HLRS

# COVISE

## Programming Guide

June 2016

**Title:**
**COVISE Programming Guide**
**January 26, 2022**

**Authors:**
**Martin Aumüller**
**Andreas Werner**
**Uwe Woessner**

# Contents

**1 Basics** **5**

1.1 Introduction . . . . . 5
1.2 Prerequisites . . . . . 5
1.3 Data flow model . . . . . 6
1.4 Execution sequence and module states . . . . . 6

**2 Module Programming** **9**

2.1 The coModule Base Class . . . . . 9
2.2 coModule's Functions . . . . . 9
2.3 coSimpleModule Set-Handler . . . . . 12
2.4 Ports and Parameters . . . . . 12
2.5 Data Ports . . . . . 13
2.6 Parameters . . . . . 15
2.6.1 Common functions . . . . . 15
2.6.2 Boolean Parameter . . . . . 16
2.6.3 Scalar Parameter . . . . . 17
2.6.4 Slider Parameter . . . . . 19
2.6.5 Vector Parameter . . . . . 21
2.6.6 String Parameter . . . . . 22
2.6.7 File Browser . . . . . 22
2.6.8 Choice Parameter . . . . . 23
2.6.9 Parameter switching groups . . . . . 25
2.7 COVISE configuration database . . . . . 27

**3 COVISE Data Objects** **29**

3.1 Available Data Object Types . . . . . 29
3.2 Two important warnings . . . . . 30
3.3 The Base Class for Distributed Objects . . . . . 30
3.4 Object types . . . . . 31
3.5 Attributes . . . . . 32
3.6 Structured Grid Types . . . . . 33
3.6.1 Abstract Structured Grid Base Class . . . . . 33
3.6.2 Uniform Grid . . . . . 33
3.6.3 Rectilinear Grid . . . . . 35
3.6.4 Structured Grid . . . . . 36
3.7 Unstructured Grid Types . . . . . 37

3.8 Data Types . . . 42
3.8.1 Scalar Data . . . 42
3.8.2 2D Vector Data . . . 42
3.8.3 3D Vector Data . . . 43
3.8.4 Tensor Data . . . 43
3.8.5 Packed RGBA Data . . . 44
3.9 Geometry data types . . . 45
3.9.1 Geometry Container Class . . . 45
3.9.2 Points . . . 46
3.9.3 Lines . . . 47
3.9.4 Polygons . . . 49
3.9.5 Triangle Strips . . . 52
3.10 Pixel Image Objects . . . 52
3.10.1 2D Textures . . . 53
3.11 Text Objects . . . 54
3.12 Integer Arrays . . . 54
3.13 Container Class coDoSet . . . 55

**4 Simulation Library** **57**
4.1 Usage models . . . 57
4.2 Basic structure . . . 57
4.3 Language bindings . . . 57
4.4 Simulation Server Commands . . . 58
4.5 Simulation Client Commands . . . 61
4.6 Handling data on parallel machines . . . 64

**5 Other Features** **67**
5.1 Error, Warning and Info Messages . . . 67
5.2 Explicit flow control commands . . . 67
5.3 Re-using objects . . . 68
5.4 Overriding parameter values . . . 68

**6 Example Modules** **73**
6.1 Hello . . . 73
6.2 Filter . . . 75

**7 OpenCOVER Plugin Programming** **81**
7.1 Background on Dynamic Libraries . . . 81
7.2 OpenSceneGraph . . . 81
7.3 OpenCOVER as a COVISE module . . . 82
7.4 OpenCOVER as a VR viewer for 3D geometry . . . 83
7.5 The Plugin Interface . . . 83

7.5.1 Plugin Life Cycle . . . . . 83
7.5.2 COVISE Interface . . . . . 84
7.5.3 Scene Graph Management . . . . . 85
7.5.4 Render Loop . . . . . 86
7.5.5 Other Events . . . . . 86
7.6 Accessing OpenCOVER from a plugin . . . . . 87
7.6.1 Access to the scene graph and its transformations . . . . . 87
7.6.2 Access to the camera . . . . . 89
7.6.3 Loading 3D Geometry . . . . . 89
7.6.4 Access to the Buttons of the Input Device . . . . . 90
7.6.5 Interactions . . . . . 90
7.6.6 Load and Unload Other Plugins and Communicate with them . . . . . 90
7.6.7 Append Buttons to the Pinboard . . . . . 91
7.7 Use OpenCOVER built-in interaction without the Pinboard . . . . . 94
7.8 Controlling a COVISE module from within OpenCOVER . . . . . 96
7.8.1 Class coFeedback . . . . . 96
7.8.2 Other functions . . . . . 97
7.8.3 A Plugin Programming Example . . . . . 98
7.8.4 Module Example . . . . . 99
7.9 Utility Classes . . . . . 99
7.9.1 The classes coIntersection and coAction . . . . . 99
7.9.2 OpenCOVER in a clustered environment . . . . . 101
7.9.3 opencover::coVRPluginSupport Class Reference . . . . . 101
7.9.4 opencover::coPointerButton Class Reference . . . . . 110
7.9.5 opencover::Isect Struct Reference . . . . . 112

**8 Quick Reference** **115**

8.1 coModule . . . . . 115
8.2 coSimpleModule . . . . . 116
8.3 coSimLib . . . . . 116
8.4 Simlib client . . . . . 116
8.5 coUifElem . . . . . 117
8.6 coPort . . . . . 117
8.7 coInputPort . . . . . 117
8.8 coOutputPort . . . . . 117
8.9 coUifPara . . . . . 118
8.10 coBooleanParam . . . . . 118
8.11 coFileBrowserParam . . . . . 118
8.12 coChoiceParam . . . . . 118
8.13 coFloatParam . . . . . 119
8.14 coFloatSliderParam . . . . . 119

8.15 coFloatVectorParam . . . . . 119
8.16 coInt32Param . . . . . 119
8.17 coIntSliderParam . . . . . 120
8.18 coIntVectorParam . . . . . 120
8.19 coStringParam . . . . . 120
8.20 coDistributedObject . . . . . 120
8.21 coDoUniformGrid . . . . . 121
8.22 coDoRectilinearGrid . . . . . 121
8.23 coDoStructuredGrid . . . . . 121
8.24 coDoUnstructuredGrid . . . . . 121
8.25 coDoFloat . . . . . 122
8.26 coDoVec2 . . . . . 122
8.27 coDoVec3 . . . . . 122
8.28 coDoTensor . . . . . 122
8.29 coDoRGBA . . . . . 122
8.30 coDoGeometry . . . . . 123
8.31 coDoPoints . . . . . 123
8.32 coDoLines . . . . . 123
8.33 coDoPolygons . . . . . 124
8.34 coDoTriangleStrips . . . . . 124
8.35 coDoTexture . . . . . 124
8.36 coDoPixelImage . . . . . 124
8.37 coDoText . . . . . 125
8.38 coDoIntArr . . . . . 125
8.39 coDoSet . . . . . 125
8.40 coCoviseConfig . . . . . 125

**List of Figures** **127**

**Index** **129**

# 1 Basics

## 1.1 Introduction

This documentation describes how to integrate new application modules into COVISE. There is also a section describing how to write plugins for the VR renderer. The programming functionality to interface and communicate with COVISE is explained in detail.

Currently, each COVISE module is realized as an operating system process. Future versions may allow to create modules which are modeled as subroutines or threads in a single process environment.

## 1.2 Prerequisites

To create new COVISE modules, the COVISE development distribution with all the necessary header files and libraries has to be installed on the target platform. For UNIX and similar systems, this includes also a set of scripts for Bourne Shell compatible shells to set up your compile environment, especially the environment variables necessary for compiling COVISE. Start a bash by typing `bash` (if this is not your default shell), change to the COVISE top level directory and type `source .covise.sh`. If your current working directory is not the COVISE top level directory and if your home directory does not contain the covise top level directory then you have to pass the location of this directory as a parameter to this script.

When compiling modules, COVISE automatically creates different directories for platform-specific parts like object files and executables: object files are generated in the objects_$ARCHSUFFIX subdirectory of your source ($COVISEDIR) directory, while binaries are put in $COVISEDIR/$ARCHSUFFIX/bin, where $ARCHSUFFIX is a platform identifier. Refer to $COVISEDIR/README-ARCHSUFFIX.txt for a full list. Here are some examples:

| ARCH | Platform |
|---|---|
| rhel6 | RedHat Enterprise Linux 6.x or compatible (CentOS, Scientific Linux) |
| win32 | Microsoft Windows using the Visual Studio 2003 compiler |
| lion | Apple Mac OS X 10.7 (x86_64) |

**Table 1: Architecture suffixes**

If you want to put your enhancements into a different directory, you can set the environment variable $COVISEDESTDIR to a directory where the added binaries should go. You should also add this directory to your $COVISE_PATH environment variable.

