Souvent une structure de données moins adapté à l'humain est plus efficace.
 - Privilège des structure de données SOA
(Structure Of Array)
 - Eviter les données structurés en AOS
(Array Of Structure)

```
struct {  
    uint8_t r, g, b;  
} AoS[N];
```

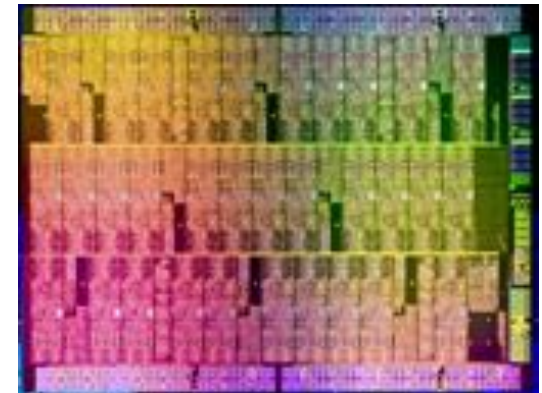
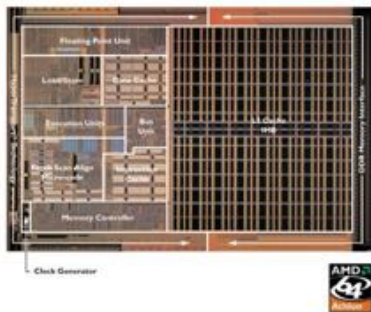
```
struct {  
    uint8_t r[N];  
    uint8_t g[N];  
    uint8_t b[N];  
} SoA;
```


- SOA mieux adapté pour utilisé la vectorisation
- Des compilateurs qui auto-vectorisent
- Un gain de temps de chargement dans le pipelines



Calcul

- Avant les machines étaient single-core.
- Maintenant, les machines sont multi-core.
- Demain, elles seront many-core.
- Loi de Moore:
“Number of transistors on integrated circuits doubles approximately every two years.”



John Carmack (Doom, Quake, Oculus) : fast inverse square root using one iteration of Newton's method

```
float Q_rsqrt( float number ){
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y = number;
    i = * ( long * ) &y; // evil floating point bit level hacking
    i = 0x5f3759df - ( i >> 1 ); // what the fuck?
    y = * ( float * ) &i;
    y = y * ( threehalfs - ( x2 * y * y ) ); // 1st iteration
    // y = y * ( threehalfs - ( x2 * y * y ) ); // 2nd iteration, this can be removed

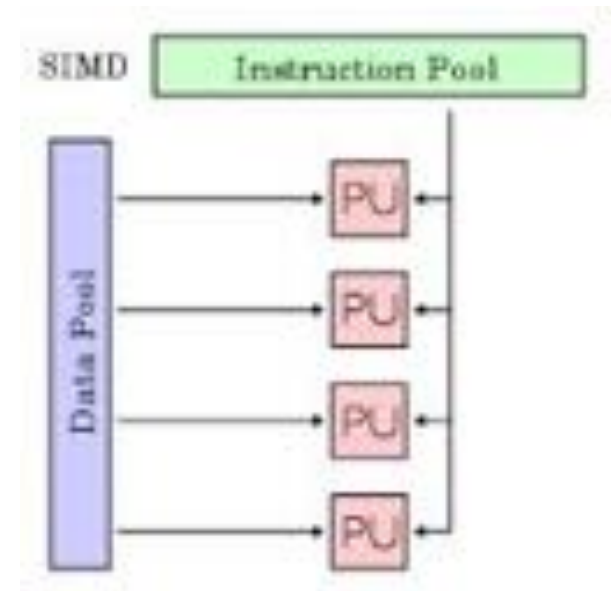
#ifdef Q3_VM
#ifdef __linux__
    assert( !isnan(y) ); // bk010122 - FPE?
#endif
#endif
    return y;
}
```

- Méthode de Newton
- Si on connaît une approximation x_n de la solution de l'équation $f(x) = 0$
- On calcule la tangente à la courbe $f(x)$ et on calcule son intersection avec l'axe Ox
- $y = f'(x_n) (x - x_n) + f(x_n) = 0$
- Ca nous donne une meilleure approximation
- $x_{n+1} = x_n - f(x_n)/f'(x_n)$

