Synopsis

Nitin Yadav
B.Tech - Computer Science
Dronacharya College of Engineering, Gurugram

and

Diptangshu Dey
B.Tech - Computer Science
NIT Durgapur

Development of Tree-node Management API for OpenFOAM GUI

This project aims to extend the OpenFOAM GUI software by creating an intermediate API. OpenFOAM[1] does not have an integrated GUI, but there are many third-party GUI software products. However, they are inadequately comprehensive for the OpenFOAM solution workflow. To achieve that, this project aims to make an API as an intermediate piece of software between OpenFOAM and the GUI software. The API will give the capability to write scripts on top of it and hence extend the GUI software. It also provides the freedom to use any GUI as the front end of the software.

References

[1] Hrvoje Jasak. "OpenFOAM: open source CFD in research and industry". In: *International Journal of Naval Architecture and Ocean Engineering* 1 (Dec. 2009), pp. 89–94. DOI: 10.3744/JNAOE.2009.1.2.089.

FOSSEE, IIT Bombay
OpenFOAM GUI
August 28, 2023

Development of Tree-node Management API for OpenFOAM GUI

Nitin Yadav
B.Tech - Computer Science
Dronacharya College of Engineering, Gurugram

and

Diptangshu Dey
B.Tech - Computer Science
NIT Durgapur

Acknowledgement

We would like to express our sincere gratitude to **Prof. Janani Srree Murllidharan** and **Mr Rajdeep Adak** for their guidance and support during our summer fellowship under the **OpenFOAM GUI project at FOSSEE IIT Bombay**. Their expertise and knowledge in the field of computational fluid dynamics and software development have been invaluable to us, and we are truly grateful for their willingness to share their time and insights with us.

We would also like to thank the staff at FOSSEE IIT Bombay for their warm welcome and support during our fellowship. It was a pleasure to work with such a talented and dedicated team.

This report would not have been possible without the support of all of the people mentioned above. We are deeply grateful for their contributions.

Contents

List of Figures

1 Introduction

OpenFOAM[1] is a free open source C++ toolbox for solving continuum mechanics problems. It requires the user to manually write the case files in text format, which may be inconvenient. Having a Graphical User Interface(GUI) for the software makes it easier to interact with it without having to learn the technicalities.

The OpenFOAM GUI project aims to develop a tool to alleviate the process of generating FOAM cases.

OpenFOAM does not have an integrated GUI, but there are many third-party GUI software products. However, they are inadequately comprehensive for the OpenFOAM solution workflow. To achieve that, we have made an API as an intermediate piece of software between OpenFOAM and the GUI software. The API will give the capability to write scripts on top of it and hence extend the GUI software. It also provides the freedom to use any GUI as the front end of the software.

The API takes input from the GUI about the parameters in the OpenFOAM case and interfaces with them in Python. It then uses the given data to generate OpenFOAM case files. It can also take a case file as input and generate a tree, allowing the GUI to represent the case files.

Currently, there are two GUI interfaces, a Blender addon and a PyQt application. Blender software provides a way to interact with its components through the 'bpy' python package. Blender also has a rich node system to represent hierarchical data. Hence a GUI is made as a Blender addon using the nodes to represent OpenFOAM case files.Along with the Blender Addon, a standalone GUI has also been developed using PyQt[2]. The PyQt library is chosen for the front-end as it allows for the creation of cross-platform GUI in an easier, faster and scalable manner. It also represents the data through nodes using the PyQt's Graphics View Framework.

2 Implementation

2.1 System Design

The architecture diagram of the API can be found in the given figure 1. The API is implemented keeping in mind the Model View Controller (MVC) architecture. In this architecture, the data of the software is stored in the model, the user interacts with the view, and the information is communicated through the controller. The controller handles the data that gets sent to the view from the model, and vice versa. We store the data in the form of YAML files. The YAML files store all the possible values for each OpenFOAM parameter. The API reads the YAML files and gives the user the option of choosing the proper parameter that the user wants in the view. The parameters set by the user in the view are stored in the form of a tree in the controller. When all the parameters are set, the OpenFOAM case files are generated by the controller. These case files that are generated can also be edited graphically. MVC helps keep the front end and the back end of the software separate and independent of each other. Currently, there is some direct connection between the view and the model, but it will be removed in the future.