If you are going to develop COVISE modules, you might want to switch on "developer features" in the Mapeditor settings dialog. This will enable you to restart a module from its context menu or add an option to start a module in a debugger to the context menu of modules in the module library.

There are following requirements before starting to program:

- A `CMakeLists.txt` file used by CMake to generate a Makefile or compile instructions for other build environments: examples are available with the example modules, which reside in `$COVISEDIR/src/application/Exa` You can also use the template in `$COVISEDIR/src/template/module` as a starting point.
- A directory for the source code of the module is required.
- The `Hello` module in `$COVISEDIR/src/application/Examples/Hello` defines the complete basic structure of a module. It is recommended to copy this module source code as the base for own developments.

For users of the make build system having sourced the .covise.sh setup file, compiling the module is as simple as typing `make` in the module source directory.

COVISE has an object-oriented architecture requiring static initialization of multiple classes. For proper initialization, the main body of each module has to be written in C++. Nevertheless it is possible to integrate FORTRAN and C routines into an application module.

Each module is executed as a separate operating system process. Modules communicate in general using TCP/IP socket connections. This applies within a machine as well as between machines. Data exchange is handled differently. Within a machine pointers to shared memory are used to avoid copying of data objects. Between machines data objects are exchanged via TCP/IP socket connections. All this is handled transparently for the application programmer. The COVISE API (Application Programming Interface) is accessible through several libraries, which have to be linked to each new module. This happens automatically, if you specify `covise_add_module` as done in the CMakeLists.txt for the module template.

The main COVISE library is called `libcoCore.so` (`.sl` on HP, `.dll`/`.lib` on Windows, `.dylib` on Mac OS X). It is used for data management, message communication, starting processes, etc.

The Application library, called `libcoAppl.so`, contains the basic functionality to write application programs. It hides the details of packing and receiving COVISE messages and provides a framework for structuring application modules.

The library `libcoApi.so` builds on these libraries and provides the Application Programming Interface, which makes programming of applications easier and less error-prone. Whenever possible, programmers are recommended to use the higher level API functions instead of the direct COVISE calls provided with the Application library.

## 1.3 Data flow model

COVISE sessions are organized as a collection of modules, which are connected in a strictly unidirectional data flow network.

A typical data flow network is shown in the next figure. Data and control flows from top to bottom. Loops are not allowed, nevertheless, there are possibilities to send feedback messages from later to earlier modules in the processing chain.

Each module in the pipeline communicates with the central controller (the process named `covise`) and a local data-manager (named `crb`) by sending or receiving control messages via a TCP/IP socket. Data flow between modules is only a visual metaphor. Within a machine data objects are stored in shared memory segments. They are accessible from different modules by mapping their storage into the address space of the respective module processes. Between machines, objects are transferred by COVISE request brokers (CRBs), including necessary format conversions. This is transparent for the modules accessing the data objects. Both data and control communication are completely hidden within the COVISE libraries.

## 1.4 Execution sequence and module states

The sequence of states a COVISE module can have is shown in the following figure.

The start-up of a module is divided into two parts: The Constructor creates the module layout, which is sent to the COVISE controller using the module start message. In order to avoid timeout conditions during running the constructor, time consuming initialization operations should be put into an additional user subroutine which is called after establishing the COVISE connectivity to allow user-supplied start-up routines to be performed without timeout problems. After the start-up sequence, COVISE enters a main event loop, which is only left for the module shut-down. In the main loop, COVISE can receive different events, which are then handled by one of the following event handlers:

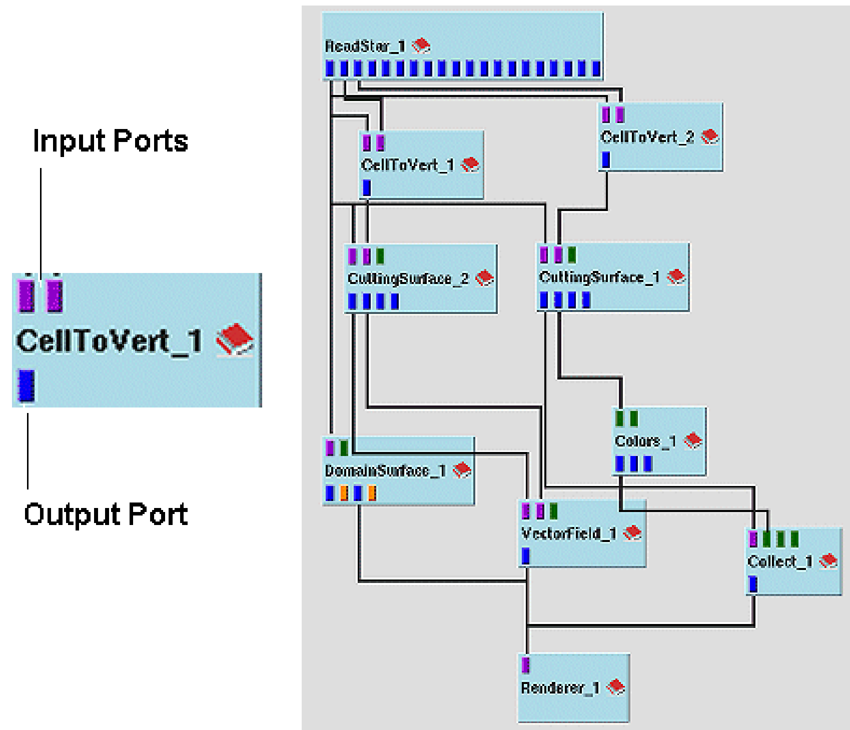



Figure 1.1: Module and Dataflow network

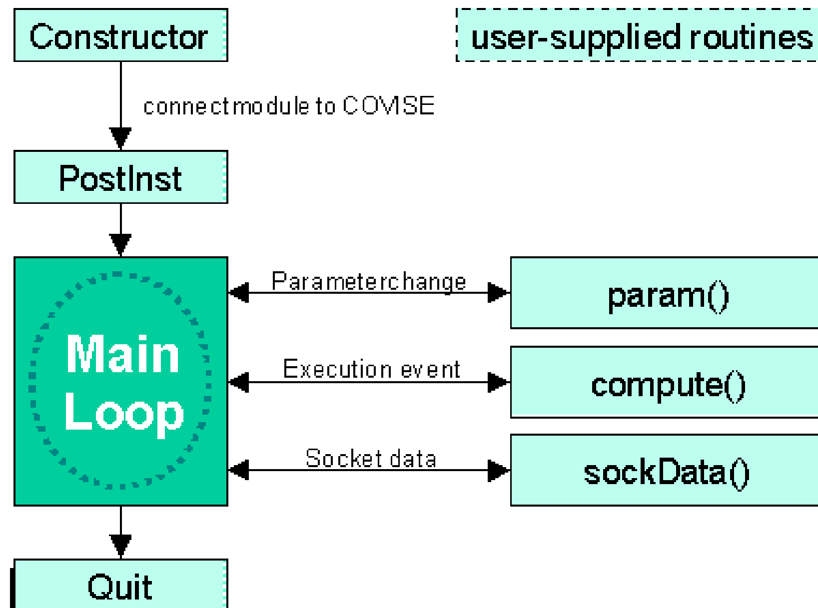



Figure 1.2: Module execution scheme

- *param*: Every change of a module parameter submits a message to the module. The `param` subroutine reacts on these change messages.
- *compute*: COVISE tells the module to read its input data and create output data. Only within this routine data objects in shared memory can be read or new data objects in shared memory can be created.
- *sockData*: The user can register open network sockets to be monitored. Whenever data on one of these sockets arrives, sockData is called.

It is important to note that several commands are only allowed when COVISE is in the appropriate state. Due to performance reasons these states are not generally checked, thus calling functions during illegal states may result in module crashes.

# 2 Module Programming

## 2.1 The coModule Base Class

The base class for all module programming is `coModule`. An application is created by deriving a class of `coModule`. The constructor of the derived class creates the module layout with input ports, output ports and parameters. Virtual functions are overloaded to implement the module's reactions on events. Every COVISE module based on `coModule` looks like:

Header file:

```
class myMod: public coModule
{
   private:
      ... // parameter and data object ports

   public:
      myMod(int argc, char *argv[]);     // Constructor: module set-up
      virtual void compute(const char *);       // Called for every execute
      virtual void param(const char *, bool inMapLoading);
                                         // Called for param changes
      virtual void sockData(int sockNo); // Called for receiving data
}
```

Source code:

```
myMod::myMod(int argc, char *argv[])
: coModule(argc, argv, "My example module")
                               {...}    // build ports and parameters
myMod::compute(const char *)         {...}    // module is executed
myMod::param(const char *, bool inMapLoading)
                               {...}    // parameters changed
myMod::postInst(const char *)  {...}    // between c'tor and main loop
myMod::quit(const char *)      {...}    // after the end of main loop
myMod::sockData(int socNo)     {...}    // data arrived on a socket

MODULE_MAIN(SomeCategory, myMod)
```

To allow future use of threads and multi-linked modules, modules should use neither global nor non-constant static variables.