- Une nécessité d'utiliser toutes les capacités de calcul
 - CPU:
 - Multi-core
 - Hyperthreading (gain entre 15 et 30%)
 - SIMD Vecteurs
 - GPU
 - CPU many-core
- Mais .. Une programmation complexe à mettre en œuvre.
 - Gabe Newell (2005) : « If writing in-order code [in terms of difficulty] is a one and writing out-of-order code is a four, then writing multicore code is a 10 »
- Différents langage, routines pour accélérer les calculs
- Une uniformisation avec OpenCL
 - API de calcul Open Source

- Comment paralléliser, le cas des CPU:
 - Les vecteurs SIMD
 - Les différents cœurs

- Vecteur SIMD:
 - Single Instruction Multiple Data
 - Différents types
 - SSE -> 4*32 bits
 - AVX -> 8*32 bits
 - AVX2 -> 16*32 bits
- Comment l'intégrer
 - A la main
 - Par auto vectorisation (dépend du compilateur)
 - En utilisant un modèle SPMD



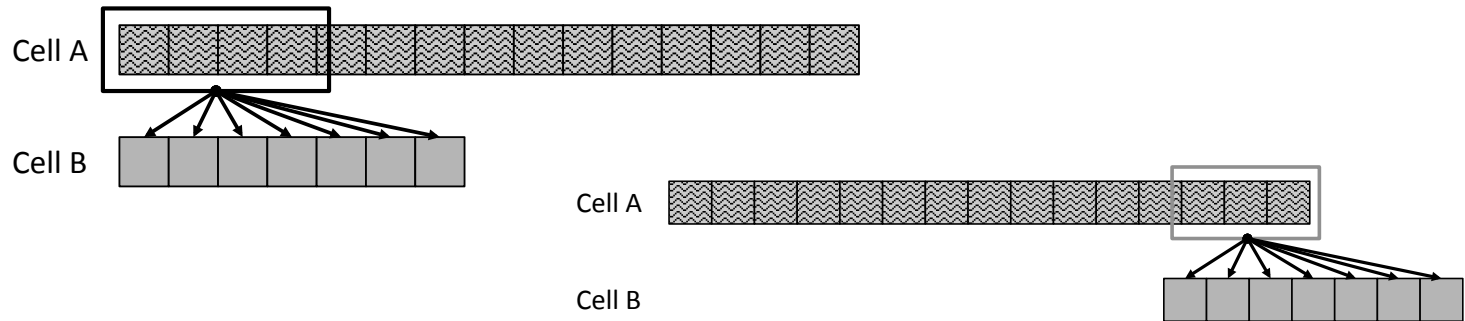
- SPMD = Single Program Multiple Data
- ISPC est un compilateur C.
- Ce compilateur produit des applications compatible SIMD en programmant de manière séquentielle.
- Ce compilateur génère des binaires ou code source compatible SIMD.
- Cette méthode est assez proche de la programmation par kernel de OpenCL.
- Il est possible d'utiliser les fichiers objets produit par cette outil avec GCC, ICC, ...
- Supporte différentes architectures: SSE2, SSE4, AVX1, AVX2, Xeon Phi, ...

```
export void simple(uniform float vin[], uniform float vout[],
                  uniform int count) {
    foreach (index = 0 ... count) {
        float v = vin[index];

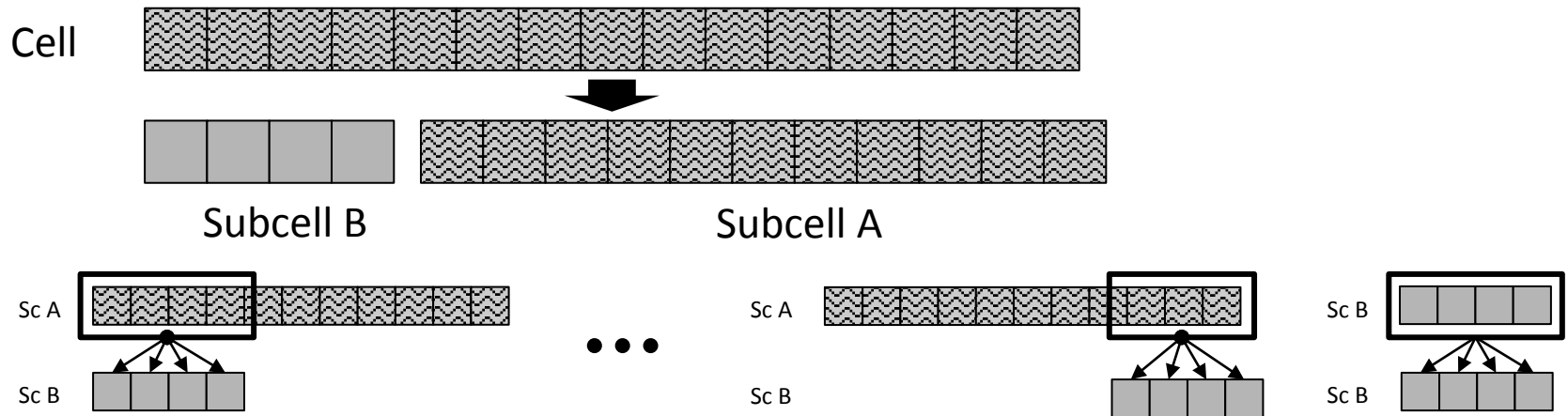
        if (v < 3.)
            v = v * v;
        else
            v = sqrt(v);

        vout[index] = v;
    }
}
```

