SimPhoNy-Mayavi Documentation

Release 0.3.0.dev0

SimPhoNy FP7 Collaboration

December 17, 2015

Contents

A plugin-library for the Simphony framework (http://www.simphony-project.eu/) to provide visualization support of the CUDS highlevel components.

CHAPTER 1

Repository

Simphony-mayavi is hosted on github: https://github.com/simphony/simphony-mayavi

CHAPTER 2

Requirements

- mayavi[app] >= 4.4.0
- simphony[H5IO] >= 0.2.1,<0.3

2.1 Optional requirements

To support the documentation built you need the following packages:

- sphinx >= 1.2.3
- sectiondoc commit 8a0c2be, https://github.com/enthought/sectiondoc
- trait-documenter, https://github.com/enthought/trait-documenter
- mock

Alternative running `pip install -r doc_requirements` should install the minimum necessary components for the documentation built.

CHAPTER 3

Installation

The package requires python 2.7.x, installation is based on setuptools:

```
# build and install
python setup.py install
```

or:

```
# build for in-place development
python setup.py develop
```

CHAPTER 4

Testing

To run the full test-suite run:

```
python -m unittest discover
```

Documentation

To build the documentation in the doc/build directory run:

```
python setup.py build_sphinx
```

Note:

- One can use the –help option with a setup.py command to see all available options.
- The documentation will be saved in the `./build` directory.

CHAPTER 6

Usage

After installation the user should be able to import the `mayavi` visualization plugin module by:

```
from simphony.visualisation import mayavi_tools
mayavi_tools.show(cuds)
```

Directory structure

- simphony-mayavi – Main package folder.
  - sources – Wrap CUDS objects to provide Mayavi Sources.
  - cuds – Wrap VTK Dataset objects to provide the CUDS container api.
  - core – Utility classes and tools to manipulate vtk and cuds objects.
- examples – Holds examples of loading and visualising SimPhoNy objects with simphony-mayavi.
- doc – Documentation related files: - The rst source files for the documentation

User Manual

8.1 SimPhoNy

Mayavi tools are available in the simphony library through the visualisation plug-in named `mayavi_tools`.

e.g:

```
from simphony.visualisation import mayavi_tools
```

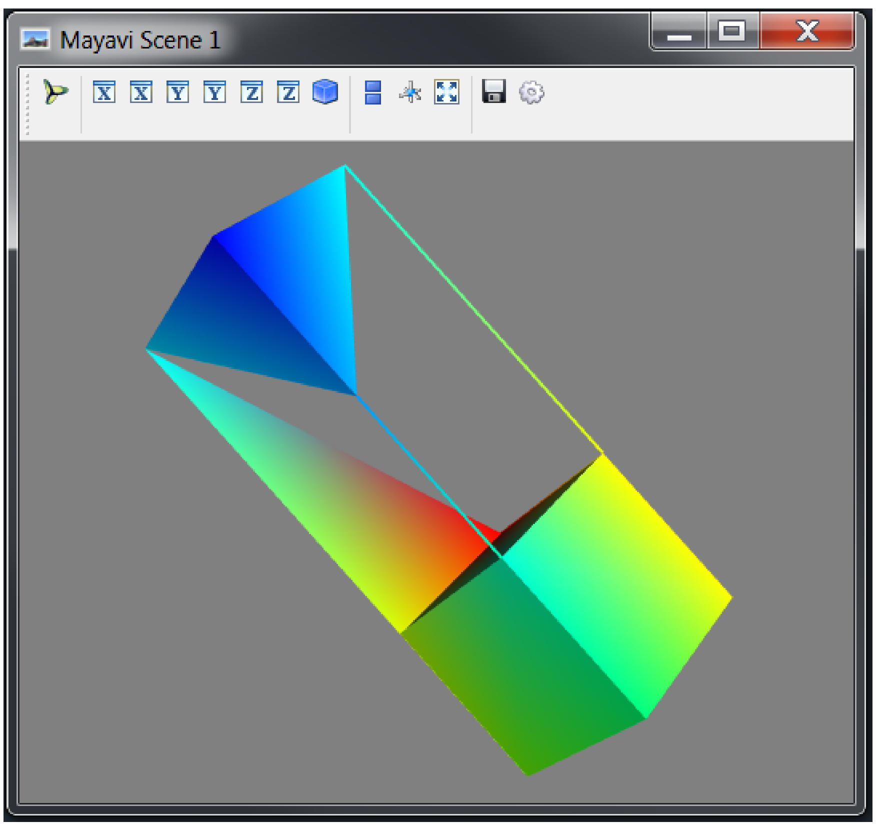
8.1.1 Visualizing CUDS

The *show()* function is available to visualise any top level CUDS container. The function will open a window containing a 3D view and a mayavi toolbar. Interaction allows the common mayavi operations.

Mesh example

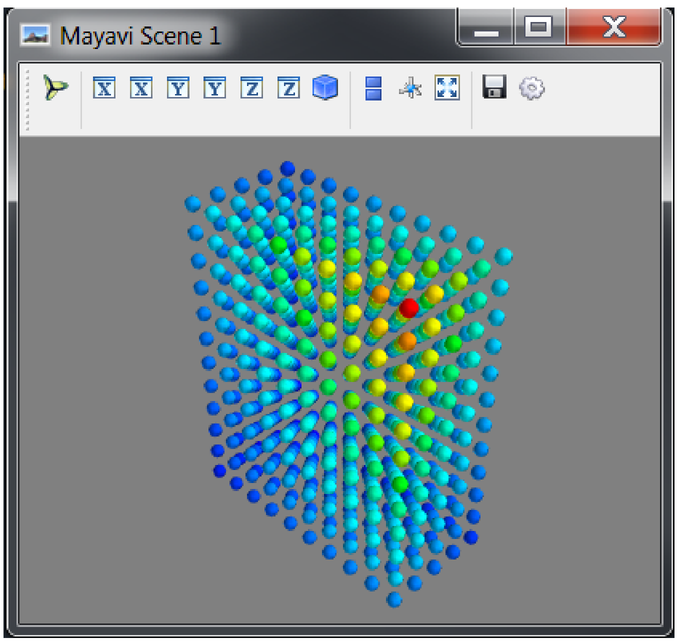
```
from numpy import array

from simphony.cuds.mesh import Mesh, Point, Cell, Edge, Face
from simphony.core.data_container import DataContainer


points = array([
    [0, 0, 0], [1, 0, 0], [0, 1, 0], [0, 0, 1],
    [2, 0, 0], [3, 0, 0], [3, 1, 0], [2, 1, 0],
    [2, 0, 1], [3, 0, 1], [3, 1, 1], [2, 1, 1]],
    'f')

cells = [
    [0, 1, 2, 3],   # tetra
    [4, 5, 6, 7, 8, 9, 10, 11]]   # hex

faces = [[2, 7, 11]]
edges = [[1, 4], [3, 8]]

mesh = Mesh('example')

# add points
point_iter = (Point(coordinates=point, data=DataContainer(TEMPERATURE=index))
              for index, point in enumerate(points))
```

```
uids = mesh.add_points(point_iter)

# add edges
edge_iter = (Edge(points=[uids[index] for index in element])
             for index, element in enumerate(edges))
edge_uids = mesh.add_edges(edge_iter)

# add faces
face_iter = (Face(points=[uids[index] for index in element])
             for index, element in enumerate(faces))
face_uids = mesh.add_faces(face_iter)

# add cells
cell_iter = (Cell(points=[uids[index] for index in element])
             for index, element in enumerate(cells))
cell_uids = mesh.add_cells(cell_iter)


if __name__ == '__main__':
    from simphony.visualisation import mayavi_tools

    # Visualise the Mesh object
    mayavi_tools.show(mesh)
```



Lattice example

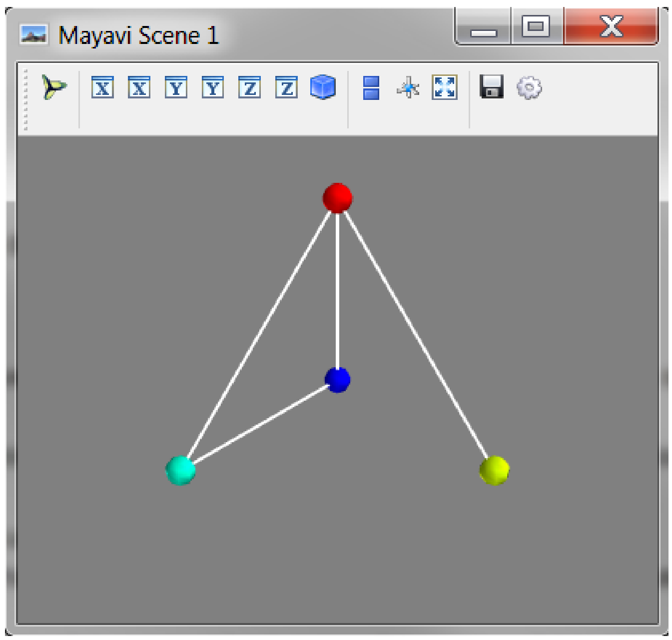
```
import numpy

from simphony.cuds.lattice import make_cubic_lattice
from simphony.core.cuba import CUBA

lattice = make_cubic_lattice('test', 0.1, (5, 10, 12))

new_nodes = []
for node in lattice.iter_nodes():
    index = numpy.array(node.index) + 1.0
    node.data[CUBA.TEMPERATURE] = numpy.prod(index)
    new_nodes.append(node)

lattice.update_nodes(new_nodes)


if __name__ == '__main__':
    from simphony.visualisation import mayavi_tools

    # Visualise the Lattice object
    mayavi_tools.show(lattice)
```



Particles example

```
from numpy import array

from simphony.cuds.particles import Particles, Particle, Bond
from simphony.core.data_container import DataContainer

points = array([[0, 0, 0], [1, 0, 0], [0, 1, 0], [0, 0, 1]], 'f')
bonds = array([[0, 1], [0, 3], [1, 3, 2]])
```

```
temperature = array([10., 20., 30., 40.])

particles = Particles('test')

# add particles
particle_iter = (Particle(coordinates=point,
                          data=DataContainer(TEMPERATURE=temperature[index]))
                 for index, point in enumerate(points))
uids = particles.add_particles(particle_iter)

# add bonds
bond_iter = (Bond(particles=[uids[index] for index in indices])
             for indices in bonds)
particles.add_bonds(bond_iter)

if __name__ == '__main__':
    from simphony.visualisation import mayavi_tools

    # Visualise the Particles object
    mayavi_tools.show(particles)
```



8.1.2 Create VTK backed CUDS

Three objects (i.e class:~. *VTKMesh*, class:~.*VTKLattice*, ~.*VTKParticles*) that wrap a VTK dataset and provide the CUDS top level container API are also available. The vtk backed objects are expected to provide memory and some speed advantages when Mayavi aided visualisation and processing is a major part of the working session. The provided examples are equivalent to the ones in section *Visualizing CUDS*.