## 2.2 coModule's Functions

### Constructor

The constructor of a module is used to set up its external connectivity. First, it calls the constructor of the base class `coModule`:

```
void coModule::coModule(int argc, char *argv[], const char *description=NULL, bool propagate
```

The constructor of coModule establishes some basic data structures and pre-sets them with defaults values.

The execution of a module constructor is time-critical. If the constructor is not finished within the timeout period given in the config file, the CRB will not accept the module and the user interface will issue an error message telling that the module did not start.

In general, the constructor should never do anything else than defining the ports and parameters of the module.

## postInst

For any additional functionality required before entering the main loop the routine

```
virtual void coModule::postInst()
```

should be overloaded by the module. It is called once before entering the main event loop. This avoids timeouts, which could occur as described in the constructor paragraph.

## param

Since COVISE 6.1, a module is notified immediately unconditionally when a parameter was changed by the user (i. e. all parameters are “immediate” now). For any changed parameters, the callback

```
virtual void coModule::param(const char *paramName, bool inMapLoading)
```

is called with the name of the port as the parameter.

## compute

The compute callback is called whenever new input data for a module is available or when the user executes a module manually.

```
virtual int coModule::compute(const char *)
```

Before calling the user supplied routine, the input objects are retrieved from the data manager and the object pointer is stored to be retrieved from the port. The return value of the compute callback must be either CONTINUE_PIPELINE or STOP_PIPELINE depending on whether modules below should be executed or not.

Most modules only need a compute function and do not use the other functions. All functions are pre-initialized with empty functions in case the user does not overload them. A warning is submitted if compute(const char *) is executed without being overloaded.

## sockData

If a module needs to use any other external communication means than the COVISE internal networking, e. g. connect to a simulation or use the X Window system, the programmer has to make sure that COVISE

messages are still received and handled. Therefore, a file descriptor, e. g. a socket, can be registered with the module via

```
void addSocket(int fd)
```

Once this has been done, COVISE adds the socket to its own sockets and calls the virtual function

```
virtual void coModule::sockData(int fd)
```

with the file descriptor as the argument. The user can then handle the input and return control to COVISE by leaving the callback. After use, the socket is removed using

```
void removeSocket(int fd)
```

## quit

A module terminates either when COVISE terminates, the user deletes a module or a new map is loaded. To allow cleaning up resources, the routine

```
virtual void coModule::quit()
```

is automatically called before the module quits.

## idle

The idle function is called whenever the module waits for COVISE messages.

```
virtual float coModule::idle()
```

You should not block in this function, otherwise the module will never get the chance to process incomming messages. If you overwrite this method, you have to return a float value. This value specifies the maximum number of seconds to wait until the idle function is called again. If zero is returned, the idle function is called again immediately after checking for COVISE messages. If a negative value is returned, the idle function will not be called until any message arrived. If a positive Value is returned, the idle function will be called after the appropriate time, or earlier if any message arrived.

Remember: You can't create COVISE distributed objects in this callback!

## Other coModule functions

```
virtual void feedback(int len, const char *data);
```

This function is called when a COVER plugin sends a feedback message.

```
virtual void feedbacksetInfo(int len, const char *datatext);
```

Allows to set an info string in the control panel of the module.

## 2.3 coSimpleModule Set-Handler

COVISE can handle hierarchical structures of data created by Set containers, e.g. timestep series or data sets consisting of multiple distinct parts. A module handling these data types has to be written recursively to be able to handle arbtrary levels of hierarchy. In most cases, modules work with 'corresponding' data, e.g. a set of geometry objects required an identical set of colors and normals, or a set of timesteps of a moving grid simulation corresponds to a similar set of data values.

For easier module development, the class coSimpleModule has been derived from coModule. It automatically un-packs all level of Set containers until reaching the lowes level, and then calls compute(const char *) for each low-level object. The module developer simply creates the module and writes a compute(const char *) function as if the data was non-hierarchical, which is then called for every corresponding set of low-level object.

The functions

```
void setComputeTimesteps(const int off);    // 0 by default
void setComputeMultiblock(const int off);   // 0 by default
```

can be called if the user wants to handle timesteps or multiblock data himself, e.g. for particle tracing.

The simple module also automatically copies all attributes from the input objects to the output objects at the port directly underneath the input port. If this is not desired, it can be switched of by calling

```
void setCopyAttributes(const int on);      // 1 by default
```

## 2.4 Ports and Parameters

Ports establish the connection of a module to the rest of the COVISE system. Three different kinds of connections can be defined:

1. Parameters: Values interactively set in the Map-Editor by the user
2. Input data ports: Receive data from other modules
3. Output data ports: Send data to other modules

All ports and parameters are created by calling a member function of coModule that returns a pointer to a port class object, which can later be used to set or retrieve port data. These pointers will usually be stored as module class private data being usable in the whole module.

The creation of ports and parameters only possible in the Constructor state, a later creation of ports is not supported. Ports can be hidden or de-activated if not currently needed.

**Never create Modules with variable numbers of Ports or Parameters: These modules will crash COVISE when being read from a map file!**

All ports and parameters must have a unique name and description, which can be any text explaining the function of the port. Names must not contain blank characters or any special ASCII characters like TAB, CR, LF, DEL, BS. The description may contain blanks, but no special characters like CR, LF or BS characters and need not be unique.

Never construct a port or parameter by its own constructor, COVISE must register it with the module. Instead, the functions add<type>Port and add<type>Param have to be used.

All input and parameter ports derive from the same base class coUifElem, which implements some base functionality for all kinds of ports. The following subroutines can therefore be called for all parameter and port class objects:

| **const char *coUifElem::getName() const** | |
|---|---|
| Description: | Get port's name |
| COVISE states: | all |
| Return value: | Port name |

| **const char *coUifElem::getDesc() const** | |
|---|---|
| Description: | Get port's description |
| COVISE states: | all |
| Return value: | Port description |

| **virtual void coUifElem::print(ostream &str) const** | | |
|---|---|---|
| Description: | Print port information into a stream | |
| COVISE states: | all | |
| IN: | str | stream to print into |

## 2.5 Data Ports

When replacing modules, data port connections can only be maintained if the ports adhere to a common naming scheme. A port name should start with a prefix from the following table. If the port is able to accept data types from several lines in this table, then choose "Data", except for when it is Geometry and grid types, then choose "Grid". As appropriate, append "In" or "Out" to the prefix. Finally you should append the number of the port: start with "0" for the first port with a certain prefix and direction, and increase the number for each other port with the same prefix and direction. Also append the final "0" if there is only one port of this type.

| Name prefix | Data object types |
|---|---|
| Data | Int, Float, Vec2, Vec3, RGBA, Mat, Tensor |
| Grid | Points, Spheres, Lines, Polygons, TriangleStrips,<br>UniformGrid, RectilinearGrid, StructuredGrid, UnstructuredGrid |
| Geometry | Geometry |
| Texture | Texture |

Data ports are created by:

| **coInputPort *coModule::addInputPort(const char *name,const char *typeList, const char *desc)** | | |
|---|---|---|
| Description: | Create an Input data port | |
| COVISE states: | Constructor | |
| IN: | name: | Port name |
| IN: | typeList: | Map editor data types |
| IN: | desc: | Parameter description |
| Return Value: | pointer to newly created port | |

| **coOutputPort *coModule::addOutputPort(const char *name, const char *typeList, const char *desc)** | | |
|---|---|---|
| Description: | Create an Output data port | |
| COVISE states: | Constructor | |
| IN: | name: | Port name |
| IN: | typeList: | Map editor data types |
| IN: | desc: | Parameter description |
| Return Value: | pointer to newly created port | |

The parameter types string lists all type names of objects allowed to connect with this port divided by a '|' character, e.g.:

```
"Float|Vec3"
```

The type string must not contain any blanks, otherwise it is not possible to connect the ports in the map editor. This type information is only given to the map editor, which then prohibits connections between ports without common types. The port information does not imply any checking of data types assigned to the port by the module, so type-checking must still be implemented in the module. The reason for not checking the data types according to the map editor types is to allow different usage of the same data type by different map editor types enforcing correct connections.

