Scalfmm – Quick Start

This draft will continuously evolve, please be sure you have the latest version.

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 To put particles in the tree

Based on the example : Tests/testOctree.cpp

The octree (FOctree) needs some informations 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. Your classes have to respect the abstract definition :

Abstract Cell

```
class FAbstractCell{
virtual MortonIndex getMortonIndex() const = 0;


virtual void setMortonIndex(const MortonIndex inIndex) = 0;


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


virtual bool hasSourcesChild() const = 0;


virtual bool hasTargetsChild() const = 0;


virtual void setSourcesChildTrue() = 0;


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

Abstract Particle

```
class FAbstractParticle{
public:

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

Theses methods are needed by the octree and the FMM algorithm, they are not related to the kernel you use. From here you have two solutions : you can inherit from the abstract classes and implement the method or you can 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 this classes exist in the lib so you do not need to create it (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). To the other one, the leaf will store 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 and our tree declaration is like :

```
FOctree<FBasicParticle, FBasicCell, FSimpleLeaf, 10, 3> tree(1.0,F3DPosition(0.5,0.5,0.5));
```

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

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

```
FBasicParticle* const particle = new FBasicParticle();
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 built in the previous section, we can iterate on each cells and each particles. 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 leafs 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 empty cells. For example, if the iterator is on a cell with a morton index 001.100 and if you ask to move right, even if the next morton index

is 001.101 the iterator will move to right until it find a cell. In our example, the next cell could be at 010.011.

In the code, iterate on the leafs 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 to iterate on the cells (including the cells at leafs level) we do :

```
typedef FOctree<FBasicParticle, FBasicCell, FSimpleLeaf, NbLevels, NbSubLevels> Octree;
Octree::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 describe upper, 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 a next part of this document

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 :

```
FBasicKernels<FTestParticle, FTestCell, NbLevels> kernels;

FFmmAlgorithm<FTestKernels, FTestParticle, FTestCell, FSimpleLeaf, NbLevels, SizeSubLevels>
algo(&tree,&kernels);

algo.execute();
```

In the previous example we suppose you already have created a tree and inserted particles. This code runs a fmm algorithm and pass the parameters to the empty

kernel.

4 Use the Thread FMM

Based on the example : Tests/testFmmAlgorithm.cpp

Here we just change the Fmm algorithm with a parallel version. Nothing change from the sequential version excepted that the kernels must have a copy constructor. In fact, we will created a kernels by thread to enable full parallel work.

```
FBasicKernels<FTestParticle, FTestCell, NbLevels> kernels;

FFmmAlgorithmArray<FTestKernels, FTestParticle, FTestCell, FSimpleLeaf, NbLevels, SizeSubLevels>
algo(&tree,&kernels);

algo.execute();
```

In the current version there are several version of parallel fmm but here we choose the best one the array openmp parallel for.

5 Use the Fmb Kernels

Based on the example : Tests/testFmbAlgorithm.cpp

In this part we describe how to use the Fmb kernel. We will first create the right cells and particles. Then we will load a ".fma" file. Finally, we will run the fmm algorithm with a fmb kernels.

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 will use extensions when it is 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 fma loader :

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

From here, we can create an octree that use the information from the fma file :

```
FOctree<FmbParticle,         FmbCell,         FSimpleLeaf,         NbLevels,         SizeSubLevels>
tree(loader.getBoxWidth(),loader.getCenterOfBox());
```

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

```
FmbParticle* particles = new FmbParticle[loader.getNumberOfParticles()];

for(int idxPart = 0 ; idxPart < loader.getNumberOfParticles() ; ++idxPart){
    loader.fillParticle(&particles[idxPart]);
    tree.insert(&particles[idxPart]);
}
```

Now we can create the kernel and the algorithm :

```
FFmbKernelsPotentialForces<FmbParticle, FmbCell, NbLevels> kernels(loader.getBoxWidth());

FFmmAlgorithm<FFmbKernelsPotentialForces,    FmbParticle,    FmbCell,    FSimpleLeaf,    NbLevels,
SizeSubLevels> algo(&tree,&kernels);

algo.execute();
```

With the fmb kernel each particles store a potential value and a forces vector. We can sum this values :

```
FReal potential = 0;
F3DPosition forces;
typedef FOctree<FBasicParticle, FBasicCell, FSimpleLeaf, NbLevels, NbSubLevels> Octree;
Octree::Iterator octreeIterator(&tree);
octreeIterator.gotoBottomLeft();
do{
    FList<FmbParticle*>::ConstBasicIterator iter(*octreeIterator.getCurrentListTargets());
    while( iter.isValide() ){
        potential += iter.value()->getPotential() * iter.value()->getPhysicalValue();
        forces += iter.value()->getForces();


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


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