Cell -> Cell



Cell self



```
#if withoutNR
#define rsqrtNR(v) v= rsqrt(v);
#else
#define rsqrtNR(v) { varying float temp = rsqrt(v); temp *= (temp * temp * (v) - 3.0f) * (-0.5f); v = temp; }
#endif
```

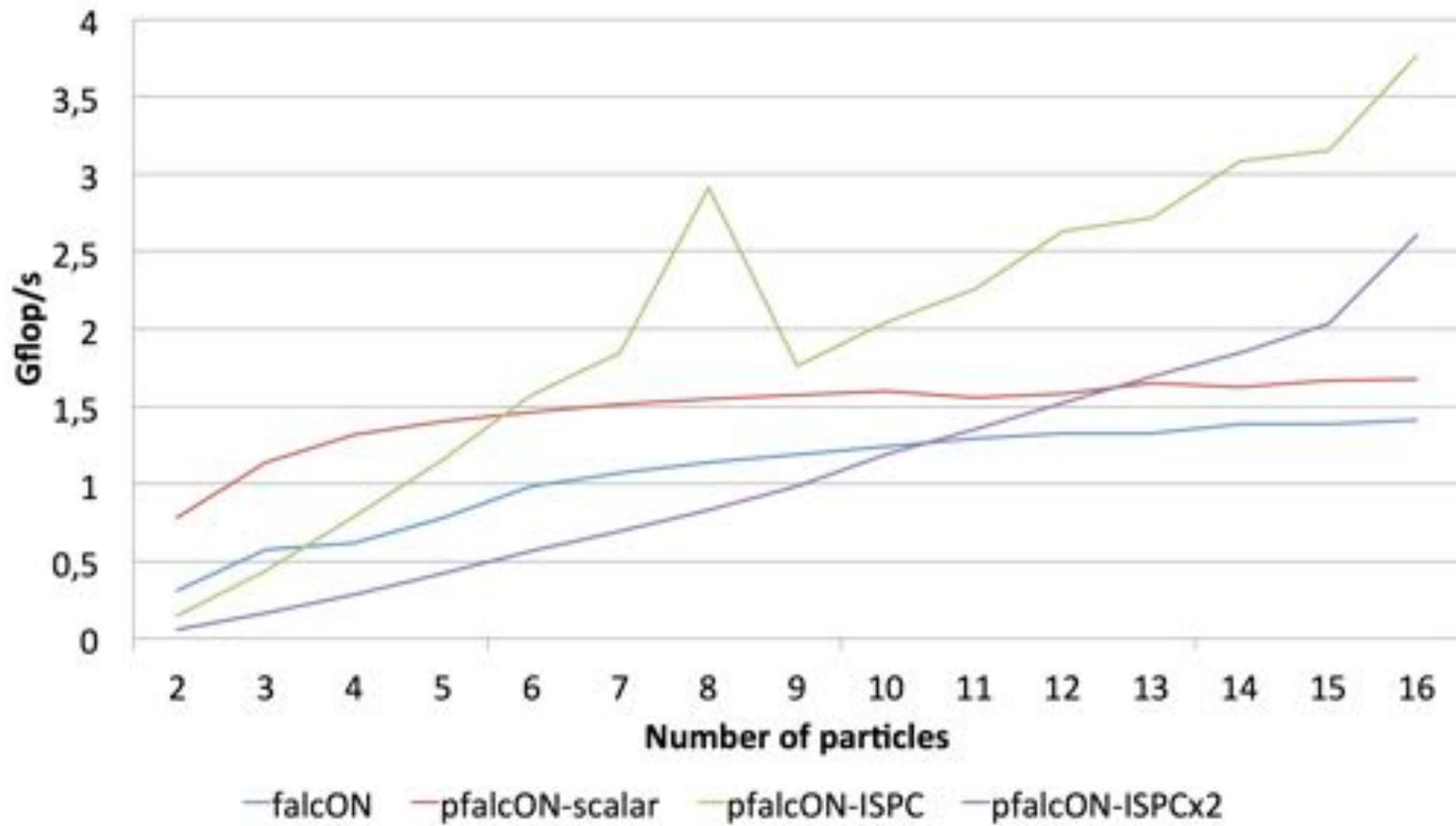
```
struct pfalCONstruct
{