By default, a module only executes (i.e. calls its compute callback) if all input ports of a module are connected. By declaring a port 'not required', the pipeline will also be executed without a connection to this port. The module will then receive a NULL pointer when trying to retrieve the port's data.

| **void coInputPort::setRequired(int isRequired)** | | |
|---|---|---|
| Description: | declare a port (not) required | |
| COVISE states: | Compute | |
| IN: | isRequired | =0:not required, else required |

Whenever the compute(const char *) callback is called by the main loop, the data objects at the input ports are opened and can be read by calling the input port's getCurrentObject() method. If the port is not connected, or if an error occurs, a NULL pointer is returned.

A non-required input port can be declared as a 'dependency' of an output port, meaning it becomes required if this output port is connected.

| **void coOutputPort::setDependency(coInputPort *port)** | | |
|---|---|---|
| Description: | declare an output port dependending on a certain input | |
| COVISE states: | Constructor | |
| IN: | port | port depended upon |

If no data is available on a connected port, the compute callback is not called and the start message is silently ignored. This allows a direct flow control by not creating some of the defined output objects which leads to branches of the data flow network that are not executed.

| **coDistributedObject *coInputPort::getCurrentObject()** | |
|---|---|
| Description: | Retrieve input data |
| COVISE states: | Compute |
| Return value: | Object received at the port |

The resulting object pointer is a pointer to a derived class of coDistributedObject, which can be casted up to the real class. To find the correct type, the base class member function `getType()` can be used.

| const char *coOutputPort:: getObjName() | |
|---|---|
| Description: | get object name for output objects |
| COVISE states: | Compute |
| Return value: | pointer to name for new data object |

This call delivers the appropriate name to create data objects for an output port. The return value must be given to the constructor of the object, which is then assigned to the port for sending it to all connected modules by

| void coOutputPort::setCurrentObject(coDistributedObject *obj) | |
|---|---|
| Description: | Assign output data object to port |
| COVISE states: | Compute |

**Never delete objects assigned to a port**

The output object will be sent to all connected modules after the compute(const char *) callback has finished and is then automatically deleted. Examples for object creation and receiving can be found in the example modules under `covise/src/examples`.

The tool-tip text, which appears when right-clicking on a port, can be set by:

| void coPort::setInfo(const char *text) | | |
|---|---|---|
| Description: | set tooltip text | |
| COVISE states: | After initialisation (postInst, compute, ...) | |
| IN: | text | tooltip text |

## 2.6 Parameters

### 2.6.1 Common functions

All parameter classes are derived from `coUifPara`, which offers common functions for all kinds of parameter ports:

To allow direct interaction with the module, all parameter changes are immediately sent to the module and the module's param() callback is fired. If the callback does not react on the parameter, the value is still updated, but no further action is taken.

A parameter can be mapped into the Control Panel by clicking the checkbox in the Module Set-up window. Nevertheless, the module programmer can control the mapping by calls to the functions:

| void coUifPara::show() | |
|---|---|
| Description: | Show/Hide a parameter in the control panel |
| COVISE states: | PostInst, all Main-Loop callbacks |

| **void coUifPara::hide()** | |
|---|---|
| Description: | Show/Hide a parameter in the control panel |
| COVISE states: | PostInst, all Main-Loop callbacks |

Parameters can also be disabled by the module. If a parameter is disabled, it is displayed in gray both in the Module set-up panel and in the control panel and no parameter changes are possible. Parameter disabling is typically used for parameters that are only required under certain circumstances, e.g. in simulation couplings, which can have different sets of parameters depending on the simulation case.

| **void coUifPara::enable()** | |
|---|---|
| Description: | Enable or disable parameter |
| COVISE states: | PostInst, all Main-Loop callbacks |

| **void coUifPara::disable()** | |
|---|---|
| Description: | Enable or disable parameter |
| COVISE states: | PostInst, all Main-Loop callbacks |

### 2.6.2 Boolean Parameter

A Boolean parameter can only have the values TRUE ($\neq 0$) or FALSE (=0).

The default value of a Boolean parameter is FALSE.

A Boolean parameter is created by:

| **coBooleanParam *coModule::addBooleanParam (const char *name, const char *desc)** | | |
|---|---|---|
| Description: | Create a Boolean parameter | |
| COVISE states: | Constructor | |
| IN: | name: | Parameter name |
| IN: | desc: | Parameter description |
| Return value: | pointer to new created port | |

This function creates a new port and registers it at the module for receiving port messages. The value of a Boolean Parameter can be requested and set by the program using:

| **int coBooleanParam::getValue()** | |
|---|---|
| Description: | get Value of a Boolean parameter |
| COVISE states: | all |
| Return value: | Value of the parameter |

| **int coBooleanParam::setValue(int value)** | | |
|---|---|---|
| Description: | set Value of a Boolean parameter | |
| COVISE states: | all | |
| IN: | value | value to be set |
| Return value: | =0 on error, =1 on success | |

A typical code fragment is:

myModule.h

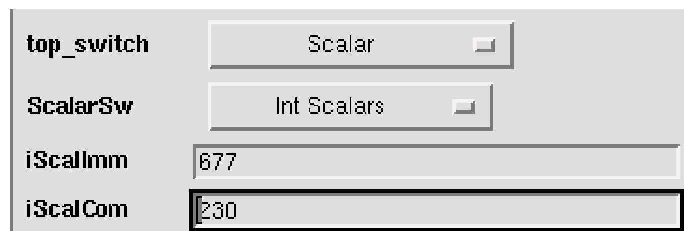
```
class myModule
{
   private:

      coBooleanParam *p_shadeFlag;
      ...

}
```

myModule.cpp

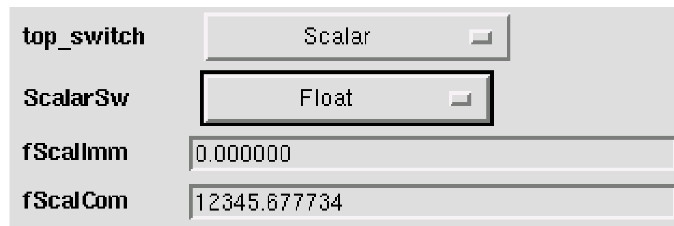
```
myModule::myModule()   // ... build ports and parameters
{
  // create the boolean port: declared as member variable
  p_shadeFlag = addBooleanParameter("shade","apply shading");

  // set a default value

  p_shadeFlag->setValue(1);
}

myModule::compute(const char *)    // ... e.g. in the compute callback
{
  int shading = p_shadeFlag->getValue();
  ...
}
```

It is a good habit to mark all ports and parameters with a common prefix. In our examples we have chosen names that start with 'p_'.

### 2.6.3 Scalar Parameter




There are integer and float scalar parameters. Both represent a single value of the corresponding type. The integer scalar parameter is exactly identical to the float one except for replacing *"...Float..."* by *"...Int..."* in the names of functions and changing the data types.




Both float and integer scalar parameters default to 0.

A float scalar parameter is created by:

| **coFloatParam *coModule::addFloatParam (const char *name, const char *desc)** | | |
|---|---|---|
| Description: | Create a float scalar parameter | |
| COVISE states: | Constructor | |
| IN: | name: | Parameter name |
| IN: | desc: | Parameter description |
| Return value: | pointer to newly created port | |

The value of a scalar parameter can be requested/set by the program using:

| **float coFloatParam::getValue()** | |
|---|---|
| Description: | get Value of a float scalar parameter |
| COVISE states: | all |
| Return value: | Value of the parameter |

| **int coFloatParam::setValue(float value)** | | |
|---|---|---|
| Description: | set Value of a float scalar parameter | |
| COVISE states: | all | |
| IN: | value | value to be set |
| Return value: | =0 on error, =1 on success | |

The corresponding commands for the integer scalar parameter are:

```
coInt32Param *coModule::addInt32Param(const char *name,
                                      const char *desc)
long coInt32Param::getValue()
int coInt32Param::setValue(long value)
```

A typical code fragment is:

myModule.h

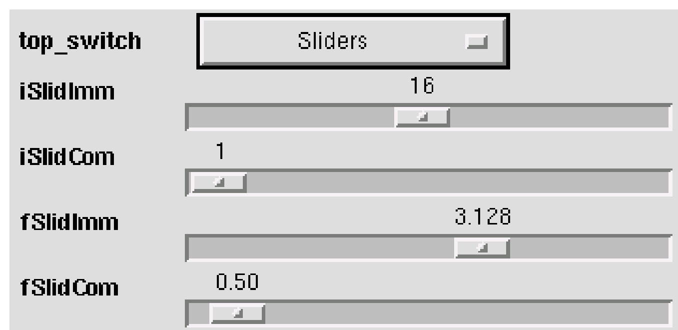
```
class myModule
{
   private:

      coFloatParam *p_timestep;
      coInt32Param   *p_numsteps;
      ...

}
```

myModule.cpp

```
myModule::myModule()   // ... build ports and parameters
{
   p_timestep=addFloatParam ("timestep","length of a timestep");
   p_numsteps=addFloatParam ("numsteps","number of steps");
```

```
   // set a default values
   p_timestep->setValue(0.001);
   p_numsteps->setValue(100);
}

myModule::compute(const char *)    // compute callback
{
   float timestep =  p_timestep->getValue();
   int   numsteps =  p_numsteps->getValue(); // we can access it here!
   ...
}

myModule::param(const char *portname)      // param callback
{
   if ( strcmp(portname,numsteps->getName()) )
      .... do something when the user changes number of steps
}
```

### 2.6.4 Slider Parameter




Sliders can also be either of type float or integer: Both of them have a minimum, maximum and a ‘current’ value, which can be set or requested by the module.

