Scalfmm – Quick Start

This draft will continuously evolve, please be sure you have the latest version.
Last modification: June 17th 2011

Author :
Berenger Bramas (berenger.bramas@inria.fr)

Description :
This quick start is for developers who want to use and customize scalfmm lib. The library contains several directories. The Test directory show several useful examples about how to create the classes you need to make your application running. This quick start extends theses examples and gives some details about what you should/must/must not do. The Tests files are not examples that illustrate how to use the lib, they describes how to program and customize the utilization of the lib.

1 Put particles in the tree

Based on the example: Tests/testOctree.cpp

The octree (FOctree) needs some information to be instantiated. First, it needs the particle & cell classes, the tree height, the suboctree height and the type of leafs. This is given using the templates. Second, you cannot pass the class you want as the template argument. Classes have to respect the fallowing abstract definition :

Abstract Cell (FabstractCell.hpp)

```
class FAbstractCell{
public:
    virtual ~FAbstractCell(){
    }
    virtual MortonIndex getMortonIndex() const = 0;

    virtual void setMortonIndex(const MortonIndex inIndex) = 0;

    virtual void setPosition(const F3DPosition& inPosition) = 0;

    virtual const F3DPosition& getCoordinate() const = 0;

     virtual void setCoordinate(const long inX, const long inY,
const long inZ) = 0;

    virtual bool hasSrcChild() const = 0;

    virtual bool hasTargetsChild() const = 0;

    virtual void setSrcChildTrue() = 0;

    virtual void setTargetsChildTrue() = 0;
};
```

Abstract Particle (FabstractParticle.hpp)

```
class FAbstractParticle{
public:

virtual F3DPosition getPosition() const = 0;
};
```

These methods are needed by the octree and the Fast Multipole Method (FMM) algorithm. They are not related to the kernel used. From here, there are two solutions:
× inherit from the abstract classes and implement the methods
× or use the extensions. Because it is easier, we choose the second solution here.

```
class FBasicParticle : public FExtendPosition{
public:
    virtual ~FBasicParticle(){
    }
};


class FBasicCell : public FExtendPosition, public FExtendMortonIndex {
public:
    virtual ~FBasicCell(){
    }
};
```

As you can see the class names are prefixed with a “F” because these classes exist in the lib so you do not need to create them (excepted for training).

Then you have to choose the type of leaf. There are two types of leaf. The first one has to be used when there is no difference between particles, all particles are sources and targets (FSimpleLeaf). For the second one, the leaf stores separately the particles depending if they are sources or targets (FtypedLeaf); it needs the particles to have some methods detailed later in this document. Here we take the FsimpleLeaf.

The last class is the container class. When the Fmm Algorithm will run, it will call the P2P of your kernel and give you the list of particles. This list is of type ContainerClass. In most of cases we use Flist or Fvector.

Finally, to have something more stable we use typedef and our tree declaration is like :

```
    typedef FVector<FBasicParticle>  ContainerClass;
    typedef FSimpleLeaf<FBasicParticle, ContainerClass >     LeafClass;
    typedef FOctree<FBasicParticle, FBasicCell, ContainerClass , LeafClass >  OctreeClass;

    OctreeClass tree(10, 3,1.0,F3DPosition(0.5,0.5,0.5));
```

The parameter given to the tree constructor are the tree height, the subtree height, the size of the box, and the center of the box.

Finally to insert particles at a random position inside the box:

```
    FBasicParticle particle;
    for(long idxPart = 0 ; idxPart < NbPart ; ++idxPart){
       particle.setPosition(FReal(rand())/RAND_MAX,FReal(rand())/RAND_MAX,FReal(rand())/RAND_MAX);
       tree.insert(particle);
    }
```

2 Iterate on the tree

Based on the example : Tests/testOctreeIter.cpp

From the tree we have built as proposed in the previous section, we can iterate on each cell and each particle. To do so we can use an iterator. This iterator has several methods and it can move on the tree :

× goto :
    × bottom left : to be at the leaves level on the most left leaf
    × right : to be at the most right cell at the current level
    × left : to be at the most left cell at the current level
    × top : to be at level root + 1 in the parent cell of the cell we call the function from
× move :
    × to top : go to the parent cell of the current cell
    × down : go to the leftest child of the current cell
    × right : go to the next cell on the right at the same level

The iterator is never on an empty cell. Lets us assume, for example, that the iterator is on a cell with a morton index 001.100 and that you ask to move right. Even if the next morton index is 001.101, the iterator will move right until it finds a non empty cell. In our example, the next cell could be at 010.011.

In the code, an iteration on the leaves looks like:

```
typedef FOctree<FBasicParticle, FBasicCell, FSimpleLeaf, NbLevels, NbSubLevels> Octree;
Octree::Iterator octreeIterator(&tree);

octreeIterator.gotoBottomLeft();
int counter = 0;
do{
    counter += octreeIterator.getCurrentListSources()->getSize();
} while(octreeIterator.moveRight());
```

In the previous code example we sum the number of sources particles.
Now, if we want to iterate on the cells (including the cells at leaves level), we do:

```
    typename OctreeClass::Iterator octreeIterator(&tree);
    octreeIterator.gotoBottomLeft();
    for(int idxLevel = NbLevels - 1 ; idxLevel >= 1 ; --idxLevel ){
        int counter = 0;
        do{
            ++counter;
        } while(octreeIterator.moveRight());
        octreeIterator.moveUp();
        octreeIterator.gotoLeft();
        std::cout << "Cells at this level " << counter << " ...\n";
    }
```

This will print the number of allocated cell at each level.

