OPEN CASCADE APPLICATION FRAMEWORK FUNCTION MECHANISM

USER'S GUIDE

White Paper

NOTICE FOR USERS:

This User Guide is a general instruction for Open CASCADE Technology study. It may be incomplete and even contain occasional mistakes, particularly in examples, samples, etc.

OPEN CASCADE S.A.S. bears no responsibility for such mistakes. If you find any mistakes or imperfections in this document, or if you have suggestions for improving this document, please, contact us and contribute your share to the development of Open CASCADE Technology: bugmaster@opencascade.com

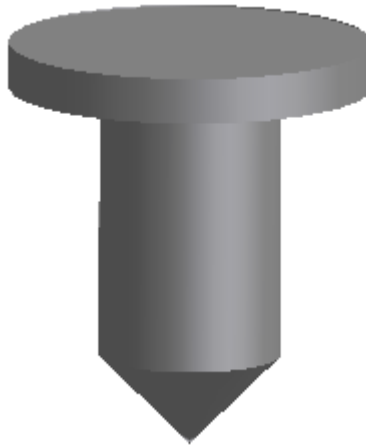**http://www.opencascade.com/contact/**

Introduction

This guide describes the usage of the Function Mechanism of Open CASCADE Application Framework on a simple example.

This example represents a "nail" composed by a cone and two cylinders of different radius and height:

Picture 1. A nail

These three objects (a cone and two cylinders) are independent, but the Function Mechanism makes them connected to each other and representing one object – a nail.

The object "nail" has the following parameters:

- o The position of the nail is defined by the apex point of the cone. The cylinders are built on the cone and therefore they depend on the position of the cone. In this way we define a dependency of the cylinders on the cone.

- o The height of the nail is defined by the height of the cone. Let's consider that the long cylinder has 3 heights of the cone and the header cylinder has a half of the height of the cone.

- o The radius of the nail is defined by the radius of the cone. The radius of the long cylinder coincides with this value. Let's consider that the header cylinder has one and a half radiuses of the cone.

So, the cylinders depend on the cone and the cone parameters define the size of the nail.

It means that re-positioning the cone (changing its apex point) moves the nail, the change of the radius of the cone produces a thinner or thicker nail, and the change of the height of the cone shortens or prolongates the nail.

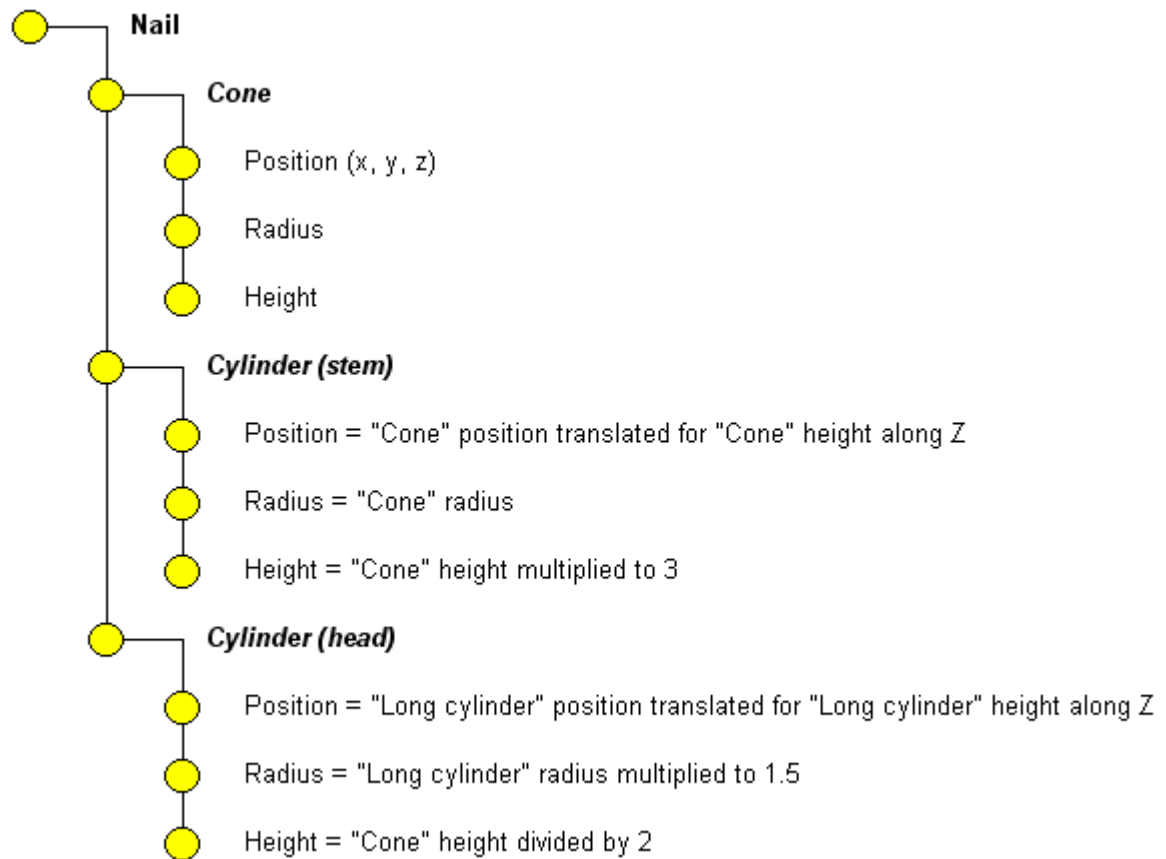 It is suggested to examine the programming steps needed to create a 3D parametric model of the "nail".