    unsigned int id;
    float SCAL;

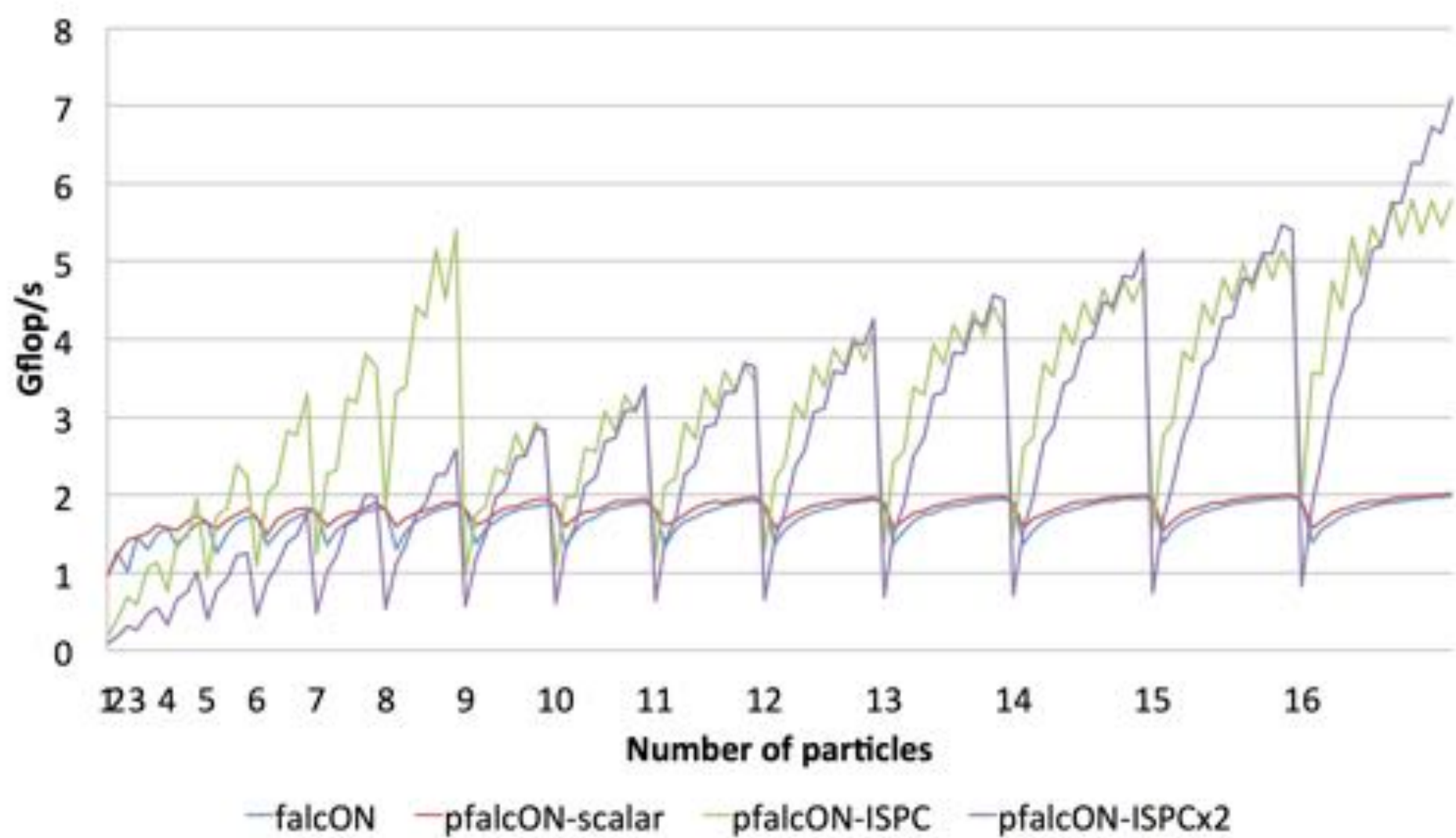
    float a[3];

    int padding[7];

};
```

```
struct pfalCONstructPROP
{
    float POT[4];
};
```





- Programmation par thread
 - Exécution de codes différents
 - Exécution du même code en parallèle