2.2 Tree Nodes

Inside the API controller, the data of each text file is stored in the form of a free. The tree is generated using the Anytree[3] library in Python. Each node of the tree is a custom data structure. This Data structure aims to represent an OpenFOAM dictionary inside Python.

At the bottom level, custom classes are used to store the basic data types, such as PropertyInt for integer data, PropertyFloat for float data, PropertyString for string data, and EnumProperty for a choice between strings. Above that, the KeyData class is used to store them in a set that can be accessed using a key similar to OpenFOAM Dictionaries.

Above KeyData, the nodes are created using anytree's node class, and thus the tree is made. This setting allows for easy scripting access to the trees for each file and also the nesting of one node inside another.

Finally, each Node of the tree is created using the Foam class, which extends the NodeMixin class from Anytree and enables the Foam class as a tree Node. Attributes can be entered in the node as key-value pairs in the class constructor itself.

The representation of a sample `fvSolution` file in the form of a tree is shown in the figure 2

2.3 Blender Addon

The Blender addon uses the existing node interface of Blender.

2.3.1 Main Window

The main window creates a custom node tree type called `OpenFOAM Nodes` where only nodes representing OpenFOAM dictionaries can be stored. Each node tree stores each text file of the OpenFOAM case file.

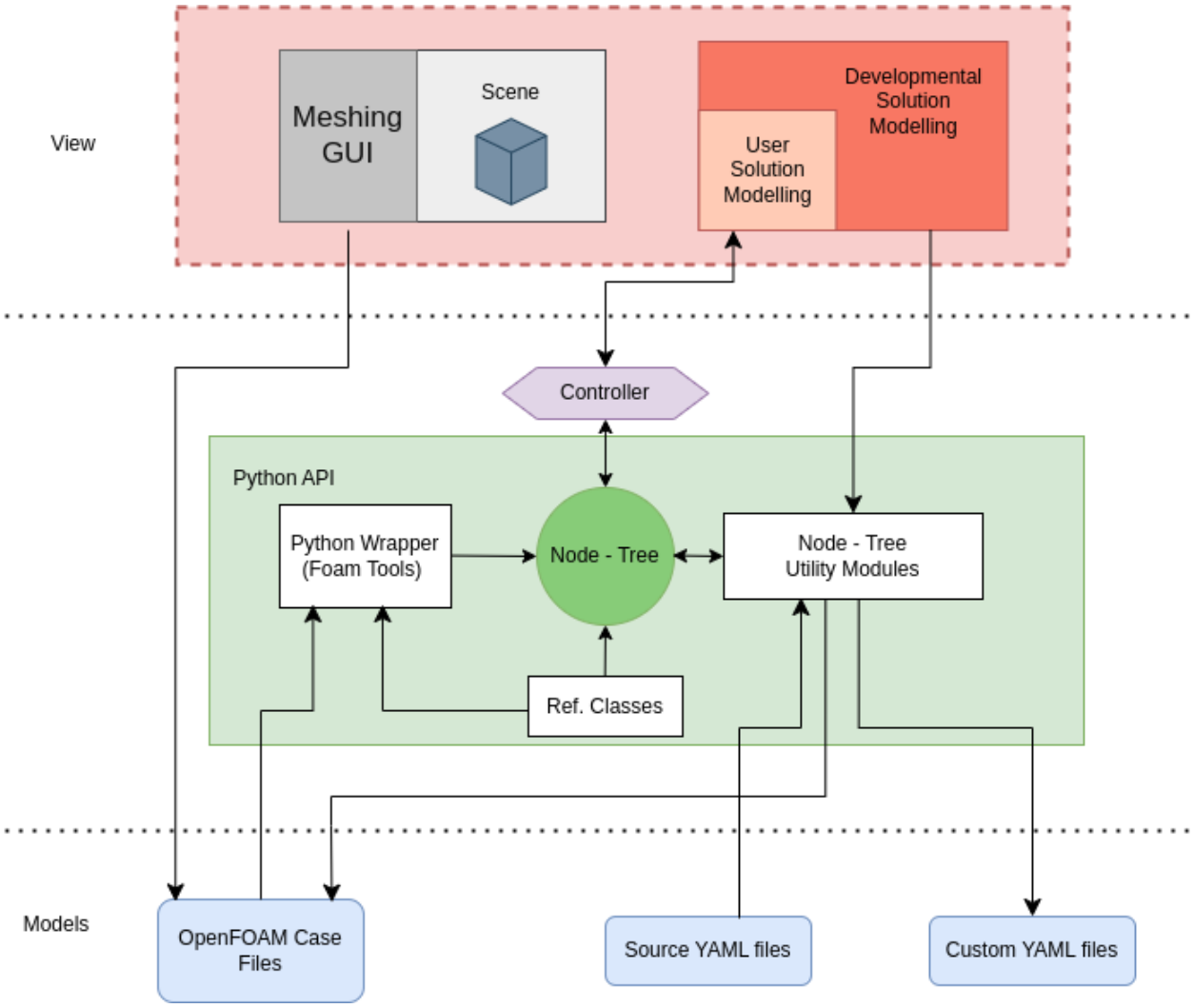

Figure 1: System Design of Venturial

2.3.2 Dictionary Node

The Dictionary Node has a parameter called 'title' which is the name of the dictionary it is storing. Input and output of the node are made using a custom Node Socket class called 'NodeSocketIn' and 'NodeSocketOut' respectively.

Input sockets each store a key, and an array. The key and array are stored as key-value pairs in the dictionary node. This is done to mimic the OpenFOAM dictionary in the node. The array can be manipulated with the help of the options provided in the side panel. Alternatively, instead of providing values in the array directly, another dictionary output can be plugged into the input socket, thus nesting a dictionary inside another.

The output socket outputs the dictionary data that is stored inside the node.