Note: Note all CUBA keys are supported for the *data* attribute of the contained items. Please see documentation for more details.

VTK Mesh example

```
from numpy import array

from simphony.cuds.mesh import Point, Cell, Edge, Face
from simphony.core.data_container import DataContainer
from simphony.visualisation import mayavi_tools


points = array([
    [0, 0, 0], [1, 0, 0], [0, 1, 0], [0, 0, 1],
    [2, 0, 0], [3, 0, 0], [3, 1, 0], [2, 1, 0],
    [2, 0, 1], [3, 0, 1], [3, 1, 1], [2, 1, 1]],
    'f')

cells = [
    [0, 1, 2, 3],  # tetra
    [4, 5, 6, 7, 8, 9, 10, 11]]  # hex

faces = [[2, 7, 11]]
edges = [[1, 4], [3, 8]]

mesh = mayavi_tools.VTKMesh('example')

# add points
point_iter = (Point(coordinates=point, data=DataContainer(TEMPERATURE=index))
              for index, point in enumerate(points))
uids = mesh.add_points(point_iter)

# add edges
edge_iter = (Edge(points=[uids[index] for index in element])
             for index, element in enumerate(edges))
edge_uids = mesh.add_edges(edge_iter)

# add faces
face_iter = (Face(points=[uids[index] for index in element])
             for index, element in enumerate(faces))
face_uids = mesh.add_faces(face_iter)

# add cells
cell_iter = (Cell(points=[uids[index] for index in element])
             for index, element in enumerate(cells))
cell_uids = mesh.add_cells(cell_iter)


if __name__ == '__main__':

    # Visualise the Mesh object
    mayavi_tools.show(mesh)
```

VTK Lattice example

```
import numpy

from simphony.core.cuba import CUBA
from simphony.cuds.primitive_cell import PrimitiveCell
```

```
from simphony.visualisation import mayavi_tools

cubic = mayavi_tools.VTKLattice.empty(
    "test", PrimitiveCell.for_cubic_lattice(0.1),
    (5, 10, 12), (0, 0, 0))

lattice = cubic

new_nodes = []
for node in lattice.iter_nodes():
    index = numpy.array(node.index) + 1.0
    node.data[CUBA.TEMPERATURE] = numpy.prod(index)
    new_nodes.append(node)

lattice.update_nodes(new_nodes)


if __name__ == '__main__':

    # Visualise the Lattice object
    mayavi_tools.show(lattice)
```

VTK Particles example

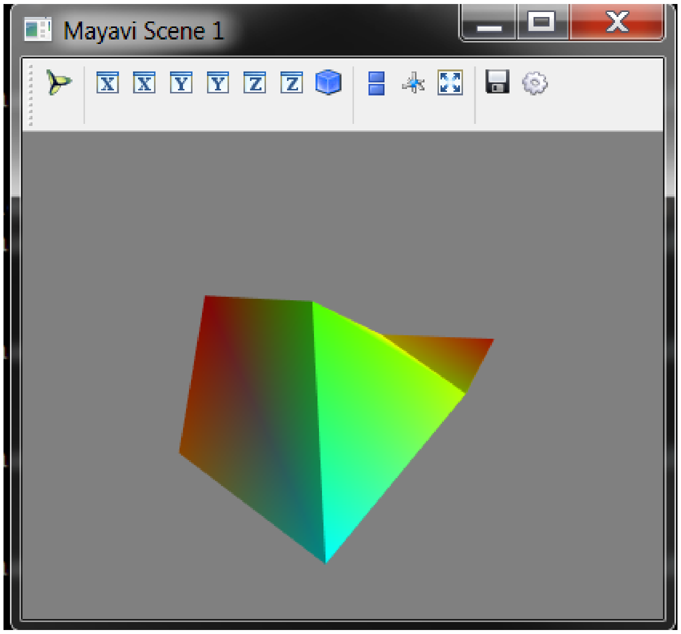
```
from numpy import array

from simphony.core.data_container import DataContainer
from simphony.cuds.particles import Particle, Bond
from simphony.visualisation import mayavi_tools

points = array([[0, 0, 0], [1, 0, 0], [0, 1, 0], [0, 0, 1]], 'f')
bonds = array([[0, 1], [0, 3], [1, 3, 2]])
temperature = array([10., 20., 30., 40.])

particles = mayavi_tools.VTKParticles('test')

# add particles
particle_iter = (Particle(coordinates=point,
                          data=DataContainer(TEMPERATURE=temperature[index]))
                 for index, point in enumerate(points))
uids = particles.add_particles(particle_iter)

# add bonds
bond_iter = (Bond(particles=[uids[index] for index in indices])
             for indices in bonds)
particles.add_bonds(bond_iter)

if __name__ == '__main__':

    # Visualise the Particles object
    mayavi_tools.show(particles)
```

8.1.3 Adapting VTK datasets

The *adapt2cuds()* function is available to wrap common VTK datsets into top level CUDS containers. The function will attempt to automatically adapt the (t)vtk Dataset into a CUDS container. When automatic conversion fails the user can always force the kind of the container to adapt into. Furthermore, the user can define the mapping of the included attribute data into corresponding CUBA keys (a common case for vtk datasets that come from vtk reader objects).

Example

```
from numpy import array, random
from tvtk.api import tvtk
from simphony.core.cuba import CUBA
from simphony.visualisation import mayavi_tools


def create_unstructured_grid(array_name='scalars'):
    points = array(
        [[0, 1.2, 0.6], [1, 0, 0], [0, 1, 0], [1, 1, 1],  # tetra
         [1, 0, -0.5], [2, 0, 0], [2, 1.5, 0], [0, 1, 0],
         [1, 0, 0], [1.5, -0.2, 1], [1.6, 1, 1.5], [1, 1, 1]], 'f')  # Hex
    cells = array(
        [4, 0, 1, 2, 3,  # tetra
         8, 4, 5, 6, 7, 8, 9, 10, 11])  # hex
    offset = array([0, 5])
    tetra_type = tvtk.Tetra().cell_type  # VTK_TETRA == 10
    hex_type = tvtk.Hexahedron().cell_type  # VTK_HEXAHEDRON == 12
    cell_types = array([tetra_type, hex_type])
    cell_array = tvtk.CellArray()
    cell_array.set_cells(2, cells)
    ug = tvtk.UnstructuredGrid(points=points)
    ug.set_cells(cell_types, offset, cell_array)
    scalars = random.random(points.shape[0])
    ug.point_data.scalars = scalars
    ug.point_data.scalars.name = array_name
    scalars = random.random((2, 1))
    ug.cell_data.scalars = scalars
    ug.cell_data.scalars.name = array_name
    return ug

# Create an example
vtk_dataset = create_unstructured_grid()

# Adapt to a mesh by converting the scalars attribute to TEMPERATURE
container = mayavi_tools.adapt2cuds(
    vtk_dataset, 'test',
    rename_arrays={'scalars': CUBA.TEMPERATURE})

if __name__ == '__main__':

    # Visualise the Lattice object
    mayavi_tools.show(container)
```



8.1.4 Loading into CUDS

The *load()* function is available to load mayavi readable files (e.g. VTK xml format) into top level CUDS containers. Using `load` the user can import inside their simulation scripts files that have been created by other simulation application and export data into one of the Mayavi supported formats.

8.2 Mayavi2

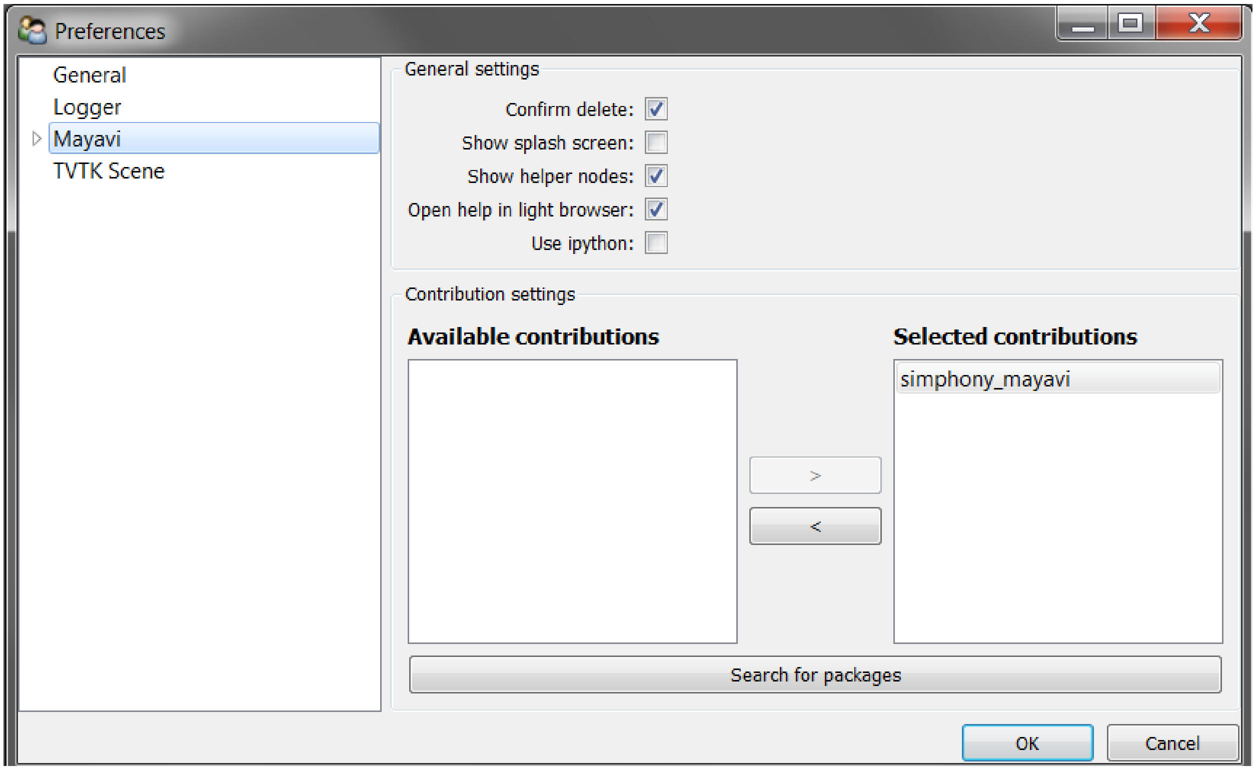
The Simphony-Mayavi library provides a plugin for Mayavi2 to easily create mayavi `Source` instances from SimPhoNy CUDS containers and files. With the provided tools one can use the SimPhoNy libraries to work inside the Mayavi2 application, as it is demonstrated in the examples.