- Comment s'y prendre:

- Fork/Join

- Implémentation bas niveau
 - Ex: dans le cadre d'activité différents

```
tid1 = fork(job1, a1);  
job2(a2);  
join tid1;
```

- Cobegin/Coend

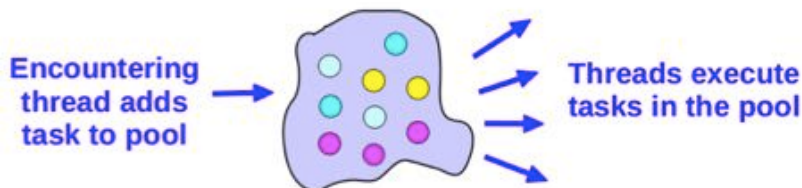
- Méthode la plus simple
 - Méthode la plus utilisé
 - Limité à des boucle non imbriquées
 - Pas compatible avec la récursion

```
cobegin  
    job1(a1);  
    job2(a2);  
coend
```

- Modèle par tâche

- Efficace pour des boucles non limités
 - Efficace pour la récursion

```
spawn(job1(a1));  
spawn(job2(a2));
```



- Comment programmer en multiprocesseur
 - **OpenMP**
 - MPI

- Instructions cobegin/coend:
 - Pour paralléliser une région
 - `#pragma omp parallel ...`
 - Différentes clauses
 - If (expression scalaire)
 - Num_thread (nombre)
 - Private (liste de variables)
 - Firstprivate (liste de variables)
 - Shared (liste de variables)
 - Default (liste de variables)

- Parallélisme appliqué aux boucles:
#pragma omp for
/* boucle for*/
- Une barrière implicite est appliquée à la fin du for
- Plusieurs options:
 - Nowait
 - Schedule
 - #pragmam omp ordered (éviter els race condition)
 - #pragmam omp critical (section critique)
 - ..
 - Plus de détails dans la doc OpenMP

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define CHUNKSIZE 10
#define N 100

int main (int argc, char *argv[])
{
    int nthreads, tid, i, chunk;
    float a[N], b[N], c[N];

    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;

    #pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
    {
        tid = omp_get_thread_num();
        if (tid == 0)
        {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
        printf("Thread %d starting...\n",tid);

        #pragma omp for schedule(dynamic,chunk)
        for (i=0; i<N; i++)
        {
            c[i] = a[i] + b[i];
            printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
        }
    }
}
```

- Le parallélisme par tâches, pour:
 - le parallélisme non défini
 - Les récursions
 - Adopter un modèle consommateur
- Une tâche est :
 - Une unité de travail
 - Dont l'exécution peut démarrer immédiatement, mais elle peut également être différé
 - Composé de :
 - Code d'exécution
 - Environnement de données
- Les tâches sont exécutées par un thread d'une équipe
- Une tâche peut être attaché à un thread
 - Par défaut, elle ne l'est pas

- En OpenMP:
 - `#pragma omp task [clause]`
- Exemples de clauses:
 - If (expression scalaire)
 - Untied
 - Private (liste de variables)
 - Firstprivate (liste de variables)
 - Shared (liste de variables)
 - Default (liste de variables)

```
#include <stdio.h>
#include <omp.h>
int fib(int n)
{
    int i, j;
    if (n<2)
        return n;
    else
    {
        #pragma omp task shared(i) firstprivate(n)
        i=fib(n-1);

        #pragma omp task shared(j) firstprivate(n)
        j=fib(n-2);

        #pragma omp taskwait
        return i+j;
    }
}

int main()
{
    int n = 10;

    omp_set_dynamic(0);
    omp_set_num_threads(4);