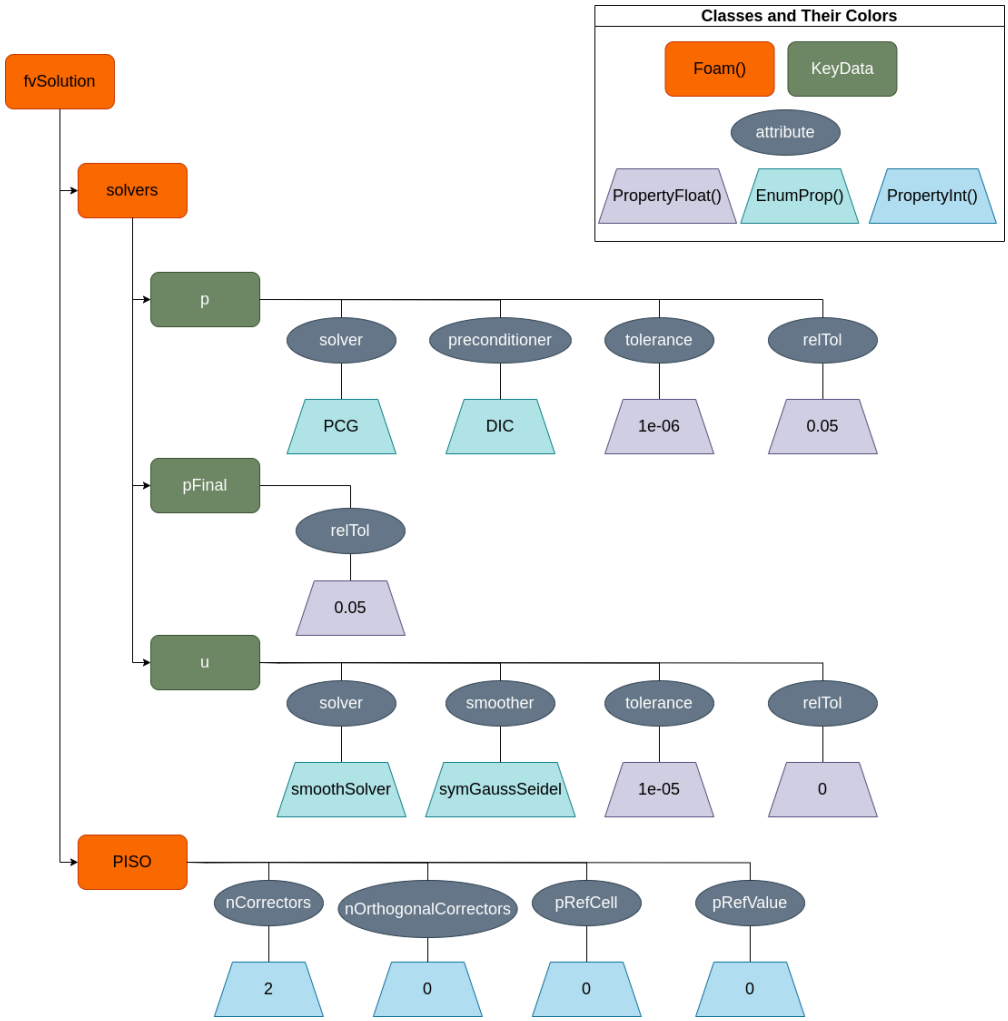

Figure 2: Tree Data structure that is generated from a sample `fvSolution` file

2.3.3 Side Panel

The side panel has the option to manipulate the node. Currently, it has operators to add and remove keys from the node. On running the 'add key' operator, the user will be presented with a window to add the name of the key and the values it will be holding. The values are displayed using an UIList class inside the operator window. Once the user is satisfied with the key and values of the entry, the key will be inserted into the dictionary node once the user confirms the dialogue window of the add key operator. The remove key operator prompts the user to enter the key to be removed, and once confirmed, the key will be deleted from the node.

2.4 PyQt GUI

Along with Blender-GUI another GUI is made in PyQt so that in the future a complete standalone GUI application can be made for OpenFOAM. Similar to Blender-GUI, it also uses nodes to represent dictionary files i.e. each dictionary is represented through a single node. A dictionary file is considered a dictionary in itself. PyQt GUI comprises of three main subsubsections: Menubar, Dock and Container Widget.

2.4.1 Menubar

Menubar consists of typical operations for a desktop application like New, Open, Save, and Close.

- The new option creates a new untitled tab to represent a new OpenFOAM dictionary file.
- The open option presents the user with a dialogue box to choose a dictionary file and opens it in a new tab. Nodes are created from the data in the file.
- The save option saves the current nodes in the OpenFOAM dictionary format.
- The close option closes the application.

2.4.2 Dock

Dock provides the main tools used to interact with GUI.

- Folder View shows currently opened folders and files.
- Scene Operations contain options which affect the scene. Currently, it only allows users to create and delete nodes. Nodes can be given custom names.
- Node Operations contains options which affect a particular node or a group of nodes. It has options for creating entries of type Integer, Float, String and Dictionary. Selected entries of the node are deleted through the delete option.

2.4.3 Container Widget