Setup plugin

To setup the mayavi2 plugin one needs to make sure that the simpony_mayavi plugin has been selected and activated in the Mayavi2 preferences dialog.

Source from a CUDS Mesh

```
from numpy import array
from mayavi.scripts import mayavi2

from simphony.cuds.mesh import Mesh, Point, Cell, Edge, Face
from simphony.core.data_container import DataContainer


points = array([
    [0, 0, 0], [1, 0, 0], [0, 1, 0], [0, 0, 1],
    [2, 0, 0], [3, 0, 0], [3, 1, 0], [2, 1, 0],
    [2, 0, 1], [3, 0, 1], [3, 1, 1], [2, 1, 1]],
```




```
    'f')

cells = [
    [0, 1, 2, 3],  # tetra
    [4, 5, 6, 7, 8, 9, 10, 11]]  # hex

faces = [[2, 7, 11]]
edges = [[1, 4], [3, 8]]

container = Mesh('test')

# add points
point_iter = (Point(coordinates=point, data=DataContainer(TEMPERATURE=index))
              for index, point in enumerate(points))
uids = container.add_points(point_iter)

# add edges
edge_iter = (Edge(points=[uids[index] for index in element],
                  data=DataContainer(TEMPERATURE=index + 20))
             for index, element in enumerate(edges))
edge_uids = container.add_edges(edge_iter)

# add faces
face_iter = (Face(points=[uids[index] for index in element],
                  data=DataContainer(TEMPERATURE=index + 30))
             for index, element in enumerate(faces))
face_uids = container.add_faces(face_iter)

# add cells
cell_iter = (Cell(points=[uids[index] for index in element],
```

```
                    data=DataContainer(TEMPERATURE=index + 40))
              for index, element in enumerate(cells))
cell_uids = container.add_cells(cell_iter)


# Now view the data.
@mayavi2.standalone
def view():
    from mayavi.modules.surface import Surface
    from simphony_mayavi.sources.api import CUDSSource

    mayavi.new_scene()  # noqa
    src = CUDSSource(cuds=container)
    mayavi.add_source(src)  # noqa
    s = Surface()
    mayavi.add_module(s)  # noqa

if __name__ == '__main__':
    view()
```

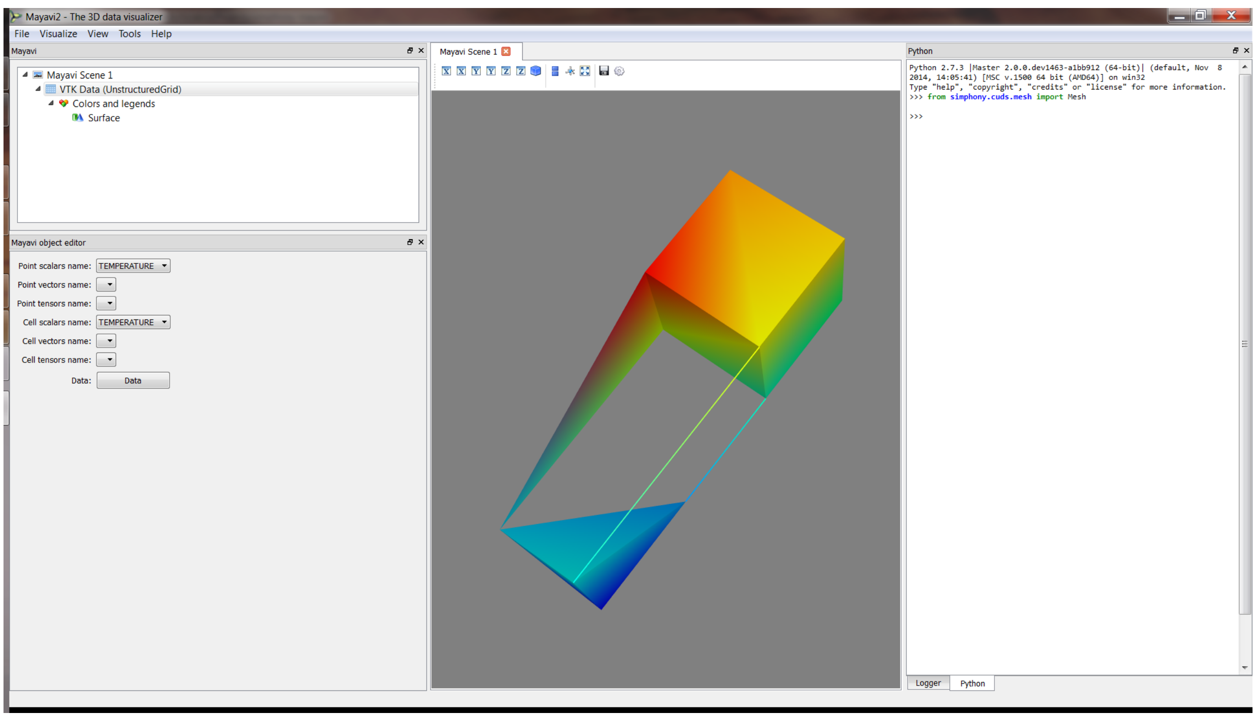


Fig. 8.1: Use the provided example to create a CUDS Mesh and visualise directly in Mayavi2.

Source from a CUDS Lattice

```
import numpy

from mayavi.scripts import mayavi2
from simphony.cuds.lattice import make_cubic_lattice
from simphony.core.cuba import CUBA

cubic = make_cubic_lattice("cubic", 0.1, (5, 10, 12))
```

```
def add_temperature(lattice):
    new_nodes = []
    for node in lattice.iter_nodes():
        index = numpy.array(node.index) + 1.0
        node.data[CUBA.TEMPERATURE] = numpy.prod(index)
        new_nodes.append(node)
    lattice.update_nodes(new_nodes)

add_temperature(cubic)


# Now view the data.
@mayavi2.standalone
def view(lattice):
    from mayavi.modules.glyph import Glyph
    from simphony_mayavi.sources.api import CUDSSource
    mayavi.new_scene()  # noqa
    src = CUDSSource(cuds=lattice)
    mayavi.add_source(src)  # noqa
    g = Glyph()
    gs = g.glyph.glyph_source
    gs.glyph_source = gs.glyph_dict['sphere_source']
    g.glyph.glyph.scale_factor = 0.02
    g.glyph.scale_mode = 'data_scaling_off'
    mayavi.add_module(g)  # noqa

if __name__ == '__main__':
    view(cubic)
```

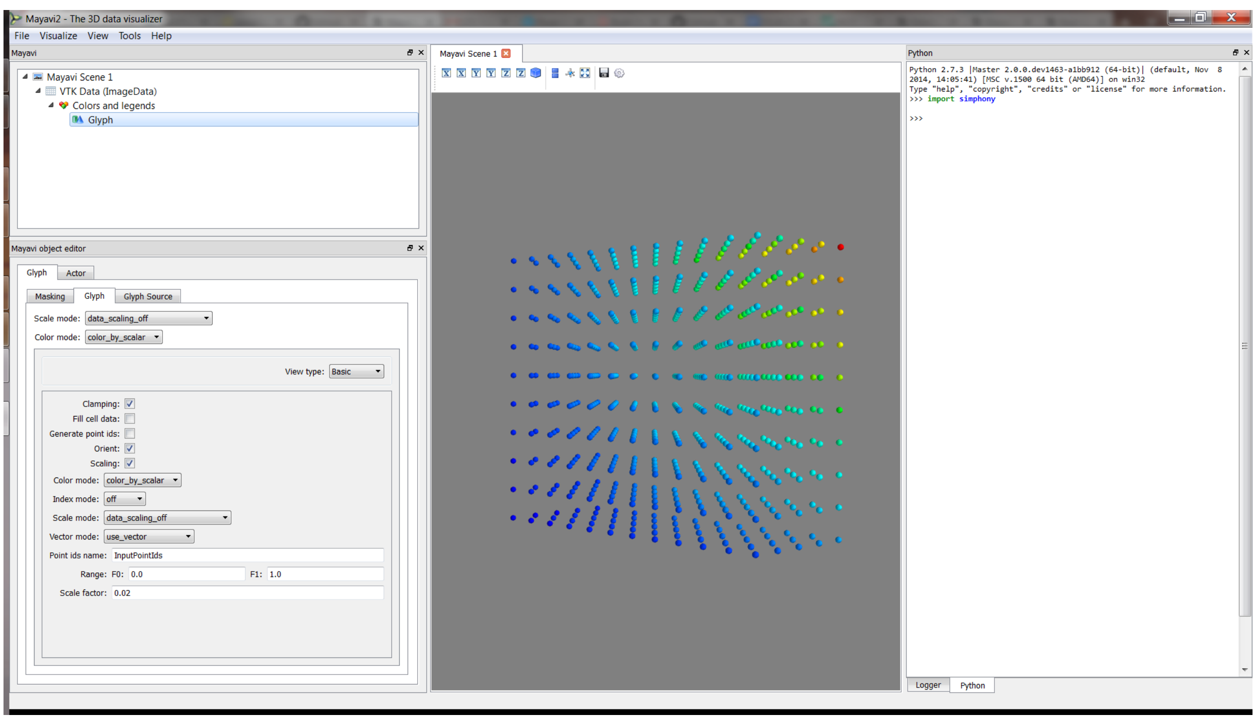


Fig. 8.2: Use the provided example to create a CUDS Lattice and visualise directly in Mayavi2.