Float sliders default to 0...0.5...1.0, int sliders to 0...127...255.

A Slider parameter is created by:

| **coFloatSliderParam *coModule::addFloatSliderParam (const char *name, const char *desc)** | | |
|---|---|---|
| Description: | Create a float slider parameter | |
| COVISE states: | Constructor | |
| IN: | name: | Parameter name |
| IN: | desc: | Parameter description |
| Return value: | pointer to newly created port | |

The value of a slider parameter can be requested/set by the program using:

| **void coFloatSliderParam::getValue (float &min, float &max, float &value)** | | |
|---|---|---|
| Description: | get Value of a float slider parameter | |
| COVISE states: | Constructor | |
| IN: | min/max | lower/upper bound of slider |
| IN: | value | current value of slider |
| Return value: | none | |

| **void coFloatSliderParam::setValue (float min, float max, float value)** | | |
|---|---|---|
| Description: | set Value of a float slider parameter | |
| COVISE states: | all | |
| IN: | min/max/value | value to be setlike getValue() |
| Return value: | =0 on error, =1 on success | |

Single values can also be requested or set:

```
float coFloatSliderParam::getMin()
float coFloatSliderParam::getMax()
float coFloatSliderParam::getValue()
void  coFloatSliderParam::getMin(float min)
void  coFloatSliderParam::getMax(float max)
void  coFloatSliderParam::getValue(float value)
```

As for the scalar parameter, there are corresponding integer functions, exactly like theirs float counterparts but with "`Int`" in the name and integer return/param type.

A typical code fragment is:

myModule.h

```
class myModule
{
   private:

      coFloatSliderParam *p_relax;
      ...

}
```

myModule.cpp

```
myModule::myModule()   // ... build ports and parameters
{
   p_relax=addFloatSliderParam ("p_relax","relaxation factor");
   p_relax->setValue(0.0,1.0,0.95);
}

myModule::compute(const char *)    // compute callback
{
   float relax =  p_relax->getValue();
   ...

   // if the relaxation is too high, we push it down...
   relax = relax * 0.9;
   p_relax->setValue(relax);
}
```

### 2.6.5 Vector Parameter

Vectors are parameters with an arbitrary number of scalar values. The number of fields is not limited by the system, but since there is no numbering in the control panel, it should be limited to a small number for clarity reasons.

Both integer and float vector parameters default to 3D null vectors.

A vector parameter is created with a default size of 3 elements by:

| **coFloatVectorParam *coModule::addFloatVectorParam (const char *name, const char *desc)** | | |
|---|---|---|
| Description: | Create a float vector parameter | |
| COVISE states: | Constructor | |
| IN: | name: | Parameter name |
| IN: | desc: | Parameter description |
| Return value: | pointer to newly created port | |

This is the only way for a user to specify vector parameters with another size than three elements.

The value of any vector parameter can be requested by the program using:

| **float coFloatVectorParam::getValue (int pos)** | | |
|---|---|---|
| Description: | get value of one element in a float vector parameter | |
| COVISE states: | all | |
| IN: | pos | select which element to get |
| Return value: | requested value | |

If the position parameter is out of bounds, a warning is issued and 0 is returned.

A single value of a scalar parameter can also be set by the module:

| **int coFloatVectorParam::setValue(int pos, float value)** | | |
|---|---|---|
| Description: | set value of one element in a vector parameter | |
| COVISE states: | all | |
| IN: | pos | select which element to get |
| IN: | value | value to be set |
| Return value: | =0 on error, =1 on success | |

To change the number of values in the field, an array can be used to initialize the parameter:

| **int coFloatVectorParam::setValue(int numElem, float *field)** | | |
|---|---|---|
| Description: | set number of elements and set all values | |
| COVISE states: | all | |
| IN: | numElem | select which element to get |
| IN: | field | value to be set |
| Return value: | =0 on error, =1 on success | |

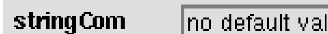
Since 3-dimensional vectors are used very often, commodity functions for dealing with 3D vectors are defined:

| **int coFloatVectorParam::setValue (float data0, float data1, float data2)** | | |
|---|---|---|
| Description: | set 3-element vector parameter values | |
| COVISE states: | all | |
| IN: | data1...3 | values to be set |
| Return value: | =0 on error, =1 on success | |

All these functions are also defined for integer vectors:

```
coIntVectorParam*coModule::addIntSliderParam(const char *name, const char *desc);
int coIntVectorParam::getValue(int pos, long &value);
int coIntVectorParam::setValue(int pos, long value);
int coIntVectorParam::setValue(int numElem, long *field);
int coIntVectorParam::setValue(Int data0, long data1, Int data2);
```

### 2.6.6 String Parameter




A string is a 0-terminated sequence of characters.

The default value is the string "no default val".

A string parameter is created by:

| **CoStringParam *coModule::addStringParam (const char *name, const char *desc)** | | |
|---|---|---|
| Description: | Create a string parameter | |
| COVISE states: | Constructor | |
| IN: | name: | Parameter name |
| IN: | desc: | Parameter description |
| Return value: | pointer to newly created port | |

The value of a string parameter can be requested by the program using:

| **const char *coStringParam::getValue()** | |
|---|---|
| Description: | get value of a string parameter |
| COVISE states: | all |
| Return value: | Value of the parameter |

The value of a string parameter can also be set by the module:

| **int coStringParam::setValue(const char *val)** | | |
|---|---|---|
| Description: | set value of a string parameter | |
| COVISE states: | all | |
| IN: | val | new value for the parameter |
| Return value: | =0 on error, =1 on success | |

### 2.6.7 File Browser

A File Browser parameter allows the selection of a file on the host the module is running on.

*The File Browser parameter cannot be updated by the module.* `setValue` calls outside the constructors are ignored.

A Browser parameter is created by:

| **coFileBrowserParam *coModule::addFileBrowserParam (const char *name, const char *desc)** | | |
|---|---|---|
| Description: | Create a browser parameter | |
| COVISE states: | Constructor | |
| IN: | name: | Parameter name |
| IN: | desc: | Parameter description |
| Return value: | pointer to newly created port | |

The value of a browser parameter can be requested by the program using:

| **const char *coFileBrowserParam::getValue()** | |
|---|---|
| Description: | get filename selected in a browser parameter |
| COVISE states: | all |
| Return value: | value of the parameter |

The start value of a browser parameter must be set by the module in the constructor

| **int void coFileBrowserParam::setValue (const char *defaultFile, const char *mask)** | | |
|---|---|---|
| Description: | set the initial value of a file browser | |
| COVISE states: | all | |
| IN: | defaultFile | default file name with path |
| IN: | mask | file selection mask, e.g. "*.dat" |
| Return value: | =0 on error, =1 on success | |

### 2.6.8 Choice Parameter

A choice parameter allows to select one entry from a list of given choice strings.