    #pragma omp parallel shared(n)
    {
        #pragma omp single
        printf ("fib(%d) = %d\n", n, fib(n));
    }
}
```

- Comment gérer les problèmes de synchronisation
 - Lock
 - Opérations atomiques
- Pour plus de détails je vous conseille de lire la documentation OpenMP
- Mais également de tester les autres librairies...

Stockage

- Gestion des IO disque
 - L'écriture sur le disque a un coup non négligeable.
 - Réserver cette étape pour les sauvegarde massive
 - Pour le faire en temps réel, réaliser l'écriture par un thread (pas le thread principal)
 - Structurer au mieux les données écrites
 - Préférer la sauvegarde binaire
 - Mettre en place un outil de gestion de ressource
- Il est strictement interdit d'utiliser les adresse absolue !

- La sérialisation des données
 - Méthode de stockage des objets
 - Principalement utilisé pour
 - le chargement
 - La sauvegarde des états
 - Les préférences
- Mais C++ n'offre pas de méthode de sérialisation
 - Soit développer ces méthodes
 - Soit utiliser des méthodes existantes

- Un outil de sérialisation doit contenir:
 - Une méthode de sauvegarde des instances
 - Une méthode de sélection de données
 - Plusieurs sauvegarde par exemple
 - Supporter la sauvegarder sous différents formats
 - Release (binaire) vs Debug (verbose)
 - Chargement de différents média
 - Contrôler la taille de la sauvegarde

Réseau

- Pourquoi communiquer entre clients ?
 - Permettre un mode multi-joueurs
 - Permettre un mode massivement multi-joueurs
 - Envoie des informations du master vers le client
- Communication entre les classes de deux applications

- Pour rappel:
 - Le réseau est orienté paquets
 - Un paquet est un petit ensemble d'information
 - Le chemin réseau n'est pas toujours le même
 - Les paquets n'arrivent pas toujours dans leur ordre initial
- De nombreuses méthodes de communications
 - Mais traditionnellement:
 - TCP
 - UDP

- Plusieurs méthodes de communications
 - Client Server
 - P2P
- Besoin de définir un protocole de communications entre les nœuds
- Prévoir des stratégies de validation
- Prévoir des méthodes de sécurité
- Les données doivent être légères
- Adopter une stratégie de load balancing pour les MMO
- Modèles de calculs identique entre les clients
- Serveur possède souvent de meilleure capacité
 - Possibilité de distribuer les calculs sur des grappes de servers

- Validation

- Il est temps d'apprendre à Debugger vos applications de manière efficace
 - Utiliser des outils de profiling
 - Ex: valgrind, gdb
 - Mettre en place des routines de tests
 - Mettre en place un mode debug et release
 - Détecter au plus tôt les erreurs
 - Adopter un modèle de programmation MVC

```
#include <iostream>
```

```
using namespace std;
```

```
int divint(int, int);
```

```
int main() {  
    int x = 5, y = 2;  
    cout << divint(x, y);  
    x = 3; y = 0;  
    cout << divint(x, y);  
    return 0;  
}
```

```
int divint(int a, int b)  
{  
    return a / b;  
}
```

```
g++ -g crash.cc -o crash
```

Lors de l'exécution:

Floating point exception (core dumped)

```
gdb crash
# Gdb prints summary information and then the (gdb) prompt

(gdb) r
Program received signal SIGFPE, Arithmetic exception.
0x08048681 in divint(int, int) (a=3, b=0) at crash.cc:21
21    return a / b;
# 'r' runs the program inside the debugger
# In this case the program crashed and gdb prints out some
# relevant information. In particular, it crashed trying
# to execute line 21 of crash.cc. The function parameters
# 'a' and 'b' had values 3 and 0 respectively.

(gdb) l
# l is short for 'list'. Useful for seeing the context of
# the crash, lists code lines near around 21 of crash.cc

(gdb) where
#0  0x08048681 in divint(int, int) (a=3, b=0) at crash.cc:21
#1  0x08048654 in main () at crash.cc:13
# Equivalent to 'bt' or backtrace. Produces what is known
# as a 'stack trace'. Read this as follows: The crash occurred
# in the function divint at line 21 of crash.cc. This, in turn,
# was called from the function main at line 13 of crash.cc

(gdb) up
# Move from the default level '0' of the stack trace up one level
# to level 1.