Container Widget is a custom PyQt Widget which acts as a container for nodes. It provides an interface to perform operations on nodes and hence abstracts away the implementation details. Scene operations and Node operations subsubsections in the Dock use this interface to perform the operations. It uses PyQt's Graphics View Framework, which provides a surface to store, interact and visualise 2D objects. The objects can be freely moved on the surface. Hence it is ideal for representing nodes as 2D objects and lines to show relationships among nodes. The framework also allows zooming and panning of the scene.

2.5 Python Wrappers

The Python API needs to communicate with OpenFOAM and make use of its components for tasks like parsing case files, creating new case files and performing a keyword search. The wrappers allow the API to perform these tasks. They use the 'ctypes' library of Python to execute C++ code. They are a mix of C++ and Python code where the C++ code is compiled into Shared Object(Dynamic Link Library) files which are then loaded at runtime through Python code.

2.5.1 Dictionary Reader

It is a Python package to read OpenFOAM dictionary files and convert them into Python objects. The parser of dictionary files is already present in OpenFOAM through the 'dictionary' class[4]. To use this parser, a function called 'open dictionary' is provided in the package which can open a dictionary file and create the dictionary data structure in memory. To provide easy traversal of this data structure through Python an iterator called 'DictionaryIterator' is implemented which can recursively traverse the dictionary data structure and fetch data as Python objects.
The user however need not concern themselves with these details as they are abstracted away and the functionality is provided with the method as follows :

```
read(filename : str) -> dict
```

Example usage :

```
import dictionaryWrapper as dw

data = dw.read('testcase/system/fvSchemes')
```

2.5.2 Foam Search

This utility is the same as OpenFOAM's 'foamSearch' utility which searches for a keyword in all files of a given filename in the given directory. It uses the same 'DictionaryIterator' class mentioned before to search for keywords in a dictionary file. It returns the output as a Python list of tuples of value and count. The utility is provided through a single method as follows :

```
search( dir_path : str, filename : str, key : str ) -> list
```