3 Use the FMM algorithm

Based on the example : Tests/testFmmAlgorithm.cpp

As we have explain above, the octree is a simple container. It as no behavior related to the FMM. To use the FMM algorithm, you first have to choose a kernel. In the current version of the lib there are several kernels :
× empty kernel : it does nothing
× test FMM kernel : it validate the FMM algorithm
× Fmb kernel : detailed in section XX

Here we only want to introduce the FMM without spending time on the kernels. To do so, we will use the empty kernels first and run a FMM algorithm on it :

```
typedef FBasicParticle              ParticleClass;
typedef FBasicCell                  CellClass;
typedef FVector<ParticleClass>      ContainerClass;


typedef FSimpleLeaf<ParticleClass, ContainerClass >                     LeafClass;
typedef FOctree<ParticleClass, CellClass, ContainerClass , LeafClass >  OctreeClass;
typedef FBasicKernels<ParticleClass, CellClass, ContainerClass >        KernelClass;


typedef   FFmmAlgorithm<OctreeClass,   ParticleClass,   CellClass,   ContainerClass,   KernelClass,
LeafClass >     FmmClass;
```

```
OctreeClass tree(NbLevels, SizeSubLevels,1.0,F3DPosition(0.5,0.5,0.5));
FTestParticle particleToFill;
for(int idxPart = 0 ; idxPart < NbPart ; ++idxPart){
 particleToFill.setPosition(FReal(rand())/RAND_MAX,FReal(rand())/RAND_MAX,FReal(rand())/RAND_MAX);
 tree.insert(particleToFill);
}
```

```
KernelClass kernels;
FmmClass algo(&tree,&kernels);
algo.execute();
```

In the previous example, we repeat the steps presented in the previous sections. Note that by changing the typedef, one does not need to touch the rest of the algorithm.

4 Use the multithreaded FMM

Based on the example : Tests/testFmmAlgorithm.cpp

Here we change the FMM algorithm with a multithreaded version. Nothing changes compared to the sequential version, except that the kernels must have a copy constructor. In fact, we create one kernel per thread to enable fully parallel work. Also, we change the particle class, cell class and kernel class.

```
typedef FTestParticle               ParticleClass;
typedef FTestCell                   CellClass;
typedef FVector<ParticleClass>      ContainerClass;


typedef FSimpleLeaf<ParticleClass, ContainerClass >                     LeafClass;
typedef FOctree<ParticleClass, CellClass, ContainerClass , LeafClass >  OctreeClass;
typedef FTestKernels<ParticleClass, CellClass, ContainerClass >         KernelClass;


typedef  FFmmAlgorithmThread<OctreeClass,  ParticleClass,  CellClass,  ContainerClass,  KernelClass,
LeafClass >     FmmClass;
```

After this code you have to repeat the code of the previous section.

5 Use the Fmb Kernels

Based on the example : Tests/testFmbAlgorithm.cpp

In this part we describe how to use the Fast Multipole with Blas (FMB) kernel. We will first create the right cells and particles. Then we load a “.fma” file.

Finally, we run the FMM algorithm with a FMB kernel.

The fmb particle has to propose several methods :
× the normal method as “basic particle”
× a physical value
× a potential or a forces vector depending on the kernel
To do that, we use extensions whenever possible.

```
class FmbParticle : public FFmaParticle, public FExtendForces, public FExtendPotential {
public:
};
```

Then we create a cell class that has :
× the normal method as “basic cell”
× a pole and a local complexes array

```
class FmbCell : public FBasicCell, public FExtendFmbCell {
public:
};
```

Before creating the tree we can create a Fast Multipole ASCII (FMA) loader :

```
FFMALoader<FmbParticle> loader(filename);
```

From here, we can create an octree that stores the data. To do so, we advise to use typedef again :

```
typedef FmbParticle             ParticleClass;
typedef FmbCell                 CellClass;
typedef FVector<ParticleClass>  ContainerClass;

typedef FSimpleLeaf<ParticleClass, ContainerClass >                     LeafClass;
typedef FOctree<ParticleClass, CellClass, ContainerClass , LeafClass >  OctreeClass;

OctreeClass tree(NbLevels, SizeSubLevels, loader.getBoxWidth(), loader.getCenterOfBox());
```

Then we load the particles and put them in the tree :

```
ParticleClass particleToFill;

for(int idxPart = 0 ; idxPart < loader.getNumberOfParticles() ; ++idxPart){
    loader.fillParticle(particleToFill);
    tree.insert(particleToFill);
}
```

Now we can create the kernel and the algorithm :

```
typedef FFmbKernels<ParticleClass, CellClass, ContainerClass >          KernelClass;
typedef  FFmmAlgorithmThread<OctreeClass,  ParticleClass,  CellClass,  ContainerClass,  KernelClass,
LeafClass > FmmClass;

KernelClass kernels(NbLevels,loader.getBoxWidth());
FmmClass algo(&tree,&kernels);
algo.execute();
```

With the FMB kernel, each particle stores a potential value and a forces vector. We can sum this values :

```
FReal potential = 0;
F3DPosition forces;
typename OctreeClass::Iterator octreeIterator(&tree);
octreeIterator.gotoBottomLeft();
do{
    typename ContainerClass::ConstBasicIterator iter(*octreeIterator.getCurrentListTargets());
    while( iter.hasNotFinished() ){
        potential += iter.data().getPotential() * iter.data().getPhysicalValue();
        forces += iter.data().getForces();


        iter.gotoNext();
    }
} while(octreeIterator.moveRight());


std::cout << "Foces Sum   x = " << forces.getX() << " y = " << forces.getY() << " z = " <<
forces.getZ() << std::endl;
std::cout << "Potential = " << potential << std::endl;
```