(gdb) list
# list now lists the code lines near line 13 of crash.cc

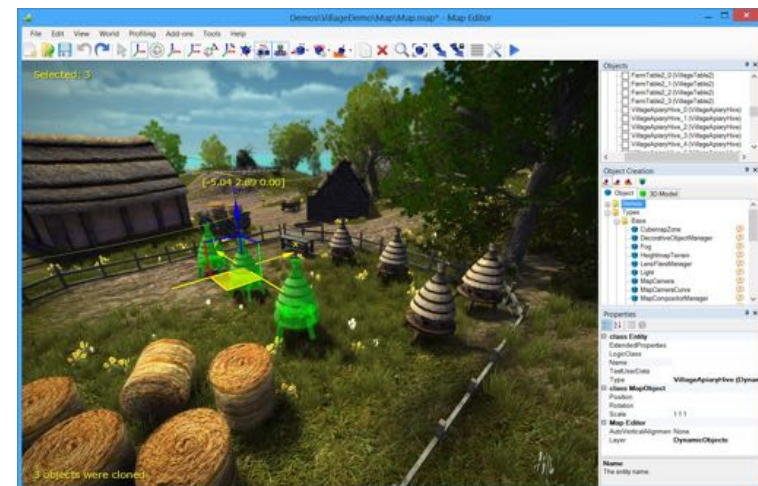
(gdb) p x
# print the value of the local (to main) variable x
```

- Il vous faudra mettre en place des tests unitaires:
 - Procédure permettant de vérifier le bon fonctionnement d'une partie du code
 - Des api pour vous aider ...
 - cppTest
 - Qt
 - Par ex: QtTest
- ```
void TestQString::toUpper()
{
 QString str = "Hello";
 QCOMPARE(str.toUpper(), QString("HELLO"));
}
```

- Avant le TP ..

- Une source d'inspiration pour vous
- Des exemples de Game Engine:
  - Unity 3D
  - Ogre3D
  - XNA
  - Unreal engine
  - Cry engine
  - Source engine
  - Blender Game Engine

- Ogre 3D
  - OO interface en C++
  - Un Framework extensible
  - Un moteur haute performance
  - Multi plateforme
  - D3D et OpenGL
- Contient:
  - Un gestionnaire de scène
  - Un gestionnaire de ressources
  - Un gestionnaire d'animation
  - Un gestionnaire de rendu
  - Des extensions





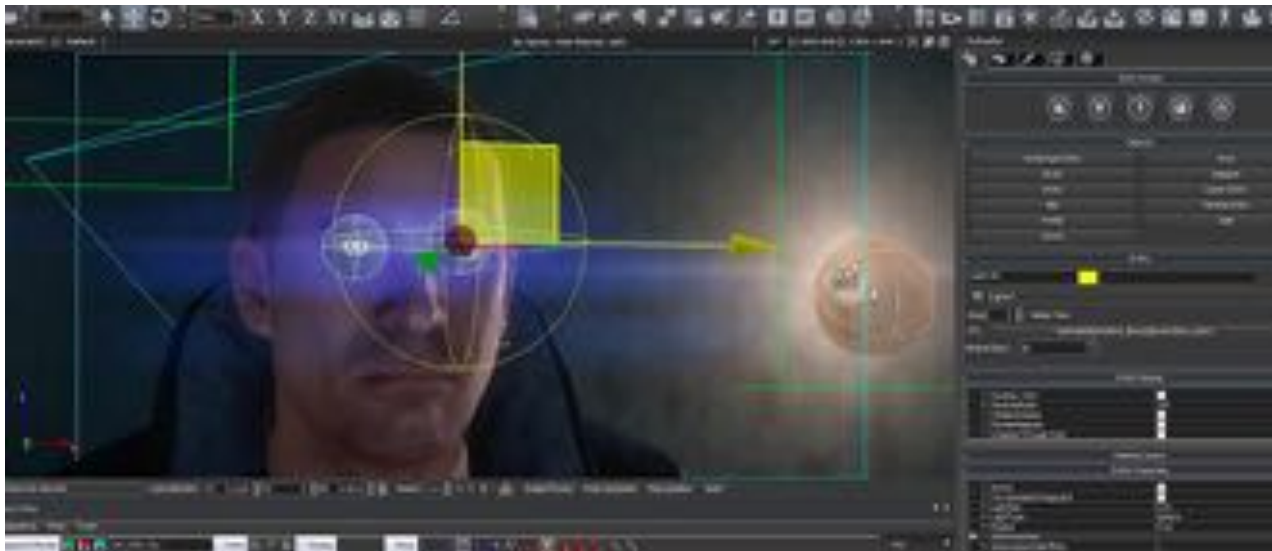
- XNA
  - Moteur de jeu en C#
  - Support de DirectX
  - Windows, Xbox
- Contient:
  - Les classes de bases pour un jeu
  - Un gestionnaire audio
  - Des fonction graphiques 2D 3D
  - La gestion de devices (manette xbox, clavier)
  - Gestionnaire de ressources
  - Gestionnaire de scène



- Unreal engine 3/4  
Open source, PC/Macos
- Des outils :
  - UnrealEd, l'éditeur de niveaux;