Example Usage :

```
from foam_search import search

dir_path = '/opt/openfoam9/tutorials'
filename = 'fvSchemes'
key = 'ddtSchemes/default'

data = search(dir_path, filename, key)

for item in data:
```

```
    value = item[0]
    count = item[1]
    print(count, value)
```

Output :

```
(venv) foamSearch  $ python search_fvSchemes.py
               8       backward
               2       CrankNicolson 0.9
               1       CrankNicolson ocCoeff { type scale ; scale linearRamp ; duration 1.0 ; value 0.9 ; }
               169     Euler
               9       localEuler
               16      none
               49      steadyState
(venv) foamSearch  $
```

Figure 3: Search Result in OpenFOAM Tutorials Folder

3 Results

This chapter shows various ways to create the following sample fvSolutions dictionary file:

```
solvers
{
    p
    {
        solver           PCG;
        preconditioner   DIC;
        tolerance        1e-06;
        relTol           0.05;
    }

    pFinal
    {
        relTol           0.00;
    }

    U
    {
        solver          smoothSolver;
        smoother        symGaussSiedel;
        tolerance       1e-05;
        relTol          0.00;
    }
}

PISO
{
    nCorrectors                 2;
    nNonOrthogonalCorrectors    0;
    pRefCell                    0;
    pRefValue                   0;
}
```

3.1 Python Script

```
import pyvnt

head = pyvnt.Foam('fvSolutions')

sl = pyvnt.Foam('solvers', parent = head)

s = pyvnt.KeyData(
    'solver',
    pyvnt.EnumProp('val1', items={'PCG', 'PBiCG', 'PBiCGStab'}, default='
   PCG')
)
pc = pyvnt.KeyData(
    'preconditioner',
    pyvnt.EnumProp('val1', items={'DIC', 'DILU', 'FDIC'}, default='DIC')
```

```
)
tol = pyvnt.KeyData(
    'tolerance',
    pyvnt.PropertyFloat('val1', minimum=0, maximum=1000, default=1e-06)
)
rt = pyvnt.KeyData(
    'relTol',
    pyvnt.PropertyFloat('val1', minimum=0, maximum=100, default=0.05)
)
p = pyvnt.Foam(
    'p',
    parent = sl,
    solver = s,
    preConditioner = pc,
    tolerance = tol,
    relTol = rt
)
relTol2 = pyvnt.KeyData(
    'relTol',
    pyvnt.PropertyFloat('val1', minimum=0, maximum=100, default=0)
)
pf = pyvnt.Foam('pFinal', parent=sl, relTol=relTol2)

sol2 = pyvnt.KeyData(
    'solver',
    pyvnt.EnumProp('val1', items={'smoothSolver'}, default='smoothSolver')
)
sm = pyvnt.KeyData(
    'smoother',
    pyvnt.EnumProp('val1', items={'symGaussSeidel', 'gaussSeidel'},
   default = 'symGaussSeidel')
)
tol2 = pyvnt.KeyData(
    'tolerance',
    pyvnt.PropertyFloat('val1', minimum=0, maximum=1000, default=1e-05)
)
relTol3 = pyvnt.KeyData('relTol', pyvnt.PropertyFloat('val1', minimum=0,
   maximum=100, default=0))

u = pyvnt.Foam(
    'U',
    parent=sl,
    solver=sol2,
    smoother=sm,
    tolerance=tol2,
    relTol=relTol3
)
ncorr = pyvnt.KeyData(
    'nCorrectors',
    pyvnt.PropertyInt('int_prop_1', minimum=0, maximum=100, default=2)
)
nnoc = pyvnt.KeyData(
    'nNonOrthogonalCorrectors',
    pyvnt.PropertyInt('int_prop_2', minimum=0, maximum=100, default=0)
```

```
)
prc = pyvnt.KeyData(
    'pRefCell',
    pyvnt.PropertyInt('int_prop_3', minimum=0, maximum=100, default=0)
)
prv = pyvnt.KeyData(
    'pRefValue',
    pyvnt.PropertyInt('int_prop_4', minimum=0, maximum=100, default=0)
)
piso = pyvnt.Foam(
    'PISO',
    parent=head,
    nCorrectors=ncorr,
    nNonOrthogonalCorrectors=nnoc,
    pRefCell=prc,
    pRefValue=prv
)

head.dispTree()
```

```
> python test.py
fvSolutions
├── solvers
│   ├── p  ->  solver: { val1 : PCG, },  preConditioner: { val1 : DIC, },  tolerance: { val1 : 1e-06, },  relTol: { val1 : 0.05, },
│   ├── pFinal  ->  relTol: { val1 : 0, },
│   └── U  ->  solver: { val1 : smoothSolver, },  smoother: { val1 : symGaussSeidel, },  tolerance: { val1 : 1e-05, },  relTol: { val1 : 0, },
└── PISO  ->  nCorrectors: { int_prop_1 : 2, },  nNonOrthogonalCorrectors: { int_prop_2 : 0, },  pRefCell: { int_prop_3 : 0, },  pRefValue: { int_prop_4 : 0,
 },
```

