1 Introduction

CombLayer is designed to facilitate the rapid production of complex MCNPX models that depend on a long list of ranged variables and a number of module flags. It is also intended to help with placement of tallies, maintaining consistant material files and some variance reduction.

1.1 Coding Convensions

CombLayer has some coding convensions beyond the standard Scott Myers Efficient C++ convensions [**?**]. These are typcially their for two reasons (i) the in a model-build system rapid build time is essential since it is impossible to have a sub-test framework for any component as the whole MCNPX model is required to check if is it valid, (ii) the code is intended to be used without complete understanding. Therefore as much as possible, each component is independent without code repetition. Akk back-references are to be minimized both in the run-time calling path and in the code build dependencies.

1.1.1 Include files

Include files (.h) are forbidden to include other files. This does several things (a) it reduces the *dependency hell* where it is almost impossible to find the definition of a function and what it depends on. (b) optimization of the include tree can be carried out and dependency continuously observered.

Namespaces are a good method of removing global name pollution but many other C++ programs allows *using namespace X* this is almost 100% forbidden except in the test for that particular namespace unit. This also applies to boost, stl, tr1 etc, to which helps distinguish external functions and domains.

CombLayers geometry is composed of a set of objects that have slightly stronger rules than a typical MCNPX model. Obviously any MCNPX model can be represented as a CombLayer model and in the extreme case that is done by defining one object to contain the MCNPX model. However, the little benefit would be derived from such an approach.

The basic geometric system is to build a number of geometric classes and construct the model by incorporating those into the desired configuration. Each geometric class is designed to be built and an arbitary position and rotation, be of an undetermined number, and interact with its surroundings in a well defined manor.

In object orientated programming, functional rules and properties are normally enforced and added by inherritance and CombLayer follows that pattern. As such most geometry item classes inherrite from base classes within the attachsystem namespace.

1.2 AttachSystem Namespace

The CombLayer system is built around the interaction of FixedComp units, ContainedComp units and LinkUnits. The use of these and their interactions are the basic geometric building tools. These object reside within the attachSystem namespace.

Almost any geometric item can be designated as a FixedComp object. This is done by public inherriting from directly from the FixedComp, or by inherriting from one of the more specialised attachSystem objects e.g. TwinComp or LinearComp.

1.3 FixedComp

The basic FixedComp object holds the origin and the orthoganal basis set (X/Y/Z) for the geometry item being built. In addition it holds a number of LinkUnits which provide information about the outer (and/or inner) surfaces and positions on the geometric item.

As with all Object-Orientated (OO) constructions their is an implicit contract that the inherrited object should adhere to. This is normally expressed as the *Liskov Substitution principle*: This principle states that functions that use pointers/references to the base object must be able to use the objects of derived classes without knowing it. In this case, that means that modification of the origin or the basis set should not invalidate it and that the object should do the expected thing. E.g. if the origin is shifted by 10 cm in the X direction the object should move by 10cm in the X direction. It also means that the basis set must remain orthogonal at all time.

Other than providing an origin and an basis set, the FixedComp has a number of link points. The link points are there to define joining surfaces, points and directions. Each link point defines all three parts.

For example a cube might have 6 linkUnits, and each linkUnit would have a point at the centre of a face, a direction that is normal to the face pointing outwards and a surface definition that is the surface pointing outwards.

$Notethatinthecasethatthelinkpointsdefineaninnervolume, forexampleinavacuumvessel, thenthesurfaces/r$

The actual link surface does not need to be a simple surface. In the case, that an external surface needs multiple surfaces to define the external contact these can be entered into a link-rule. For example, if the cube above was replaces with a box with two cylindrical surfaces the link surface would be defined as the out going cylinder intersection with a plane choosing the side.

In the case of an equiry for the linkSurface (e.g. to do an line intersection) then it is the first surface that takes presidence. However, all actions can be carried out on the link-rule including line intersections etc.

1.4 ContainedComp

The ContainedComp defined both the external and interal enclosed volume of the geometric item. It is most often used to exclude the item from a larger enclosing geometric object: e.g. A moderator will be excluded from a reflector, or it can be used to exclude a part of the geometric item from another geometric object. E.g. two pipes which overlap can have one exclude itself from the other.

In CombLayer, the ContainedComp are considered the primary geometric item, i.e. it is the ContainedComp that is removed from the other items. However, it is used in a two stage process wereby cells are registered to be updated by the ContainedComp at a later date. This was to allow forward dependency planning but has more or less been superseded by the attachControl system.

2 Model Runtime control

C++ programs start from the main() function and in CombLayer the runtime control has been keep mostly in the main() function. Clearly that could be further refactored out but CombLayer lacks the sophisticated top level type abstraction that is required to do this in a generic way, so copy/pasted structure is used with variance to the particular model required. The sole advantage of the abscence of a top level abstraction is that the user is the freedom in writing new objects which allows other programs to be incorporated by making their main function a minor function and directly calling.

The structure of two example main()s will be compared from the units that exist with the stanard CombLayer distribution. That is *bilbau.cxx* and *reactor.cxx.* These build the delft reactor model and the Biblau low energy spallation source.

First part of the code is along list of #include's. They are the mainly dependency list of the objects *Simulation, weightManager, and tallySelector.* This can and should be copied at will. Do not make an file with them all in [see 0.1.1].

At the end of the include section there is typically, one or two model specific includes. These normally include *makeXXX.h* file and anything that they directly depend on. In the case of bilbau it is just *makeBib.h* whilst for reactor it is both *makeDelft.h* and *ReactorGrid.h.*

2.1 makeModel

The makeModel object is the place that creates, initializes and manages inquires for the instances of all the geometric components. Primary objects need to be created and registered with the objectRegister **??**. The makeModel component is

3 Components

3.1 ObjectRegister

The objectRegister is a singleton object [it should be per simulation], which keeps each and then deletes when at its lifetime end, each object registered with it. It only accepts two types of object, a dummy name object and a FixedComp object.

If a dummy object is required, the name (and possibly number) of the object is provided and the objectRegister singleton provides a unique range of cell and surface numbers, typically 10,000 units of each, but can be user selected. This is its only responsibility and to ensure that the name is unique.

Significantly more complex is the FixedComp registration, in this case a *boost::shared_ptr* of FixedComp must be provided by the calling method. Obviously, for a shared_ptr the object memory must be allocated, i.e. an initial `new object(...)` is normally called directly or previously. A typical structrue might be:

```
boost::shared_ptr<BeamPipe> A
   = new BeamPipe("LongPipe");

ModelSupport::objectRegister& OR=
    ModelSupport::objectRegister::Instance();

OR.addObject(A);
```

From this example, the BeamPipe class is inherrited from FixedComp, this is manditory. A tempory reference *OR* is created by calling the static Instance() method. All singletons in CombLayer provide an Instance() method for this purpose. Then the object pointer is referenced to the objectRegiste with *addObject*.

However, hidden from view is a call to objectRegister in FixedComp's constructor, which must be called as all registered object must derive from this class. That occures during the operator new call and results in the allocation of the cell/surface numerical range. If it is necessary to trap that error, the try/catch block must be around the new opertor.