Source for a CUDS Particles

```
from numpy import array
from mayavi.scripts import mayavi2

from simphony.cuds.particles import Particles, Particle, Bond
from simphony.core.data_container import DataContainer

points = array([[0, 0, 0], [1, 0, 0], [0, 1, 0], [0, 0, 1]], 'f')
bonds = array([[0, 1], [0, 3], [1, 3, 2]])
temperature = array([10., 20., 30., 40.])

container = Particles('test')

# add particles
particle_iter = (Particle(coordinates=point,
                          data=DataContainer(TEMPERATURE=temperature[index]))
                 for index, point in enumerate(points))
uids = container.add_particles(particle_iter)

# add bonds
bond_iter = (Bond(particles=[uids[index] for index in indices])
             for indices in bonds)
container.add_bonds(bond_iter)


# Now view the data.
@mayavi2.standalone
def view():
    from mayavi.modules.surface import Surface
    from mayavi.modules.glyph import Glyph
    from simphony_mayavi.sources.api import CUDSSource

    mayavi.new_scene()  # noqa
    src = CUDSSource(cuds=container)
    mayavi.add_source(src)  # noqa
    g = Glyph()
    gs = g.glyph.glyph_source
    gs.glyph_source = gs.glyph_dict['sphere_source']
    g.glyph.glyph.scale_factor = 0.05
    g.glyph.scale_mode = 'data_scaling_off'
    s = Surface()
    s.actor.mapper.scalar_visibility = False

    mayavi.add_module(g)  # noqa
    mayavi.add_module(s)  # noqa

if __name__ == '__main__':
    view()
```

Source from a CUDS File

```
from contextlib import closing

from mayavi.scripts import mayavi2
import numpy
```

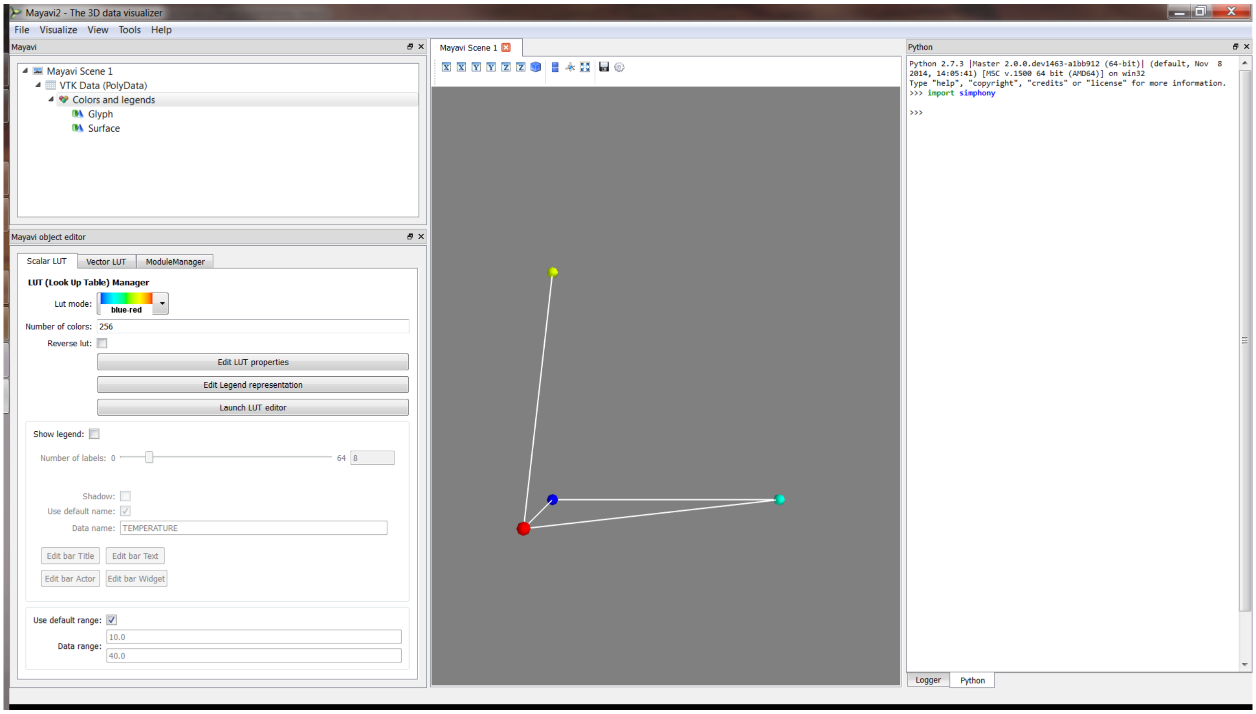


Fig. 8.3: Use the provided example to create a CUDS Particles and visualise directly in Mayavi2.

```
from numpy import array
from simphony.core.data_container import DataContainer
from simphony.core.cuba import CUBA
from simphony.cuds.particles import Particles, Particle, Bond
from simphony.cuds.lattice import (
    make_hexagonal_lattice, make_orthorhombic_lattice,
    make_body_centered_orthorhombic_lattice)
from simphony.cuds.mesh import Mesh, Point, Cell, Edge, Face
from simphony.io.h5_cuds import H5CUDS

points = array([[0, 0, 0], [1, 0, 0], [0, 1, 0], [0, 0, 1]], 'f')
bonds = array([[0, 1], [0, 3], [1, 3, 2]])
temperature = array([10., 20., 30., 40.])

# particles container
particles = Particles('particles_example')

# add particles
particle_iter = (Particle(coordinates=point,
                          data=DataContainer(TEMPERATURE=temperature[index]))
                 for index, point in enumerate(points))
uids = particles.add_particles(particle_iter)

# add bonds
bond_iter = (Bond(particles=[uids[index] for index in indices])
             for indices in bonds)
particles.add_bonds(bond_iter)


hexagonal = make_hexagonal_lattice(
```

```
    'hexagonal', 0.1, 0.1, (5, 5, 5), (5, 4, 0))
orthorhombic = make_orthorhombic_lattice(
    'orthorhombic', (0.1, 0.2, 0.3), (5, 5, 5), (5, 4, 0))
body_centered = make_body_centered_orthorhombic_lattice(
    'body_centered', (0.1, 0.2, 0.3), (5, 5, 5), (5, 10, 12))


def add_temperature(lattice):
    new_nodes = []
    for node in lattice.iter_nodes():
        index = numpy.array(node.index) + 1.0
        node.data[CUBA.TEMPERATURE] = numpy.prod(index)
        new_nodes.append(node)
    lattice.update_nodes(new_nodes)


def add_velocity(lattice):
    new_nodes = []
    for node in lattice.iter_nodes():
        node.data[CUBA.VELOCITY] = node.index
        new_nodes.append(node)
    lattice.update_nodes(new_nodes)

add_temperature(hexagonal)
add_temperature(orthorhombic)
add_temperature(body_centered)
add_velocity(hexagonal)
add_velocity(orthorhombic)
add_velocity(body_centered)

points = array([
    [0, 0, 0], [1, 0, 0], [0, 1, 0], [0, 0, 1],
    [2, 0, 0], [3, 0, 0], [3, 1, 0], [2, 1, 0],
    [2, 0, 1], [3, 0, 1], [3, 1, 1], [2, 1, 1]],
    'f')

cells = [
    [0, 1, 2, 3],   # tetra
    [4, 5, 6, 7, 8, 9, 10, 11]]   # hex

faces = [[2, 7, 11]]
edges = [[1, 4], [3, 8]]

mesh = Mesh('mesh_example')

# add points
uids = mesh.add_points((Point(coordinates=point,
                              data=DataContainer(TEMPERATURE=index))
                        for index, point in enumerate(points)))

# add edges
edge_iter = (Edge(points=[uids[index] for index in element],
                  data=DataContainer(TEMPERATURE=index + 20))
             for index, element in enumerate(edges))
edge_uids = mesh.add_edges(edge_iter)

# add faces
face_uids = mesh.add_faces((Face(points=[uids[index] for index in element],
```

```
                                data=DataContainer(TEMPERATURE=index + 30))
                           for index, element in enumerate(faces)))

# add cells
cell_uids = mesh.add_cells((Cell(points=[uids[index] for index in element],
                                data=DataContainer(TEMPERATURE=index + 40))
                           for index, element in enumerate(cells)))


# save the data into cuds.
with closing(H5CUDS.open('example.cuds', 'w')) as handle:
    handle.add_dataset(mesh)
    handle.add_dataset(particles)
    handle.add_dataset(hexagonal)
    handle.add_dataset(orthorhombic)
    handle.add_dataset(body_centered)


# Now view the data.
@mayavi2.standalone
def view():
    mayavi.new_scene()   # noqa

if __name__ == '__main__':
    view()
```