This guide describes in detail usage of the Function Mechanism. Other aspects, such as the data model and the interfaces are mentioned in brief.

Step 1: Data Tree

The first step consists in model data allocation in the OCAF tree. In other words, it is necessary to decide where to put the data.

In this case, the data can be organized into a simple tree using references for definition of dependent parameters:

Picture 2. A data tree of a nail

The "nail" object has three sub-leaves in the tree: the cone and two cylinders.

The cone object is independent.

The long cylinder representing a "stem" of the nail refers to the corresponding parameters of the cone to define its own data (position, radius and height). It means that the long cylinder depends on the cone.

The parameters of the head cylinder may be expressed through the cone parameters only or through the cone and the long cylinder parameters. It is suggested to express the position and the radius of the head cylinder through the position and the radius of the long cylinder, and the height of the head cylinder through the height of the cone. It means that the head cylinder depends on the cone and the long cylinder.

Step 2: Interfaces

The interfaces of the data model are responsible for dynamic creation of the data tree of the represented at the previous step, data modification and deletion.

The interface called `INail` should contain the methods for creation of the data tree for the nail, setting and getting of its parameters, computation, visualization and removal.

Creation of the nail

This method of the interface creates a data tree for the nail at a given leaf of OCAF data tree.

It creates three sub-leaves for the cone and two cylinders and allocates the necessary data (references at the sub-leaves of the long and the head cylinders).

It sets the default values of position, radius and height of the nail.

Setting and getting the data of the nail

The nail has the following user parameters:

- o   The position – coincides with the position of the cone

- o   The radius of the stem part of the nail – coincides with the radius of the cone

- o   The height of the nail – a sum of heights of the cone and both cylinders

The values of the position and the radius of the nail are defined for the cone object data. The height of the cone is recomputed as 2 * heights of nail and divided by 9.

Computation

The Function Mechanism is responsible for re-computation of the nail. It will be described in detail later in this document.

A data leaf consists of the reference to the location of the real data and a real value defining a coefficient of multiplication of the referenced data.

For example, the height of the long cylinder is defined as a reference to the height of the cone with coefficient 3 (see picture 2). The data leaf of the height of the long cylinder should contain two attributes: a reference to the height of cone and a real value equal to 3.

Visualization

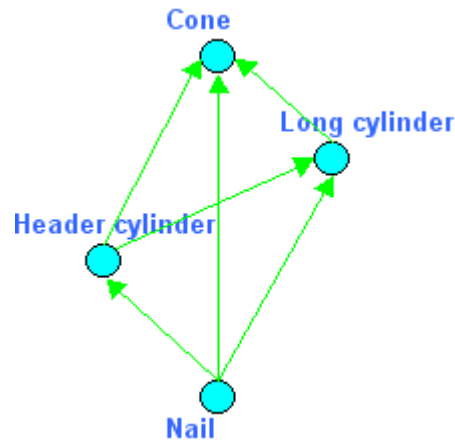 The shape resulting of the nail function can be displayed using the standard OCAF visualization mechanism.

Removal of the nail

 To automatically erase the nail from the viewer and the data tree it is enough to clean the nail leaf from attributes.

Step 3: Functions

The nail is defined by four functions: the cone, the two cylinders and the nail function.

The function of the cone is independent. The functions of the cylinders depend on the cone function. The nail function depends on the results of all functions:

Picture 3. A graph of dependencies between functions

Computation of the model starts with the cone function,. then the long cylinder, after that the header cylinder and, finally, the result is generated by the nail function at the end of function chain.

The Function Mechanism of Open CASCADE Technology creates this graph of dependencies and allows iterating it following the dependencies. The only thing the Function Mechanism requires from its user is the implementation of pure virtual methods of *TFunction_Driver*:

   o   *::Arguments*() – returns a list of arguments for the function

   o   *::Results()* – returns a list of results of the function

These methods give the Function Mechanism the information on the location of arguments and results of the function and allow building a graph of functions. The class *TFunction_Iterator* iterates the functions of the graph in the execution order.

The pure virtual method *TFunction_Driver::Execute*() calculating the function should be overridden.