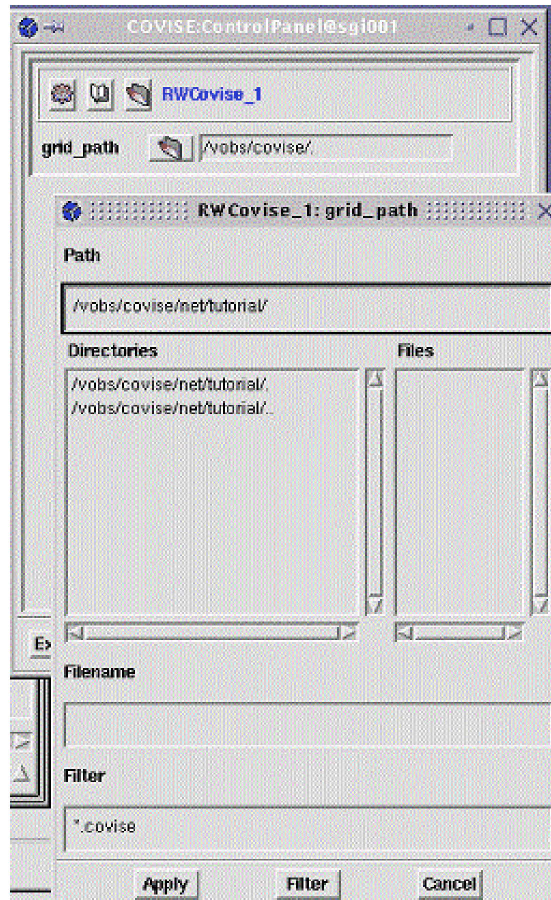
A choice parameter is created by:

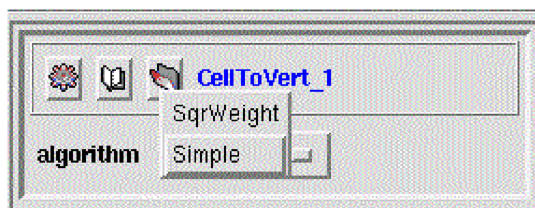
| **coChoiceParam *coModule::addChoiceParam (const char *name, const char *desc)** | | |
|---|---|---|
| Description: | Show a parameter in control panel | |
| COVISE states: | Constructor | |
| IN: | name: | Parameter name |
| IN: | desc: | Parameter description |
| Return value: | pointer to newly created port | |

The active choice list entry can be retrieved by:

| **int coChoiceParam::getValue()** | |
|---|---|
| Description: | get value of a choice parameter |
| COVISE states: | all |
| Return value: | active choice, count starts with 1 |

The choice list can be set by:


COVISE:ControlPanel@sgi001
RWCovise_1
grid_path
/vobs/covise/.
RWCovise_1: grid_path
Path
/vobs/covise/net/tutorial/
Directories
Files
/vobs/covise/net/tutorial/.
/vobs/covise/net/tutorial/..
Filename
Filter
*.covise
Apply
Filter
Cancel


CellToVert_1
SqrWeight
algorithm
Simple

| int coChoiceParam::setValue(const char *val) | | |
|---|---|---|
| Description: | set value of a choice parameter | |
| COVISE states: | all | |
| IN: | val | new value for the parameter |
| Return value: | =0 on error, =1 on success | |

| int coChoiceParam::setValue<br>(int num, const char * const *label, int active) | | |
|---|---|---|
| Description: | set value of a choice parameter | |
| COVISE states: | all | |
| IN: | num | number of entries |
| IN: | label | array of strings:<br>define as 'char *arr[num]' |
| IN: | active | pre-set value: count starts with 1 |
| Return value: | =0 on error, =1 on success | |

If you want to use the same **contents** of a choice parameter, even if it appears at a different **position (number)** use the update function:

| int coChoiceParam::updateValue<br>(int num, const char * const *label, int active) | | |
|---|---|---|
| Description: | set value of a choice parameter | |
| COVISE states: | all | |
| IN: | num | number of entries |
| IN: | label | array of strings:<br>define as 'char *arr[num]' |
| IN: | active | pre-set value: count starts with 1, use current label if possible. If this is possible, ignore active |
| Return value: | =0 on error, =1 on success | |

The current value can be set by:

| int coFloatVectorParam::setValue(int pos, float value) | | |
|---|---|---|
| Description: | set value of one element in a vector parameter | |
| COVISE states: | all | |
| IN: | pos | select which element to get |
| IN: | value | value to be set |
| Return value: | =0 on error, =1 on success | |

See the example code `$COVISEDIR/src/examples/UpdateChoice` for an example using all features of the Choice parameter.

### 2.6.9 Parameter switching groups

Parameters can be grouped in the control panel. The syntax for grouping parameters is:

```
paraSwitch("top_switch","Now switch Parameters"); // switch with
                                                     name and label
   paraCase("First possibility label");    // first possibility
      // ... all parameters we want to change in this group
   paraEndCase();
```

```
   paraCase("other possibility");   // Next switch possibility
      // ... all parameters we want to change in this group
      // second level of switching
      paraSwitch("subSwitch","Sub-Cases"); // nested switch
         paraCase("one case");
            // ...parameters // one case
         paraEndCase();
         paraCase("other case");
            // ...parameters
         paraEndCase();
      // second level of switching ends here
      paraEndSwitch();
   // The 'Scalar' case ends here
   paraEndCase();
paraEndSwitch();
```

Each `paraSwitch` call creates a 'master' choice parameter, which automatically shows and hides the parameters of its `paraCase` groups. The name of the choice is the first parameter of the `paraSwitch` command, while the labels of the `paraSwitch` and all following `paraCase` commands are automatically added to the choice list. As the first choice, the 'master' displays its own label and shows none of its cases.

The user can access the switch choices as usual choice parameters using the `setValue` command, e.g. to change the choice labels, but the assignment between choice result number and `paraCase` can not be influenced, so choice number 1 always is 'no parameters', number 2 the first case, number 3 the second and so on. This feature will typically be used to de-activate complete cases from the control panel.

Notice: the module set-up panel will always show ALL parameters of the module.

The example program TestUIF shows the implementation of a 2-level hierarchy of switched parameter groups.

| **coChoiceParam *coModule::paraSwitch(const char *choiceName, const char *label)** | | |
|---|---|---|
| Description: | Start parameter switching group,<br>define name and top label of choice | |
| COVISE states: | Constructor | |
| IN: | choiceName | Unique parameter port name |
| IN: | label | Label for the choice |
| Return value: | Pointer to the 'master' choice port | |

| **void coModule::paraEndSwitch()** | |
|---|---|
| Description: | End parameter switching group |
| COVISE states: | Constructor |

| **int coModule::paraCase(const char *label)** | | |
|---|---|---|
| Description: | Start a case within a switched group and give corresponding choice label | |
| COVISE states: | Constructor | |
| IN: | label | Label for the choice |
| Return value: | -1 on error, 0 on success | |

| void coModule::paraEndCase() | |
|---|---|
| Description: | End a group within a switched parameter area |
| COVISE states: | Constructor |

## 2.7 COVISE configuration database

COVISE has a central configuration database, which resides in XML files in the `$COVISEDIR/config` directory or in the `.covise` subdirectory of the user's home directory.

The configuration database is accessed by calling static methods of the `coCoviseConfig` singleton.

Several functions are provided to access entries in the configuration database. Most of these access functions require a single 'entry' parameter `Scope.Var`. This 'entry' parameter describes the path along which to descend in the tree of nested XML elements. Elements are separated by a dot ("."). However, the first two path elements are defined implictly (e. g. "COCONFIG.GLOBAL") and must not be specified. For all data types, there are two kinds of query functions. Those taking only an 'entry', return the value of the "value" attribute of the corresponding XML element. For querying another attribute, you have to use the method with 'variable' and 'entry' arguments.

Consider the following configuration file:

```
<?xml version="1.0"?>
<COCONFIG version="1">
 <GLOBAL>
  <Module>
   <MyModule>
    <BufferSize value="1024" />
   </MyModule>
  <Module>
 </GLOBAL>
</COCONFIG>
```

The "value" attribute of "BufferSize" could be queried with `coCoviseConfig::getInt("value", "Module.MyModule.Buf` or `coCoviseConfig::getInt("Module.MyModule.BufferSize")`.

| **const char *getEntry(const char *entry)**<br>**const char *getEntry(const char *variable, const char *entry)** | | |
|---|---|---|
| Description: | get a single string entry | |
| IN: | entry | path to XML tag |
| IN: | variable | attribute, the value of which should be returned |
| Returns | value of configuration | |

| **int getInt(const char *entry)**<br>**int getInt(const char *variable, const char *entry)** | | |
|---|---|---|
| Description: | get a single string entry | |
| IN: | entry | path to XML tag |
| IN: | variable | attribute, the value of which should be returned |
| Returns | value of configuration | |

| **long getLong(const char *entry)**<br>**long getLong(const char *variable, const char *entry)** | | |
|---|---|---|
| Description: | get a single string entry | |
| IN: | entry | path to XML tag |
| IN: | variable | attribute, the value of which should be returned |
| Returns | value of configuration | |

| **float getFloat(const char *entry)**<br>**float getFloat(const char *variable, const char *entry)** | | |
|---|---|---|
| Description: | get a single string entry | |
| IN: | entry | path to XML tag |
| IN: | variable | attribute, the value of which should be returned |
| Returns | value of configuration | |

| **bool isOn(const char *entry)**<br>**bool isOn(const char *variable, const char *entry)** | | |
|---|---|---|
| Description: | get a single string entry | |
| IN: | entry | path to XML tag |
| IN: | variable | attribute, the value of which should be returned |
| Returns | value of configuration | |

| **const char **getScopeEntries(const char *scope))** | | |
|---|---|---|
| Description: | get all entries of one scope | |
| IN: | scope | name of scope |
| Returns | NULL-terminated array, alternating<br>$name_0$,$value_0$,$name_1$,$value_1$,...$value_n$,NULL | |

| **const char *getScopeEntry (const char *scope, const char *name)** | | |
|---|---|---|
| Description: | get a single entry | |
| IN: | scope | Scope to be looked in |
| IN: | name | Name of variable |
| Returns | value of scope.name | |