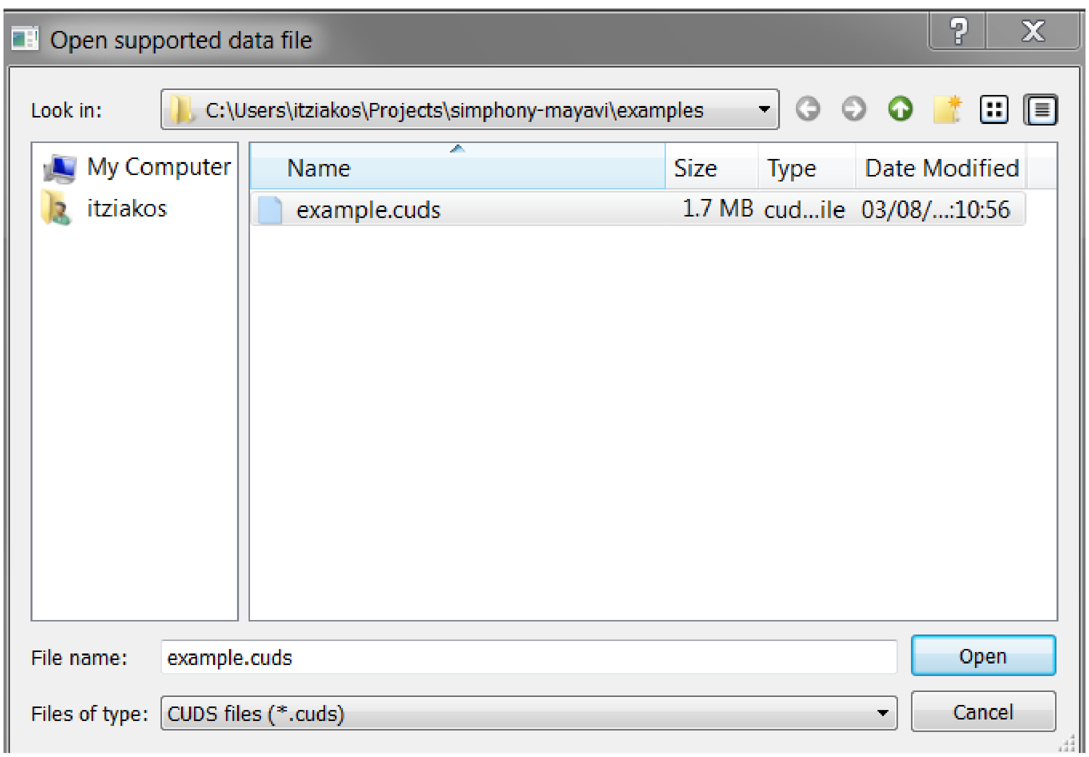


Fig. 8.4: Cuds files are supported in the `Open File..` dialog. After running the provided example load the `example.cuds` file into Mayavi2.

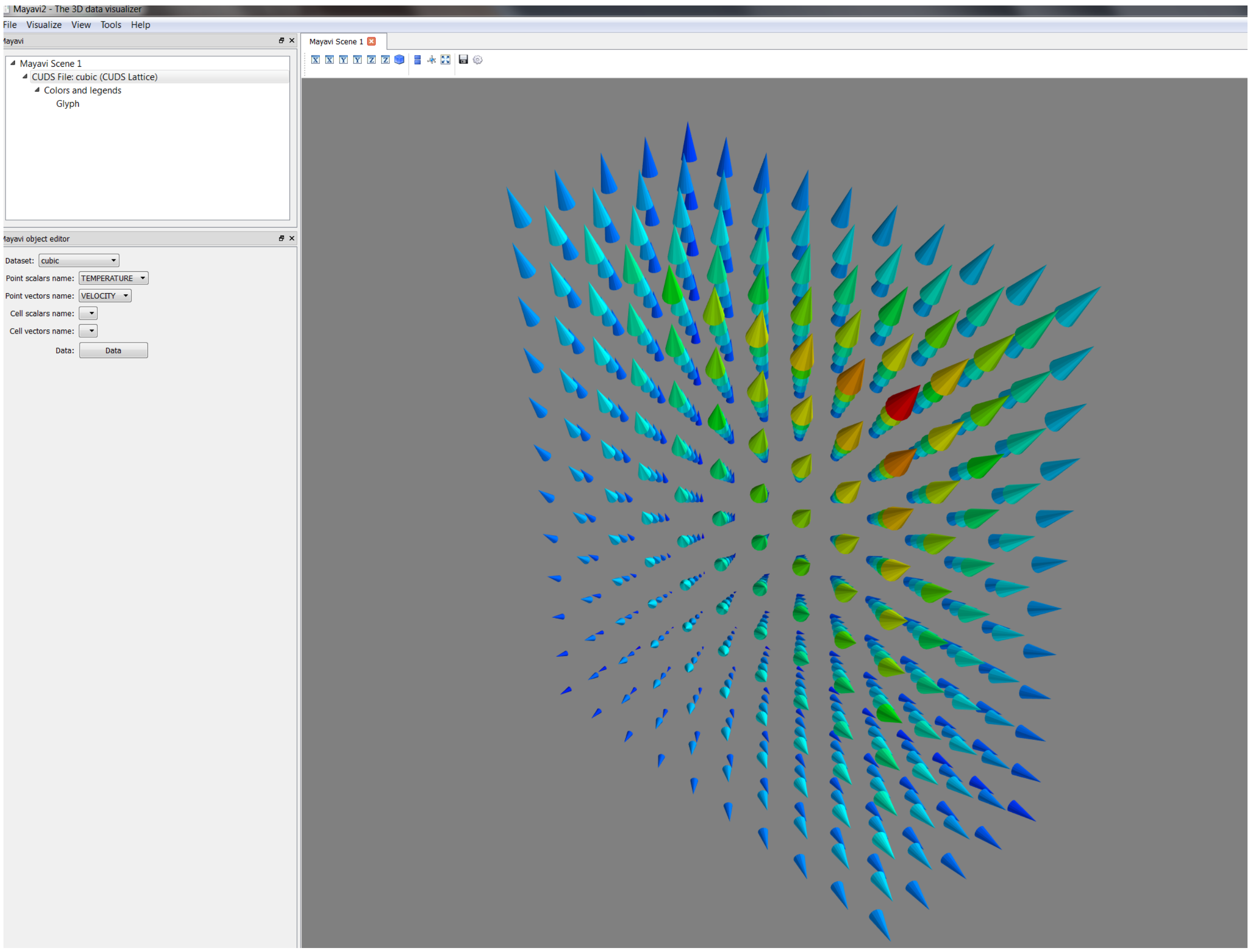


Fig. 8.5: When loaded a CUDSFile is converted into a Mayavi Source and the user can add normal Mayavi modules to visualise the currently selected CUDS container from the available containers in the file.

In the example we load the container named `cubic` and attach the Glyph module to draw a cone at each point to visualise `TEMPERATURE` and `VELOCITY` in the Mayavi Scene.

CHAPTER 9

API Reference

9.1 Plugin module

This module `simphony_mayavi.plugin` provides a set of tools to visualize CUDS objects. The tools are also available as a visualisation plug-in to the simphony library.

`simphony_mayavi.show.`**`show`**(*cuds*)

Show the cuds objects using the default visualisation.

Parameters **`cuds`** – A top level cuds object (e.g. a mesh). The method will detect the type of object and create the appropriate visualisation.

`simphony_mayavi.snapshot.`**`snapshot`**(*cuds*, *filename*)

Shave a snapshot of the cuds object using the default visualisation.

Parameters

- **`cuds`** – A top level cuds object (e.g. a mesh). The method will detect the type of object and create the appropriate visualisation.
- **`filename`** (*string*) – The filename to use for the output file.

`simphony_mayavi.adapt2cuds.`**`adapt2cuds`**(*data_set*, *name='CUDS container'*, *kind=None*, *rename_arrays=None*)

Adapt a TVTK dataset to a CUDS container.

Parameters

- **`data_set`** (*tvtk.Dataset*) – The dataset to import and wrap into CUDS container.
- **`name`** (*string*) – The name of the CUDS container. Default is 'CUDS container'.
- **`kind`** – The kind {'mesh', 'lattice', 'particles'} of the container to return. Default is None, where the function will use some heuristics to infer the most appropriate type of CUDS container to return
- **`rename_array`** (*dict*) – Dictionary mapping the array names used in the dataset object to their related CUBA keywords that will be used in the returned CUDS container.

Note: When set a shallow copy of the input data_set is created and used by the related vtk -> cuds wrapper.

Raises

- **ValueError** – When `kind` is not a valid CUDS container type.

- **TypeError** – When it is not possible to wrap the provided data_set.

simphony_mayavi.load.**load**(*filename*, *name=None*, *kind=None*, *rename_arrays=None*)
Load the file data into a CUDS container.

Parameters

- **filename** (*string*) – The file name of the file to load.
- **name** (*string*) – The name of the returned CUDS container. Default is 'CUDS container'.
- **kind** – The kind {'mesh', 'lattice', 'particles'} of the container to return. Default is None, where the function will use some heuristics to infer the most appropriate type of CUDS container to return (using adapt2cuds).
- **rename_array** (*dict*) – Dictionary mapping the array names used in the dataset object to their related CUBA keywords that will be used in the returned CUDS container.

Note: Only CUBA keywords are supported for array names so use this option to provide a translation mapping to the CUBA keys.

9.2 Sources module

A module containing objects that wrap CUDS objects and files to Mayavi compatible sources. Please use the simphony_mayavi.sources.api module to access the provided tools.

Classes

CUDSSource	A mayavi source of a SimPhoNy CUDS container.
CUDSFileSource	A mayavi source of a SimPhoNy CUDS File.