Figure 4: The example fvSolution file rendered in the form of a tree using Python API

3.2 Blender GUI

This section shows screenshots from the blender addon used to store the above-given fvSolution file.

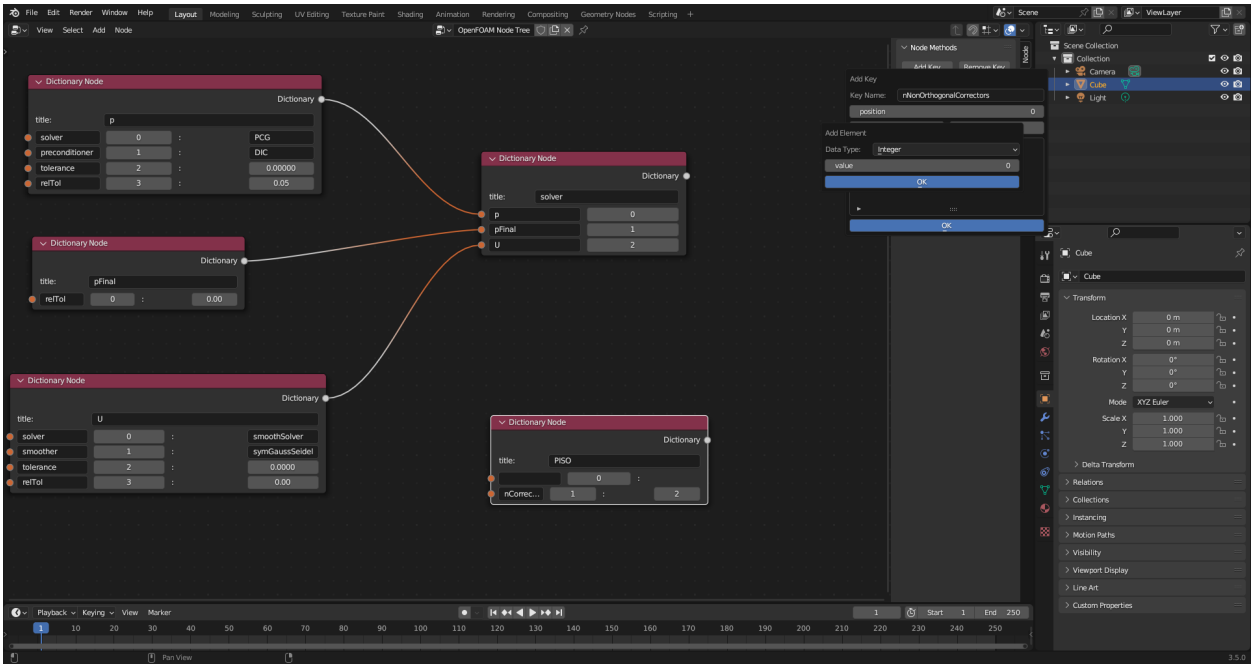

Figure 5: Set values window

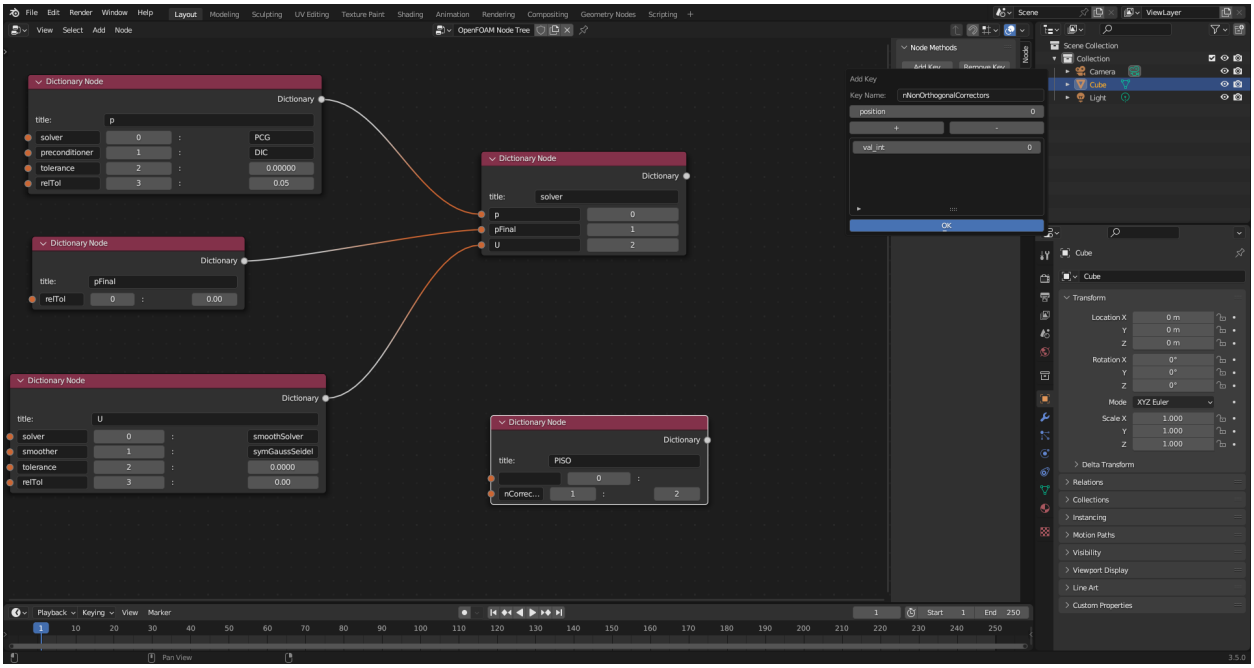

Figure 6: Set Key and value window

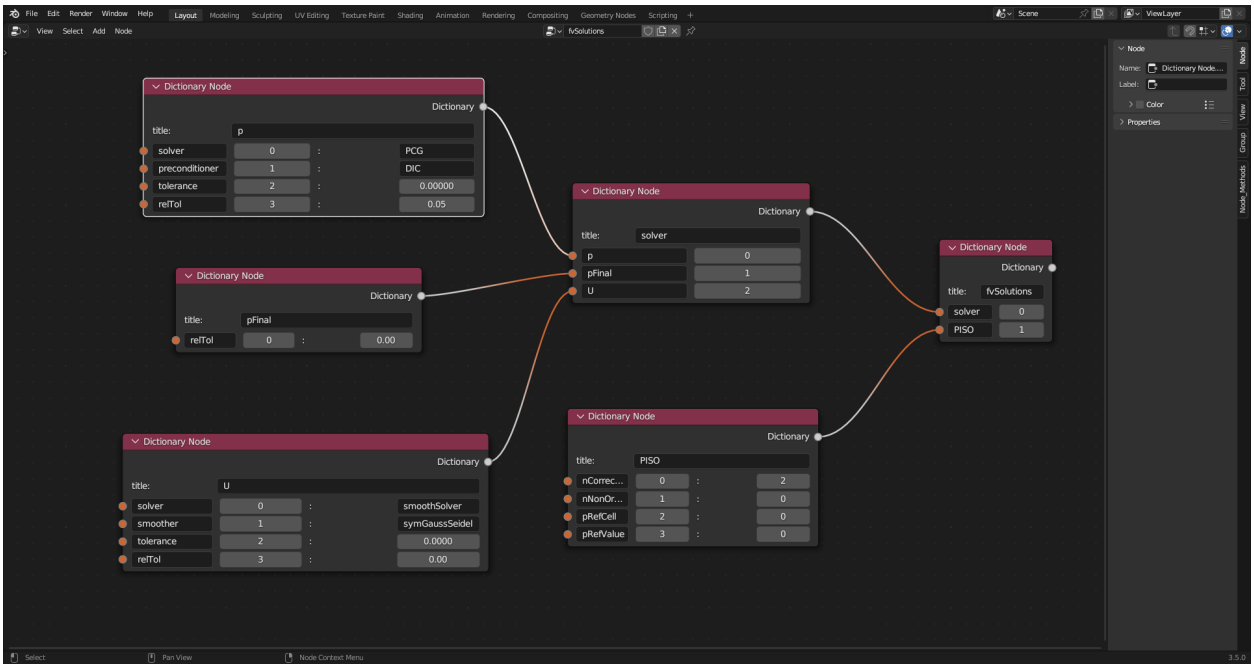

Figure 7: The entire tree displayed in blender

3.3 PyQt GUI

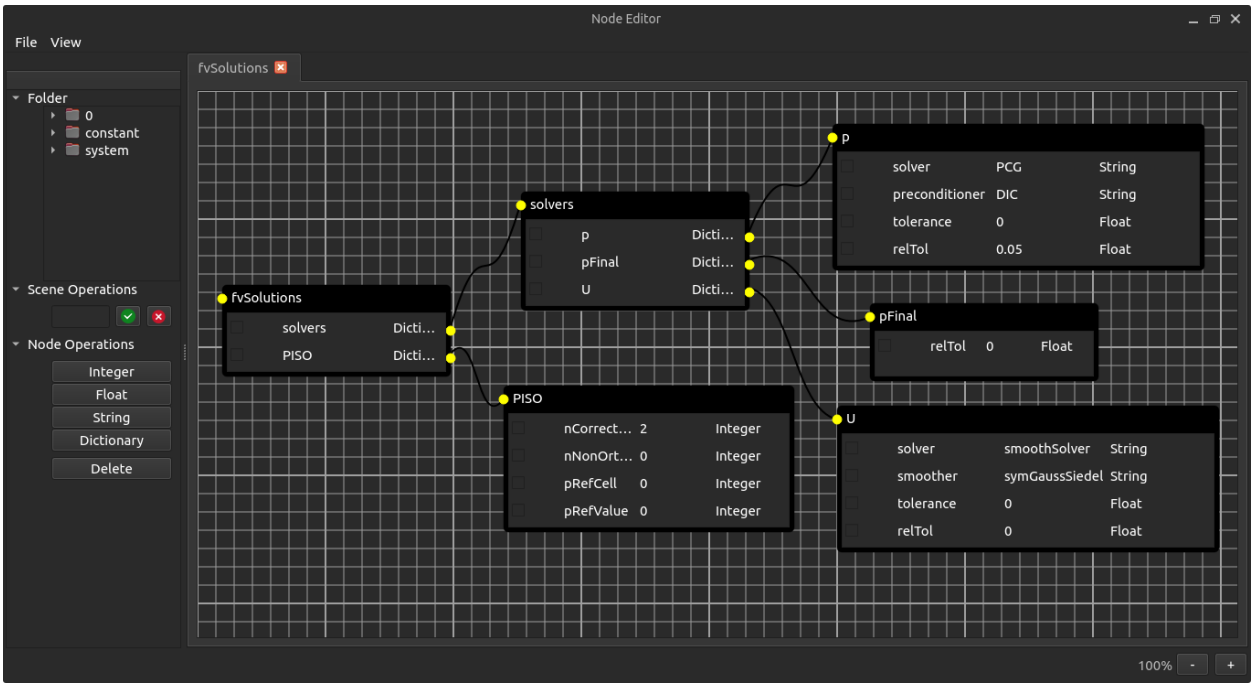

Figure 8: Nodes created from the fvSolution file

4 Conclusion

We were successful to create a GUI which can read Integer, Float, String and Dictionary entries from a file and represent it through graphical nodes. The nodes created are interactive and editable.

This GUI project is still only a prototype and more features need to be added before it can become a usable software. Some of those features can be case management, running OpenFOAM on cases, better appearance and customization.

References

[1] Hrvoje Jasak. "OpenFOAM: open source CFD in research and industry". In: *International Journal of Naval Architecture and Ocean Engineering* 1 (Dec. 2009), pp. 89–94. DOI: 10.3744/JNAOE.2009.1.2.089.

[2] Riverbank Computing. *Reference Guide PyQt Documentation v6.5.1*. https://www.riverbankcomputing.com/static/Docs/PyQt6/.

[3] AnyTree. *AnyTree 2.8.0 documentation*. https://anytree.readthedocs.io/en/latest/.

[4] The OpenFOAM Foundation. *dictionary Class Reference*. https://cpp.openfoam.org/v9/classFoam_1_1dictionary.html.