# 3 COVISE Data Objects

## 3.1 Available Data Object Types

Before the access for reading input objects is explained, we give in this section a short description of all available data object classes the programmer can use.

| Data type | Description |
|---|---|
| coDistributedObject | Base class for all data object types |
| coDoAbstractStructuredGrid | Abstract base class for structured grids |
| coDoUniformGrid<br>coDoRectilinearGrid<br>coDoStructuredGrid | Structured Grid Types |
| coDoUnstructuredGrid | Unstructured Grid Types |
| coDoCoordinates | Abstract base class for geometry data |
| coDoPoints<br>coDoLines<br>coDoTriangleStrips<br>coDoPolygons<br>coDoSpheres | Geometry data types |
| coDoFloat | Scalar data |
| coDoVec2<br>coDoVec3 | Vector data |
| coDoTensor | Tensor data |
| coDoRGBA | Packed colour values |
| coDoSet | Container type for grouping of data objects |
| coDoGeometry | Renderer geometry container data type (as created by Collect module) |

**Table 2: COVISE data types**

The data types have been designed to meet the requirements of common scientific visualization problems and are similar to data models of widespread visualization packages.

There are five main groups of data in the COVISE data model:

1. Structured grids: They have implicit connectivity information. Addressing is typically done by using 3 indices in x, y and z direction. The most simple structured grid is an equidistant orthogonal grid, i. e. a collection of cubes or bricks. Also non-equidistant and deformed structured grid types are supported.
2. Unstructured grids: They are constructed of primitive elements by explicit definition of the element vertices of each single element.
3. Geometric Objects: They contain geometry element descriptions in various formats, which can be displayed by a renderer. They can be combined with color, normal and texture information using the Collect module.
4. Data objects: They contain data elements, which usually reside either on unstructured or structured grids, but also the other geometric elements are possible.

5. Set Containers: They allow grouping of multiple objects, typically of the same kind. Thus, either multi-part objects or time-series of objects can be described. Containers can be used recursively.

Each of these groups can be divided in sub-groups as shown in the table above.

## 3.2 Two important warnings

***Distributed objects may only be used in the compute callback or functions called from it***

***Access to pointers into shared memory cannot be checked by the system, so writing out of bounds will cause the system to crash***

## 3.3 The Base Class for Distributed Objects

This class is the base class of all available data object classes. It offers some basic functionality to all object classes:

| **const char *getName()** | |
|---|---|
| Description: | get the name of an object |
| Return value: | object name |

| **const char *getType()** | |
|---|---|
| Description: | get the type of an object |
| Return value: | object type |

| **int isType(const char *reqType)** | |
|---|---|
| Description: | check whether a an object has a certain type |
| Return value: | object type |

| **char coDistributedObject::objectOk()** | |
|---|---|
| Description: | Check whether object was created correctly |
| COVISE states: | Compute |
| Return value: | NONZERO on success, =0 on error |

The coDistributedObject class is generally only used as the base class for all object classes. Direct usage of coDistributedObject class objects is usually not necessary, except for:

- receiving data at an input port: A port can deliver different kinds of objects, so it returns a pointer to the base class. The user can then request the type with getType and then safely up-cast it to the real data type.
- building/retrieving sets: The data container `coDoSet` can hold groups of arbitrary data objects, which are represented by base class pointers.

## 3.4 Object types

To identify the type of an object, the getType or isType functions can be used. This function works with a unique 6-digit string, documented in the following table

| Data Object Type | Type name |
|---|---|
| coDoUniformGrid | UNIGRD |
| coDoRectilinearGrid | RCTGRD |
| coDoStructuredGrid | STRGRD |
| coDoUnstructuredGrid | UNSGRD |
| coDoPoints | POINTS |
| coDoLines | LINES |
| coDoPolygons | POLYGN |
| coDoTriangleStrips | TRIANG |
| coDoSpheres | SPHERE |
| coDoFloat | USTSDT |
| coDoVec3 | USTVDT |
| coDoTensor | USTTDT |

**Table 2: COVISE Type Identifiers**

A typical code fragment from the beginning of the `compute()` callback function:

```
myApplication::compute(const char * /*port*/)     // ... module is executed
{
   // get the grid object from the port and check,
   // whether it was received at all
   coDistributedObject *gridObj = p_grid->getCurrentObject();
   if (!gridObj)
   {
      sendError("Did not receive object at port %s",
                p_grid->getName());
      return;
   }
```

```
   // if this is an unstructured grid...
   if ( coDoUnstructuredGrid *unsGrd = dynamic_cast<coDoUnstructuredGrid *>(gridObj) )
   {
      ... handle unstructured grid
   }

   // it is not an unstructured grid, maybe a structured grid...
   else if ( coDoStructuredGrid *strGrd = dynamic_cast<coDoStructuredGrid *>(gridObj) )
   {
      ... handle structured grid
   }

   // if it is not a case we can handle, we have to submit an error
   else
   {
      sendError("Illegal object type at port %s : %s",
                              p_grid->getName(), grid->get_type());
      return;
   }
}
```

The code fragment first checks whether an object was created and then forks into different branches dependent on the received object type. dynamic_cast is used to check if an object is of a certain type and to up-cast the object pointer delivered by the port.

## 3.5 Attributes

Object attributes can be added to all objects derived from the coDistributedObject class. Attributes are only meaningful to modules which know about the existence of a certain attribute. Attributes can be used to pack additional information (only character data is possible) into an data object. Attributes are created by:

| **addAttribute(const char *name, const char *value)** | | |
|---|---|---|
| Description: | attach a single attribute to an object | |
| IN: | name | attribute name |
| Return | attribute value | |

| **addAttributes (int num, const char * const* names, const char * const* values)** | | |
|---|---|---|
| Description: | attach multiple attributes to an object | |
| IN: | name | attribute name |
| IN: | values | attribute values |
| Return | attribute value | |

The difficult-looking ‘const’ declarations are necessary to allow constant fields to be assigned to this argument. Any two-dimensional char array or list pointer to a list of char pointers can be assigned to this parameter.

This is how attributes are retrieved from an object:

| **const char *getAttribute(const char *name)** | | |
|---|---|---|
| Description: | retrieve attribute values | |
| IN: | name | attribute name |
| Return | attribute value, NULL if attribute does not exist | |

| **int getAllAttributes(const char ***names, const char ***content)** | | |
|---|---|---|
| Description: | retrieve all attribute values | |
| OUT: | names | attribute name |
| OUT: | contents | attribute contents |

| **void copyAllAttributes(coDistributedObject *src)** | | |
|---|---|---|
| Description: | copy all attribute values of object 'src' to this object | |
| IN: | src | the source distributed object |

## 3.6 Structured Grid Types

A structured grid can be either a regular orthogonal grid with constant spacing (coDoUniformGrid) or a regular orthogonal grid with variable spacing (coDoRectilinerGrid) or a curvilinear grid (coDoStructuredGrid). Typical for all these grids is, that the grid is constructed by intersecting lines. Furthermore it is not necessary to have a special connection list of the neighbouring points (contrary to unstructured grids) because each point can be addressed by using the indices of the computational domain.

### 3.6.1 Abstract Structured Grid Base Class

For easier handling of all these similiar structured grid types, they are derived from the common abstract base class coDoGeneralStructuredGrid, which provides the following methods:

| **void coDoGeneralStructuredGrid::getGridSize (int *x_size, int *y_size, int *z_size)** | | |
|---|---|---|
| Description: | get the grid dimensions | |
| COVISE states: | Compute | |
| IN: | {x\|y\|z}_size | ptr to variables in which dimensions should be stored |

| **void getPointCoordinates (int i, float *x_coord, int j, float *y_coord, int k, float *z_coord)** | | |
|---|---|---|
| Description: | get the coordinates of a single grid point | |
| COVISE states: | Compute | |
| IN: | i,j,k | coordinate index in {x\|y\|z} direction |
| OUT: | {x\|y\|z}_ccord | ptr to variables in which cordinates should be stored |

### 3.6.2 Uniform Grid

A uniform grid is described by 3 sets of equidistantly split coordinate ranges. The interpretation of a uniform grid may vary with the used coordinate system. COVISE currently only supports Cartesian coordinates, so any uniform grid handed over to a COVISE module is interpreted as shown in the following drawing:

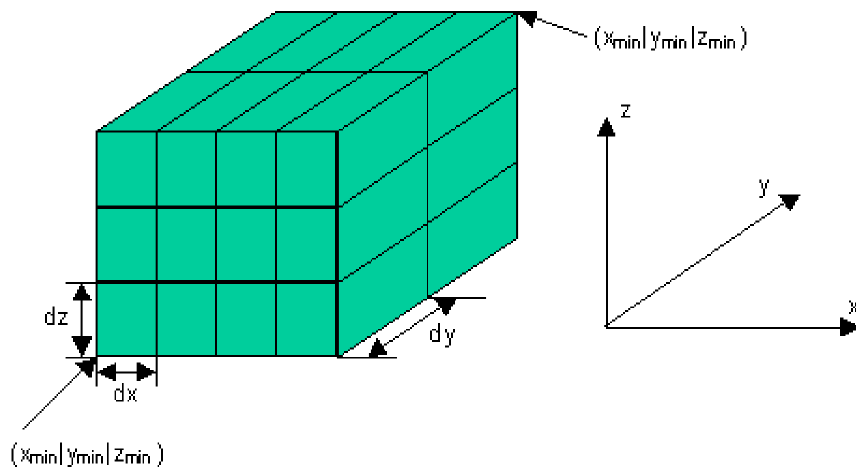



**Figure 3.1: Uniform Grid**

Uniform grids are seldom, since the spatial resolution is constant, while most real-world applications require varying spatial resolutions.