9.2.1 Description

class simphony_mayavi.sources.cuds_source.**CUDSSource**
Bases: mayavi.sources.vtk_data_source.VTKDataSource

A mayavi source of a SimPhoNy CUDS container.

cuds = Property(depends_on='_cuds')
The CUDS container

output_info = PipelineInfo(datasets=['image_data', 'poly_data', 'unstructured_grid'], attribute_types=['any'], attribu
Output information for the processing pipeline.

class simphony_mayavi.sources.cuds_file_source.**CUDSFileSource**
Bases: *simphony_mayavi.sources.cuds_source.CUDSSource*

A mayavi source of a SimPhoNy CUDS File.

dataset = DEnum(values_name='datasets')
The name of the CUDS container that is currently loaded.

datasets = ListStr
The names of the contained datasets.

file_path = Instance(FilePath, '', desc='the current file name')
The file path of the cuds file to read.

initialize (*filename*)
Initialise the CUDS file source.

start ()

update ()

9.3 Cuds module

A module containing tvtk dataset wrappers to simphony CUDS containers.

Classes

VTKParticles(name[, data, data_set, mappings])	Constructor.
VTKMesh(name[, data, data_set, mappings])	Constructor.
VTKLattice(name, primitive_cell, data_set[, ...])	Constructor.

9.3.1 Description

class simphony_mayavi.cuds.vtk_particles.**VTKParticles** (*name*, *data=None*, *data_set=None*, *mappings=None*)

Bases: simphony.cuds.abc_particles.ABCParticles

Constructor.

Parameters

- **name** (*string*) – The name of the container.
- **data** (*DataContainer*) – The data attribute to attach to the container. Default is None.
- **data_set** (*tvtk.DataSet*) – The dataset to wrap in the CUDS api. Default is None which will create a tvtk.PolyData
- **mappings** (*dict*) – A dictionary of mappings for the particle2index, index2particle, bond2index and bond2element. Should be provided if the particles and bonds described in data_set are already assigned uids. Default is None and will result in the uid <-> index mappings being generated at construction.

add_bonds (*iterable*)
Adds a set of bonds to the container.

Also like with particles, if any bond has a defined uid, it won't add the bond if a bond with the same uid already exists, and if the bond has no uid the particle container will generate an uid. If the user wants to replace an existing bond in the container there is an 'update_bonds' method for that purpose.

iterable [iterable of Bond objects] the new bond that will be included in the container.

Returns

uuid [list of uuid.UUID] The uuids of the added bonds.

Raises **ValueError** – when there is a bond with an uuid that already exists in the container.

Examples

Add a set of bonds to a Particles container.

```
>>> bonds_list = [Bond(), Bond()]
>>> particles = Particles(name="foo")
>>> particles.add_bonds(bonds_list)
```

add_particles(*iterable*)
Adds a set of particles from the provided iterable to the container.

If any particle have no uids, the container will generate a new uids for it. If the particle has already an uids, it won't add the particle if a particle with the same uid already exists. If the user wants to replace an existing particle in the container there is an 'update_particles' method for that purpose.

iterable [iterable of Particle objects] the new set of particles that will be included in the container.

Returns

uids [list of uuid.UUID] The uids of the added particles.

Raises **ValueError** – when there is a particle with an uids that already exists in the container.

Examples

Add a set of particles to a Particles container.

```
>>> particle_list = [Particle(), Particle()]
>>> particles = Particles(name="foo")
>>> uids = particles.add_particles(particle_list)
```

bond2index = None
The mapping from uid to bond index

count_of(*item_type*)
Return the count of item_type in the container.

Parameters **item_type** (*CUDSItem*) – The CUDSItem enum of the type of the items to return the count of.

Returns **count** (*int*) – The number of items of item_type in the container.

Raises **ValueError** – If the type of the item is not supported in the current container.

data
Easy access to the vtk CellData structure

data_set = None
The vtk.PolyData dataset

classmethod from_dataset(*name*, *data_set*, *data=None*)
Wrap a plain dataset into a new VTKParticles.

The constructor makes some sanity checks to make sure that the tvtk.DataSet is compatible and all the information can be properly used.

Parameters

- **name** (*str*) – The name of the container.
- **data_set** (*tvtk.DataSet*) – The dataset to wrap in the CUDS api. Default is None which will create a tvtk.PolyData
- **data** (*DataContainer*) – The data attribute to attach to the container. Default is None.

Raises TypeError – When the sanity checks fail.

classmethod from_particles(*particles*)
Create a new VTKParticles copy from a CUDS particles instance.

get_bond(*uid*)
Returns a copy of the bond with the 'bond_id' id.

Parameters uid (*uuid.UUID*) – the uid of the bond

Raises KeyError – when the bond is not in the container.

Returns bond (*Bond*) – A copy of the internally stored bond info.

get_particle(*uid*)
Returns a copy of the particle with the 'particle_id' id.

Parameters uid (*uuid.UUID*) – the uid of the particle

Raises KeyError – when the particle is not in the container.

Returns particle (*Particle*) – A copy of the internally stored particle info.

has_bond(*uid*)
Checks if a bond with the given uid already exists in the container.

has_particle(*uid*)
Checks if a particle with the given uid already exists in the container.

index2bond = None
The reverse mapping from index to bond uid

index2particle = None
The reverse mapping from index to point uid

is_connected(*bond*)
Test if the connectivity described in bonds is valid i.e. the particles are part of the container

Parameters bond (*Bond*) –

Returns valid (*bool*)

iter_bonds(*uids=None*)
Generator method for iterating over the bonds of the container.

It can receive any kind of sequence of bond ids to iterate over those concrete bond. If nothing is passed as parameter, it will iterate over all the bonds.

uids [iterable of uuid.UUID, optional] sequence containing the id's of the bond that will be iterated. When the uids are provided, then the bonds are returned in the same order the uids are returned by the iterable. If uids is None, then all bonds are returned by the iterable and there is no restriction on the order that they are returned.

Yields bond (*Bond*) – The next Bond item

Raises KeyError – if any of the ids passed as parameters are not in the container.

Examples

It can be used with a sequence as parameter or without it:

```
>>> particles = Particles(name="foo")
>>> ...
>>> for bond in particles.iter_bonds([id1, id2, id3]):
        ...   #do stuff
```

```
>>> for bond in particles.iter_bond():
        ...   #do stuff; it will iterate over all the bond
```

iter_particles(*uids=None*)
Generator method for iterating over the particles of the container.

It can receive any kind of sequence of particle uids to iterate over those concrete particles. If nothing is passed as parameter, it will iterate over all the particles.

uids [iterable of uuid.UUID, optional] sequence containing the uids of the particles that will be iterated. When the uids are provided, then the particles are returned in the same order the uids are returned by the iterable. If uids is None, then all particles are returned by the iterable and there is no restriction on the order that they are returned.

Yields **particle** (*Particle*) – The Particle item.

Raises **KeyError** – if any of the ids passed as parameters are not in the container.

Examples

It can be used with a sequence as parameter or without it:

```
>>> particles = Particles(name="foo")
>>> ...
>>> for particle in particles.iter_particles([uid1, uid2, uid3]):
    ...   #do stuff
>>> for particle in particles.iter_particles():
    ...   #do stuff
```

particle2index = None
The mapping from uid to point index

remove_bonds(*uids*)
Remove the bonds with the provided uids.

The uids passed as parameter should exists in the container. If any uid doesn't exist, an exception will be raised.

uids [iterable of uuid.UUID] the uids of the bond to be removed.

Raises **KeyError** – If any bond doesn't exist.

Examples

Having a set of uids of existing bonds, pass it to the method.

```
>>> particles = Particles(name="foo")
>>> ...
>>> particles.remove_bonds([uid1, uid2])
```

remove_particles (*uids*)
Remove the particles with the provided uids from the container.

The uids inside the iterable should exists in the container. Otherwise an exception will be raised.

uids [iterable of uuid.UUID] the uids of the particles to be removed.

Raises KeyError – If any particle doesn't exist.

Examples

Having a set of uids of existing particles, pass it to the method.

```
>>> particles = Particles(name="foo")
>>> ...
>>> particles.remove_particles([uid1, uid2])
```

supported_cuba = None
The currently supported and stored CUBA keywords.

update_bonds (*iterable*)
Updates a set of bonds from the provided iterable.

Takes the uids of the bonds and searches inside the container for those bond. If the bonds exists, they are replaced in the container. If any bond doesn't exist, it will raise an exception.

iterable [iterable of Bond objects] the bonds that will be replaced.

Raises ValueError – If any bond doesn't exist.

Examples

Given a set of Bond objects that already exists in the container (taken with the 'get_bond' method for example) just call the function passing the set of Bond as parameter.

```
>>> particles = Particles(name="foo")
>>> ...
>>> bond1 = particles.get_bond(uid1)
>>> bond2 = particles.get_bond(uid2)
>>> ... #do whatever you want with the bonds
>>> particles.update_bonds([bond1, bond2])
```

update_particles (*iterable*)
Updates a set of particles from the provided iterable.

Takes the uids of the particles and searches inside the container for those particles. If the particles exists, they are replaced in the container. If any particle doesn't exist, it will raise an exception.

iterable [iterable of Particle objects] the particles that will be replaced.

Raises ValueError – If any particle inside the iterable does not exist.

Examples

Given a set of Particle objects that already exists in the container (taken with the 'get_particle' method for example), just call the function passing the Particle items as parameter.

```
>>> part_container = Particles(name="foo")
>>> ... #do whatever you want with the particles
>>> part_container.update_particles([part1, part2])
```

class simphony_mayavi.cuds.vtk_mesh.**VTKMesh**(*name*, *data=None*, *data_set=None*, *mappings=None*)

Bases: simphony.cuds.abc_mesh.ABCMesh

Constructor.