  - Unreal Kismet, éditeur de scripts (en Flowgraph) ;
  - Unreal PhAT, éditeur pour la physique dans le jeu (collisions, ragdolls etc) ;
  - Unreal Matinee, éditeur de cinématiques ;
  - Unreal Swarm, pour la distribution de calculs ;
  - L'éditeur SpeedTree, pour créer arbres, feuilles, herbes etc ;
  - FaceFX, pour l'animation des visages.



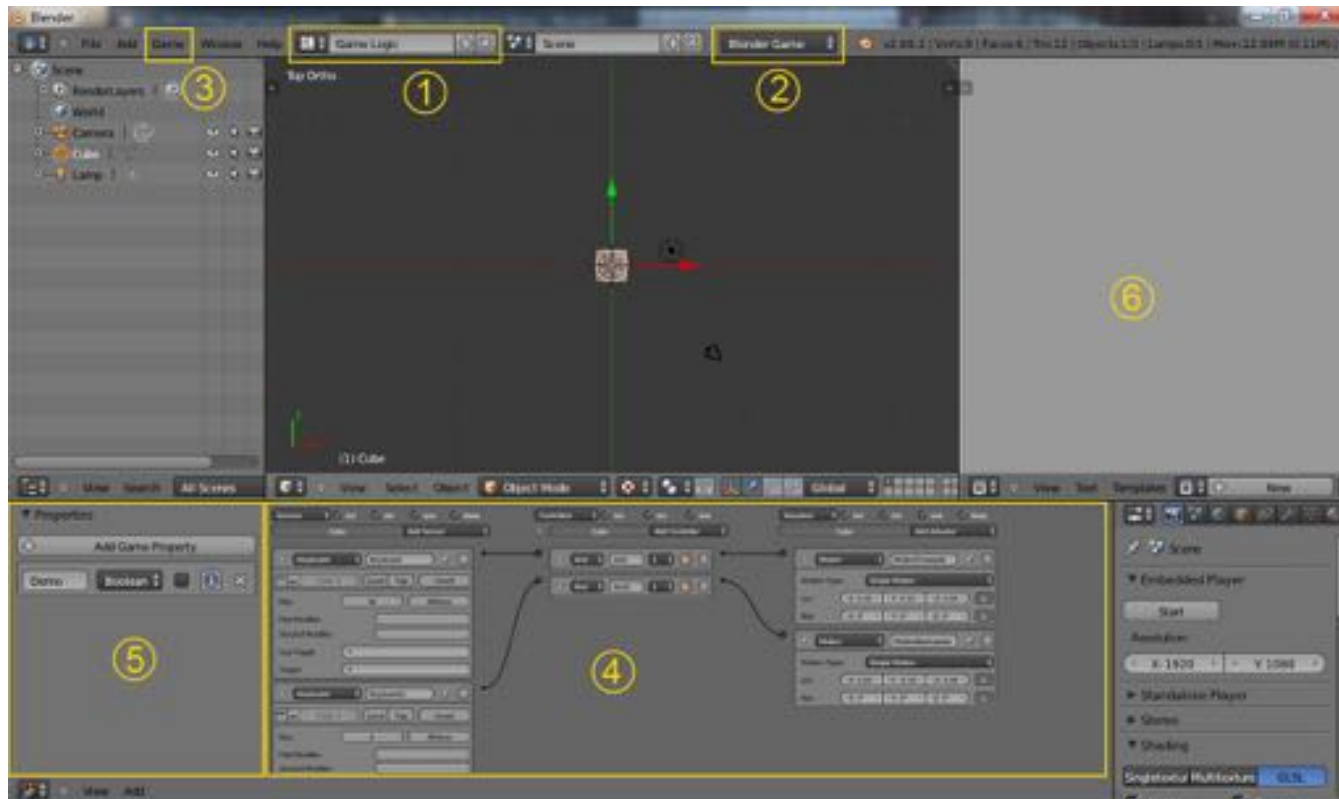
- Cry engine/Cinebox (Crytek)
  - What You See Is What You Play Sandbox editor
  - Cinebox for special effects / cinematics
  - vehicles and physics, scripting, advanced lighting (including real time, moving shadows)
  - Character inverse kinematics and animation blending, Deferred Lighting, Normal Maps & Parallax Occlusion Maps
  - Dynamic music and Advanced Modular AI System.



- Source 1 and 2 engine (Valve)
  - Counter-strike, Half-life, Dota 2
  - PC/Linux/Macos



- Blender Game Engine (open source)
  - Game logic of sensors, controllers, actuators
  - Programmable in python



- Blender Game Engine tools
  - Recast - a state of the art navigation mesh construction tool set for games.
  - Detour - a path-finding and spatial reasoning toolkit.
  - Bullet - a physics engine featuring 3D collision detection, soft body dynamics, and rigid body dynamics
  - Audaspace - a sound library for control of audio. Uses OpenAL or SDL

- Et ....
- Maintenant ...
- Vous pouvez réaliser votre dernier commit pour le TP précédent.

# TP