ScalFMM - Tree Building (Draft)

Berenger Bramas, Olivier Coulaud, Cyrille Piacibello

October 9, 2014

How to Use

There is a ccp class, named FTreeBuilder, which can fill a tree with particles.

Old way

The old method is the following :

1. Load a Particle File.
2. Create an octree.
3. Insert each particle in the Tree.

Code will looks like that :

```
FFmaGenericLoader loader(filename);
OctreeClass tree(TreeHeight, SubTreeHeight,
                 loader.getBoxWidth(), loader.getCenterOfBox());
int nbParts = loader.getNumberOfParticles();
for(int idxPart = 0 ; idxParts < nbParts ; ++idxParts){
  FPoint position;
  FReal physicalValue;
  loader.fillParticles(&position,&physicalValue);
  tree.insert(position,physicalValue);
}
```

New way

1. Load a Particle File.
2. Create an octree.
3. Store each part in a container.
4. Call TreeBuilder on that container and a pointer to the tree.

Code will looks like that :

```
FFmaGenericLoader loader(filename);
OctreeClass tree(TreeHeight, SubTreeHeight,
                 loader.getBoxWidth(), loader.getCenterOfBox());
int nbParts = loader.getNumberOfParticles();
FmaRWParticle * parts = new FmaRWParticle[nbParts];
for(int idxPart = 0 ; idxParts < nbParts ; ++idxParts){
  FPoint position;
  FReal physicalValue;
  loader.fillParticles(&position,&physicalValue);
  parts[idxPart].setPosition(position);
  parts[idxPart].setPhysicalValue(physicalValue);
}
typedef FTreeBuilder<FmaRWParticle,OctreeClass,LeafClass> TreeBuilder;
TreeBuilder::BuildTreeFromArray(parts,nbParts,&tree);
```

How it works

The main idea to reduce the time wasted in the insertion. The operation is long for two reasons, the fact that the parts are included one after the other, and the fact that we must go through all the levels of the tree to find where to store the part.

Finally, the parts are not stored as an array of struct, but as several arrays for each particle's attribute. Those array need to be redimensionned each time we had a particle.

Main Idea to improve the time used : Insert arrays of particles instead of particles.

The ParticleClass (in the example, it's a FmaRWParticle<R,W>)used need to provide at least a getPosition() and a getPhysicalValue() method.

Step by step analysis :

1. The array is copied into a IndexedParticle structure :

   ```
       struct IndexedParticle{
         public:
         MortonIndex index;
         ParticleClass particle;
         operator MortonIndex() const {
           return this->index;
         }
         bool operator<=(const IndexedParticle& rhs){
           return this->index <= rhs.index;
         }
       };
   ```

   The index are setted using GetTreeCoordinate() (method copied from OctreeClass) and GetMortonIndex() methods from FTreeCoordinate.

2. The array is then sorted in order to know where each particle belongs. This step can be skipped if the array is already sorted along MortonIndex. To skip it, just pass true as the third args of BuildTreeFromArray();

   If the array needs to be sorted, then it is, using FQuickSort.hpp, our threaded implementation of the quick sort algorithm.

3. After the sort, we count the number of different leaves. And we store the offset in an other array. Furthermore, we copy for each leaves the position into a FPoint array, and the physical value into a FReal array.

   Example of MortonIndex Distributions and Offset array corresponding :

Parts :	0	1	1	2	4	4	4	5	6	6

Offset :	0	1	3	4	7	8

4. We create every leaves in the tree (note that there are empty). And we store the pointer to each leaves in an array.

5. For each leaves created, we fill it with a new method in FSimpleParticle and FBasicParticleContainer, pushArray. This method is called on the array of position and the array of physical value.