Parameters

- **name** (*string*) – The name of the container
- **data** (*DataContainer*) – The data attribute to attach to the container. Default is None.
- **data_set** (*tvtk.DataSet*) – The dataset to wrap in the CUDS api. Default is None which will create a tvtk.UnstructuredGrid.
- **mappings** (*dict*) – A dictionary of mappings for the point2index, index2point, element2index and index2element. Should be provided if the points and elements described in data_set are already assigned uids. Default is None and will result in the uid <-> index mappings being generated at construction.

add_cells(*cells*)

Adds a set of new cells to the mesh.

cells [iterable of Cell] Cell to be added to the mesh

Raises ValueError – If other cell with a duplicated uid was already in the mesh

add_edges(*edges*)

Adds a set of new edges to the mesh.

edges [iterable of Edge] Edge to be added to the mesh

Raises ValueError – If other edge with a duplicated uid was already in the mesh

add_faces(*faces*)

Adds a set of new faces to the mesh.

faces [iterable of Face] Face to be added to the mesh

Raises ValueError – If other face with a duplicated uid was already in the mesh

add_points(*points*)

Adds a set of new points to the mesh.

points [iterable of Point] Points to be added to the mesh

Raises ValueError – If other point with a duplicated uid was already in the mesh.

count_of(*item_type*)

Return the count of item_type in the container.

Parameters item_type (*CUDSItem*) – The CUDSItem enum of the type of the items to return the count of.

Returns count (*int*) – The number of items of item_type in the container.

Raises ValueError – If the type of the item is not supported in the current container.

data
Easy access to the vtk PointData structure

data_set = None
The vtk.PolyData dataset

element2index = None
The mapping from uid to bond index

element_data = None
Easy access to the vtk CellData structure

classmethod **from_dataset** (*name*, *data_set*, *data=None*)
Wrap a plain dataset into a new VTKMesh.

The constructor makes some sanity checks to make sure that the tvtk.DataSet is compatible and all the information can be properly used.

Parameters

- **name** (*string*) – The name of the container
- **data_set** (*tvtk.DataSet*) – The dataset to wrap in the CUDS api. Default is None which will create a tvtk.UnstructuredGrid.
- **data** (*DataContainer*) – The data attribute to attach to the container. Default is None.

Raises **TypeError** – When the sanity checks fail.

classmethod **from_mesh** (*mesh*)
Create a new VTKMesh copy from a CUDS mesh instance.

get_cell (*uid*)
Returns a cell with a given uid.

Returns the cell stored in the mesh identified by uid. If such a cell does not exists an exception is raised.

Parameters **uid** (*uuid.UUID*) – uid of the desired cell.

Returns **cell** (*Cell*) – Cell identified by uid

Raises

- **KeyError** – If the cell identified by uuid was not found
- **TypeError** – When `uid` is not uuid.UUID

get_edge (*uid*)
Returns an edge with a given uid.

Returns the edge stored in the mesh identified by uid. If such edge do not exists an exception is raised.

Parameters **uid** (*uuid.UUID*) – uid of the desired edge.

Returns **edge** (*Edge*) – Edge identified by uid

Raises

- **KeyError** – If the edge identified by uid was not found
- **TypeError** – When `uid` is not uuid.UUID

get_face (*uid*)
Returns a face with a given uid.

Returns the face stored in the mesh identified by uid. If such a face does not exists an exception is raised.

Parameters **uid** (*uuid.UUID*) – uid of the desired face.

Returns **face** (*Face*) – Face identified by uid

Raises

- **KeyError** – If the face identified by uid was not found
- **TypeError** – When `uid` is not uuid.UUID

get_point(*uid*)

Returns a point with a given uid.

Returns the point stored in the mesh identified by uid. If such point do not exists an exception is raised.

Parameters **uid** (*uuid.UUID*) – uid of the desired point.

Returns **point** (*Point*) – Mesh point identified by uuid

Raises

- **KeyError** – If the point identified by uid was not found
- **TypeError** – When `uid` is not uuid.UUID

has_cells()

Check if the mesh has cells

Returns **result** (*bool*) – True of there are cells inside the mesh, False otherwise

has_edges()

Check if the mesh has edges

Returns **result** (*bool*) – True of there are edges inside the mesh, False otherwise

has_faces()

Check if the mesh has faces

Returns **result** (*bool*) – True of there are faces inside the mesh, False otherwise

index2element = None

The reverse mapping from index to bond uid

index2point = None

The reverse mapping from index to point uid

iter_cells(*uids=None*)

Returns an iterator over cells.

uids [iterable of uuid.UUID or None] When the uids are provided, then the cells are returned in the same order the uids are returned by the iterable. If uids is None, then all cells are returned by the iterable and there is no restriction on the order that they are returned.

Yields **cell** (*Cell*)

iter_edges(*uids=None*)

Returns an iterator over edges.

uids [iterable of uuid.UUID or None] When the uids are provided, then the edges are returned in the same order the uids are returned by the iterable. If uids is None, then all edges are returned by the iterable and there is no restriction on the order that they are returned.

Yields **edge** (*Edge*)

iter_faces(*uids=None*)

Returns an iterator over faces.

uids [iterable of uuid.UUID or None] When the uids are provided, then the faces are returned in the same order the uids are returned by the iterable. If uids is None, then all faces are returned by the iterable and there is no restriction on the order that they are returned.

Yields **face** (*Face*)

`iter_points`(*uids=None*)
Returns an iterator over points.

uids [iterable of uuid.UUID or None] When the uids are provided, then the points are returned in the same order the uids are returned by the iterable. If uids is None, then all points are returned by the iterable and there is no restriction on the order that they are returned.

Yields **point** (*Point*)

`point2index` = **None**
The mapping from uid to point index

`supported_cuba` = **None**
The currently supported and stored CUBA keywords.

`update_cells`(*cells*)
Updates the information of a set of cells.

Gets the mesh cell identified by the same uid as the provided cell and updates its information with the one provided with the new cell.

cells [iterable of Cell] Cell to be updated

Raises **ValueError** – If the any cell was not found in the mesh

`update_edges`(*edges*)
Updates the information of a set of edges.

Gets the mesh edge identified by the same uid as the provided edge and updates its information with the one provided with the new edge.

edges [iterable of Edge] Edge to be updated

Raises **ValueError** – If the any edge was not found in the mesh

`update_faces`(*faces*)
Updates the information of a set of faces.

Gets the mesh face identified by the same uid as the provided face and updates its information with the one provided with the new face.

faces [iterable of Face] Face to be updated

Raises **ValueError** – If the any face was not found in the mesh

`update_points`(*points*)
Updates the information of a set of points.

Gets the mesh point identified by the same uid as the provided point and updates its information with the one provided with the new point.

points [iterable of Point] Point to be updated

Raises **ValueError** – If the any point was not found in the mesh

class `simphony_mayavi.cuds.vtk_lattice.`**`VTKLattice`**(*name*, *primitive_cell*, *data_set*, *data=None*)

Bases: `simphony.cuds.abc_lattice.ABCLattice`

Constructor.

Parameters

- **`name`** (*string*) – The name of the container.
- **`primitive_cell`** (*PrimitiveCell*) – primitive cell specifying the 3D Bravais lattice
- **`data_set`** (*tvtk.DataSet*) – The dataset to wrap in the CUDS api. If it is a tvtk.PolyData, the points are assumed to be arranged in C-contiguous order so that the first point is the origin and the last point is furthest away from the origin
- **`data`** (*DataContainer*) – The data attribute to attach to the container. Default is None.

`count_of`(*item_type*)

Return the count of item_type in the container.

Parameters **`item_type`** (*CUDSItem*) – The CUDSItem enum of the type of the items to return the count of.

Returns **count** (*int*) – The number of items of item_type in the container.

Raises **ValueError** – If the type of the item is not supported in the current container.

`data`

The container data

classmethod **`empty`**(*name*, *primitive_cell*, *size*, *origin*, *data=None*)

Create a new empty Lattice.

Parameters

- **`name`** (*string*) – The name of the container.
- **`primitive_cell`** (*PrimitiveCell*) – Primitive cell specifying the 3D Bravais lattice
- **`size`** (*tuple*) – lattice dimensions (nx, ny, nz)
- **`origin`** (*tuple*) – lattice origin (x, y, z)
- **`data`** (*DataContainer*) – The data attribute to attach to the container. Default is None.

Returns **lattice** (*VTKLattice*)

classmethod **`from_dataset`**(*name*, *data_set*, *data=None*)

Create a new Lattice and try to guess the `primitive_cell`

Parameters

- **`name`** (*str*) –
- **`data_set`** (*tvtk.ImageData or tvtk.PolyData*) – The dataset to wrap in the CUDS api. If it is a PolyData, the points are assumed to be arranged in C-contiguous order
- **`data`** (*DataContainer*) – The data attribute to attach to the container. Default is None.

Returns **lattice** (*VTKLattice*)

Raises **TypeError** – If data_set is not either tvtk.ImageData or tvtk.PolyData

IndexError: If the lattice nodes are not arranged in C-contiguous order

classmethod `from_lattice` (*lattice*)
Create a new Lattice from the provided one.

Parameter

lattice : simphony.cuds.lattice.Lattice

Returns **lattice** (*VTKLattice*)

Raises

ValueError

- if bravais_lattice attribute of the primitive cell indicates a cubic/tetragonal/orthorhombic lattice but the primitive vectors are inconsistent with this attribute
- if bravais_lattice is not a member of BravaisLattice

`get_coordinate` (*ind*)
Get coordinate of the given index coordinate.

ind [int[3]] node index coordinate

Returns

coordinates : float[3]

`get_node` (*index*)
Get the lattice node corresponding to the given index.

index [int[3]] node index coordinate

Returns **node** (*LatticeNode*)

`iter_nodes` (*indices=None*)
Get an iterator over the LatticeNodes described by the indices.

indices [iterable set of int[3], optional] When indices (i.e. node index coordinates) are provided, then nodes are returned in the same order of the provided indices. If indices is None, there is no restriction on the order of the returned nodes.

Returns

iterator: An iterator over LatticeNode objects

`origin`
lattice origin (x, y, z)

`point_data` **= None**
Easy access to the vtk PointData structure

`primitive_cell`
Primitive cell specifying the 3D Bravais lattice

`size`
lattice dimensions (nx, ny, nz)

`supported_cuba` **= None**
The currently supported and stored CUBA keywords.

update_nodes (*nodes*)
Update the corresponding lattice nodes.

nodes : iterator of LatticeNodes

9.4 Core module

A module containing core tools and wrappers for vtk data containers used in simphony_mayavi.