The method *::MustExecute*() calls the method ::*Arguments*() of the function driver and ideally this information (knowledge of modification of arguments of the function) is enough to make a decision whether the function should be executed or not. Therefore, this method usually shouldn't be overridden.

Implementation

The cone and cylinder functions differ only in geometrical construction algorithms. Other parameters are the same (position, radius and height).

It means that it is possible to create a base class – function driver for the three functions, and two descendent classes producing: a cone or a cylinder.

For the base function driver the methods *::Arguments*() and *::Results*() will be overridden. Two descendent function drivers responsible for creation of a cone and a cylinder will override only the method *::Execute*().

The method *::Arguments*() of the function driver of the nail returns the results of the functions located under it in the tree of leaves (see the picture 2). The method *::Execute*() just collects the results of the functions and makes one shape – a nail.

This way the data model using the Function Mechanism is ready for usage. Don't forget to introduce the function drivers for a function driver table with the help of *TFunction_DriverTable* class.

Appendix 1

This appendix gives an example of the code for iteration and execution of functions.

```
// The scope of functions is defined.
Handle(TFunction_Scope) scope = TFunction_Scope::Set( anyLabel );


// The information on modifications in the model is received.
TFunction_Logbook& log = scope->GetLogbook();


// The iterator is iInitialized by the scope of functions.
TFunction_Iterator iterator( anyLabel );
Iterator.SetUsageOfExecutionOrder( true );


// The function is iterated,  its dependency is checked on the modified data and  executed if necessary.
for (; iterator.more(); iterator.Next())
{
        // The function iterator may return a list of current functions for execution.
        // It might be useful for multi-threaded execution of functions.
        const TDF_LabelList& currentFunctions = iterator.Current();


        //The list of current functions is iterated.
        TDF_ListIteratorOfLabelList currentterator( currentFucntions );
        for (; currentIterator.More(); currentIterator.Next())
        {
            // An interface for the function is created.
            TFunction_IFunction interface( currentIterator.Value() );


            // The function driver is retrieved.
            Handle(TFunction_Driver) driver = interface.GetDriver();


            // The dependency of the function on the  modified data is checked.
            If (driver->MustExecute( log ))
            {
                    // The function is executed.
                    int ret = driver->Execute( log );
                    if ( ret )
```

return false;

} // end if check on modification

} // end of iteration of current functions

} // end of iteration of functions.

Appendix 2

This appendix gives an example of the code for a cylinder function driver. In order to make the things clearer, the methods *::Arguments*() and *::Results*() from the base class are also mentioned.

// A virtual method ::Arguments( ) returns a list of arguments of the function.

CylinderDriver::Arguments(TDF_LabelList& args)

{

    // The direct arguments, located at sub-leaves of the fucntion, are collected (see picture 2).

    TDF_ChildIterator cIterator(Label(), false);

    for (; cIterator.More(); cIterator.Next())

    {

         // Direct argument.

         TDF_Label sublabel = cIterator.Value();

         Args.Append(sublabel);

         // The references to the external data are checked.

         Handle(TDF_Reference) ref;

         If (sublabel.FindAttribute(TDF_Reference::GetID(), ref))

         {

             args.Append(ref->Get());

         }

}

// A virtual method ::Results( ) returns a list of result leaves.

CylinderDriver::Results(TDF_LabelList& res)

{

         // The result is kept at the function label.

         Res.Append(Label());

}

// Execution of the function driver.

Int CylinderDriver::Execute(TFunction_Logbook& log)

{

         // Position of the cylinder – position of the first function (cone)

         //is elevated along Z for height values of all previous functions.

         gp_Ax2 axes = …. // out of the scope of this guide.

         // The radius value is retrieved.

```
// It is located at second child sub-leaf (see the picture 2).
TDF_Label radiusLabel = Label().FindChild( 2 );

// The multiplicator of the radius ()is retrieved.
Handle(TDataStd_Real) radiusValue;
radiusLabel.FindAttribute( TDataStd_Real::GetID(), radiusValue);

// The reference to the radius is retrieved.
Handle(TDF_Reference) refRadius;
RadiusLabel.FindAttribute( TDF_Reference::GetID(), refRadius );

// The radius value is calculated.
double radius = 0.0;
if ( refRadius.IsNull() )
        radius = radiusValue->Get();
else
{
        // The referenced radius value is retrieved.
        Handle(TDataStd_Real) referencedRadiusValue;
        RefRadius->Get().FindAttribute(TDataStd_Real::GetID() ,referencedRadiusValue );
        radius = referencedRadiusValue->Get() * radiusValue->Get();
}

// The height value is retrieved.
double height = … // similar code to taking the radius value.

// The cylinder is created.
TopoDS_Shape cylinder = BRepPrimAPI_MakeCylinder(axes, radius, height);

// The result (cylinder) is set
TNaming_Builder builder( Label() );
Builder.Generated( cylinder );

// The modification of the result leaf is saved in the log.
log.SetImpacted( Label() );

return 0;}
```