Classes

`CubaData`(attribute_data[, stored_cuba, ...])	Map a vtkCellData or vtkPointData object to a sequence of DataContainers.
`CellCollection`([cell_array])	A mutable sequence of cells wrapping a tvtk.CellArray.
`mergedocs`(other)	Merge the docstrings of other class to the decorated.
`CUBADataAccumulator`([keys])	Accumulate data information per CUBA key.
`CUBADataExtractor`(**traits)	Extract cuba data from cuds items iterable.

Functions

`supported_cuba`()	Return the list of CUBA keys that can be supported by vtk.
`default_cuba_value`(cuba)	Return the default value of the CUBA key as a scalar or numpy array.
`cell_array_slicer`(data)	Iterate over cell components on a vtk cell array
`mergedoc`(function, other)	Merge the docstring from the other function to the decorated function.

9.4.1 Description

class `simphony_mayavi.core.cuba_data.`**`CubaData`** (*attribute_data*, *stored_cuba=None*, *size=None*, *masks=None*)

Bases: `_abcoll.MutableSequence`

Map a vtkCellData or vtkPointData object to a sequence of DataContainers.

The class implements the `MutableSequence` api to wrap a `tvtk.CellData` or `tvtk.PointData` array where each CUBA key is a `tvtk.DataArray`. The aim is to help the conversion between column based structure of the `vtkCellData` or `vtkPointData` and the row based access provided by a list of `~.DataContainer`.

While the wrapped tvkt container is empty the following behaviour is active:

- Using `len` will return the `initial_size`, if defined, or 0.
- Using element access will return an empty *class:~.DataContainer*.
- No field arrays have been allocated.

When values are first added/updated with non-empty `DataContainers` then the necessary arrays are created and the `initial_size` info is not used anymore.

Note: Missing values for the attribute arrays are stored in separate attribute arrays named "<CUBA.name>-mask" as `0` while present values are designated with a `1`.

Constructor

attribute_data: tvtk.DataSetAttributes The vtk attribute container.

stored_cuba [set] The CUBA keys that are going to be stored default is the result of running `supported_cuba()`

size [int] The initial size of the container. Default is None. Setting a value will activate the virtual size behaviour of the container.

mask [tvtk.FieldData] A data arrays containing the mask of some of the CUBA data in `attribute_data`.

Raises

ValueError : When a non-empty `attribute_data` container is provided while size != None.

cubas
The set of currently stored CUBA keys.

For each cuba key there is an associated `DataArray` connected to the `PointData` or `CellData`

classmethod empty(*type_=<AttributeSetType.POINTS: 1>*, *size=0*)
Return an empty sequence based wrapping a vtkAttributeDataSet.

Parameters

- **size** (*int*) – The virtual size of the container.
- **type_** (*AttributeSetType*) – The type of the vtkAttributeSet to create.

insert(*index*, *value*)
Insert the values of the DataContainer in the arrays at row=‘‘index‘‘.

If the provided DataContainer contains new, but supported, cuba keys then a new empty array is created for them and updated with the associated values of `value`. Unsupported CUBA keys are ignored.

Note: The underline data structure is better suited for append operations. Inserting values in the middle or at the front will be less efficient.

class `simphony_mayavi.core.cell_collection.`**CellCollection**(*cell_array=None*)
Bases: `_abcoll.MutableSequence`

A mutable sequence of cells wrapping a tvtk.CellArray.

Constructor

Parameters **cell_array** (*tvtk.CellArray*) – The tvtk object to wrap. Default value is an empty tvtk.CellArray.

__delitem__(*index*)
Remove cell at `index`.

Note: This operation will need to create temporary arrays in order to keep the data info consistent.

__getitem__(*index*)
Return the connectivity list for the cell at `index`.

__len__()
The number of contained cells.

__setitem__(*index*, *value*)
Update the connectivity list for cell at `index`.

Note: If the size of the connectivity list changes a slower path creating temporary arrays is used.

insert (*index*, *value*)
Insert cell at `index`.

Note: This operation needs to use a slower path based on temporary array when index < sequence length.

class `simphony_mayavi.core.cuba_data_accumulator.`**CUBADataAccumulator** (*keys=()*)
Bases: `object`

Accumulate data information per CUBA key.

A collector object that stores :class:`DataContainer` data into a list of values per CUBA key. By appending `DataContainer` instanced the user can effectively convert the per item mapping of data values in a CUDS container to a per CUBA key mapping of the data values (useful for coping data to vtk array containers).

The Accumulator has two modes of operation `fixed` and `expand`. `fixed` means that data will be stored for a predefined set of keys on every `append` call and missing values will be saved as `None`. Where `expand` will extend the internal table of values whenever a new key is introduced.

expand operation

```
>>> accumulator = CUBADataAccumulator():
>>> accumulator.append(DataContainer(TEMPERATURE=34))
>>> accumulator.keys()
{CUBA.TEMPERATURE}
>>> accumulator.append(DataContainer(VELOCITY=(0.1, 0.1, 0.1))
>>> accumulator.append(DataContainer(TEMPERATURE=56))
>>> accumulator.keys()
{CUBA.TEMPERATURE, CUBA.VELOCITY}
>>> accumulator[CUBA.TEMPERATURE]
[34, None, 56]
>>> accumulator[CUBA.VELOCITY]
[None, (0.1, 0.1, 0.1), None]
```

fixed operation

```
>>> accumulator = CUBADataAccumulator([CUBA.TEMPERATURE, CUBA.PRESSURE]):
>>> accumulator.keys()
{CUBA.TEMPERATURE, CUBA.PRESSURE}
>>> accumulator.append(DataContainer(TEMPERATURE=34))
>>> accumulator.append(DataContainer(VELOCITY=(0.1, 0.1, 0.1))
>>> accumulator.append(DataContainer(TEMPERATURE=56))
>>> accumulator.keys()
{CUBA.TEMPERATURE, CUBA.PRESSURE}
>>> accumulator[CUBA.TEMPERATURE]
[34, None, 56]
>>> accumulator[CUBA.PRESSURE]
[None, None, None]
>>> accumulator[CUBA.VELOCITY]
KeyError(...)
```

Constructor

Parameters keys (*list*) – The list of keys that the accumulator should care about. Providing this value at initialisation sets up the accumulator to operate in `fixed` mode. If no keys are provided then accumulator operates in `expand` mode.

__getitem__(*key*)
Get the list of accumulated values for the CUBA key.

Parameters key (*CUBA*) – A CUBA Enum value

Returns result (*list*) – A list of data values collected for `key`. Missing values are designated with `None`.

__len__()
The number of values that are stored per key

Note: Behaviour is temporary and will probably change soon.

append(*data*)
Append info from a `DataContainer`.

Parameters data (*DataContainer*) – The data information to append.

If the accumulator operates in `fixed` mode:

- Any keys in `self.keys()` that have values in `data` will be stored (appended to the related key lits).
- Missing keys will be stored as `None`

If the accumulator operates in `expand` mode:

- Any new keys in *Data* will be added to the `self.keys()` list and the related list of values with length equal to the current record size will be initialised with values of `None`.
- Any keys in the modified `self.keys()` that have values in `data` will be stored (appended to the list of the related key).
- Missing keys will be store as `None`.

keys
The set of CUBA keys that this accumulator contains.

load_onto_vtk(*vtk_data*)
Load the stored information onto a vtk data container.

Parameters vtk_data (*vtkPointData or vtkCellData*) – The vtk container to load the value onto.

Data are loaded onto the vtk container based on their data type. The name of the added array is the name of the CUBA key (i.e. `CUBA.name`). Currently only scalars and three dimensional vectors are supported.

class `simphony_mayavi.core.cuba_data_extractor.`**CUBADataExtractor**(***traits*)
Bases: `traits.has_traits.HasStrictTraits`

Extract cuba data from cuds items iterable.

The class that supports extracting data values of a specific CUBA key from an iterable that returns low level CUDS objects (e.g. `Point`).

available = Property(Set(CUBATrait), depends_on='_available')
The list of cuba keys that are available (read only). The value is recalculated at initialialisation and when the `reset` method is called.

data = Property(Dict(UUID, Any), depends_on='_data')
The dictionary mapping of item uid to the extracted data value. A change Event is fired for `data` when `selected` or `keys` change or the `reset` method is called.

function = ReadOnly
The function to call that returns a generator over the desired items (e.g. Mesh.iter_points). This value cannot be changed after initialisation.

keys = Either(None, Set(UUID))
The list of uuid keys to restrict the data extraction. This attribute is passed to the function generator method to restrict iteration over the provided keys (e.g Mesh.iter_points(uids=keys))

reset()
Reset the `available` and `data` attributes.

selected = CUBATrait
Currently selected CUBA key. Changing the selected key will fire events that will result in executing the generator function and extracting the related values from the CUDS items that the iterator yields. The resulting mapping of `uid -> value` will be stored in `data`.

class simphony_mayavi.core.doc_utils.**mergedocs**(*other*)
Bases: `object`

Merge the docstrings of other class to the decorated.

simphony_mayavi.core.cuba_utils.**supported_cuba**()
Return the list of CUBA keys that can be supported by vtk.

simphony_mayavi.core.cuba_utils.**default_cuba_value**(*cuba*)
Return the default value of the CUBA key as a scalar or numpy array.

Int type values have `-1` as default, while float type values have `numpy.nan`.

Note: Only vector and scalar values are currently supported.

simphony_mayavi.core.cell_array_tools.**cell_array_slicer**(*data*)
Iterate over cell components on a vtk cell array

VTK stores the associated point index for each cell in a one dimensional array based on the following template:

```
[n, id0, id1, id2, ..., idn, m, id0, ...]
```

The iterator takes a cell array and returns the point indices for each cell.

simphony_mayavi.core.doc_utils.**mergedoc**(*function*, *other*)
Merge the docstring from the other function to the decorated function.

Symbols

A

B

C

D

M

O

P

R

S

U

V