To create a new uniform grid object in shared memory, the constructor is used:

| **coDoUniformGrid(const char *name, int x_size, int y_size, int z_size, float x_min, float x_max, float y_min, float y_max, float z_min, float z_max** | | |
|---|---|---|
| Description: | create a new uniform grid by number of steps | |
| COVISE states: | Compute | |
| IN: | name | new object name |
| IN: | {x\|y\|z}_size | grid dimensions in {x\|y\|z} direction |
| IN: | {x\|y\|z}_{min/max} | minimum/maximum coordinate in {x\|y\|z} direction |

The new object name is usually taken from the output port's `getObjName()` function, except for building part objects in a `coDoSet` container, as explained later.

Earlier API versions also contained a constructor, which retrieved the object from shared memory. This constructor is not necessary any more, since the `getCurrentObject()` method of the input port automatically retrieves all objects and hands over an object pointer.

The following functions extract the information of a received uniform grid:

| **void coDoUniformGrid::getGridSize (int *x_size, int *y_size, int *z_size)** | | |
|---|---|---|
| Description: | get the grid dimensions | |
| COVISE states: | Compute | |
| IN: | {x\|y\|z}_size | ptr to variables in which dimensions should be stored |

| **void getPointCoordinates (int i, float *x_coord, int j, float *y_coord, int k, float *z_coord)** | | |
|---|---|---|
| Description: | get the coordinates of a single grid point | |
| COVISE states: | Compute | |
| IN: | i,j,k | coordinate index in {x\|y\|z} direc- |

| **void getMinMax( float *x_min, float *x_max, float *y_min, float *y_max, float *z_min, float *z_max)** | | |
|---|---|---|
| Description: | get the coordinates of a single grid point | |
| IN: | {x\|y\|z}_{min/max} | minimum/maximum coordinate in {x\|y\|z} direction |

### 3.6.3 Rectilinear Grid

A rectilinear grid is similar to a uniform grid, except that the spacing along each coordinate axis is non-uniform. The coordinates are the coordinates on the x, y and z-axis.

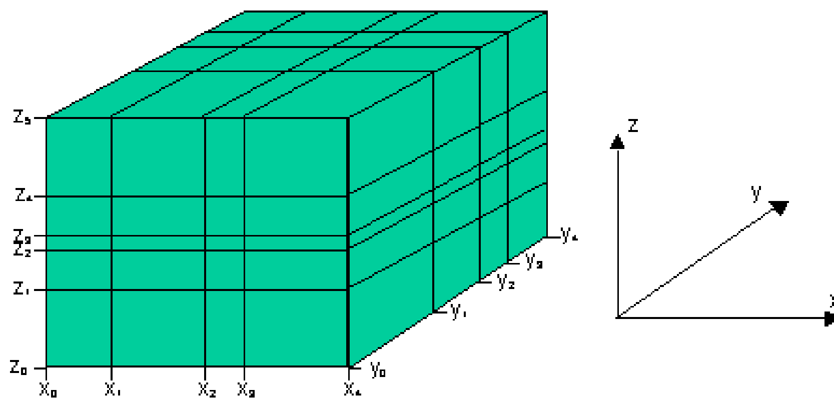


Figure 3.2: Rectilinear Grid

A new rectilinear grid can be constructed using two different constructors :

| **coDoRectilinearGrid(const char *name, int x_size, int y_size, int z_size)** | | |
|---|---|---|
| Description: | Create a rectilinear grid | |
| IN: | name | name of rectilinear grid object |
| IN: | {x\|y\|z}_size | dimension in {x\|y\|z}-direction |

This is the usual constructor: it allocates the fields in shared memory but does not set any values. The user can then request pointers to the fields in shared memory and create the coordinate fields directly without need for an additional copy in the local memory space.

| **coDoRectilinearGrid(const char *name, int x, int y, int z, float *x_coord, float *y_coord, float *z_coord)** | | |
|---|---|---|
| Description: | Create a rectilinear grid with given set of coords | |
| Parameters | name: | name of rectilinear grid object |
| IN: | {x\|y\|z} | size in {x\|y\|z}-direction |
| IN: | {x\|y\|z}_coord | cordinates in {x\|y\|z}-direction |

If the field of coordinates exists in memory, the object can be created in shared memory. This is exactly like creating the object, retrieving the field pointers and copying the three coordinate fields.

The following routines give information about a rectangular grid object:

| **void getGridSize<br>(int *x_dim, int *y_dim, int *z_dim)** | | |
|---|---|---|
| Description: | Get dimensions of the grid | |
| OUT: | {x\|y\|z}_dim | Pointer to int variables to return {x\|y\|z}-sizes at index i |

| **void getPointCoordinates<br>(int i, float *x_coord, int j, float *y_coord, int k, float *z_coord)** | | |
|---|---|---|
| Description: | get the coordinates of a single grid point | |
| IN: | {i\|j\|k} | location in {x\|y\|z} direction |
| OUT: | {x\|y\|z}_coord | Pointer to float variables to return {x\|y\|z}-cordinates at index {i\|j\|k} |

The last function gives direct access to the shared memory storage for the coordinate arrays. *Warning: coordinate indices are not checked, so writing beyond the bounds of the allocated array usually crashes the system.*

| **void getAddresses(float **x_start, float **y_start, float **z_start)** | | |
|---|---|---|
| Description: | Get coordinate array pointers | |
| OUT: | {x\|y\|z}_start | Pointer to (float *) variable, which is set to start of {x\|y\|z} coordinate field |

### 3.6.4 Structured Grid

A structured grid is an arbitrarily deformed hexahedral grid, which still has a primitive structure of i x j x k hexahedra. All vertex coordinates are stored independently, but the connectivity is still implicit.

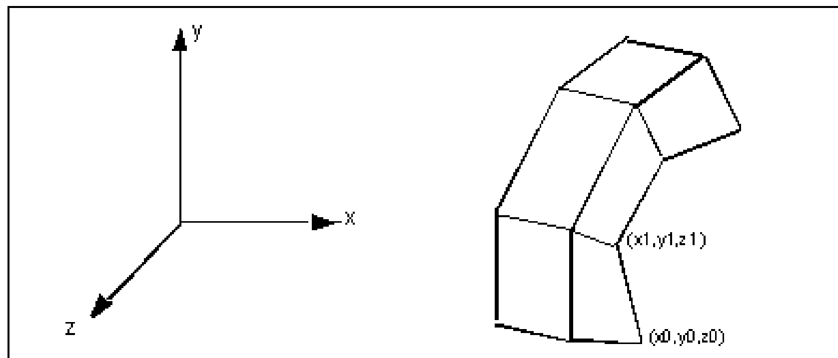



**Figure 3.3: Structured Grid**

Like the rectilinear grid, the structured grid can be constructed with or without pre-setting the coordinate arrays:

| coDoStructuredGrid(const char *name, int x_nv, int y_nv, int z_nv) | | |
|---|---|---|
| Description: | Create a structured grid | |
| IN: | name | name of structured grid object |
| IN: | {x\|y\|z}_nv | number of vertices in {x\|y\|z}-direction |

| coDoStructuredGrid(char *name, int x_nv, int y_nv, int z_nv float *x_coord, float *y_coord, float *z_coord) | | |
|---|---|---|
| Description: | Create a structured grid | |
| Parameters | name: | name of structured grid object |
| IN: | {x\|y\|z}_coord | {x\|y\|z}-coordinates (Field size: ixjxk) |
| IN: | {x\|y\|z}_nv | number of vertices in {x\|y\|z}-direction |

The size and the sequence of the 3 coordinate arrays is arr[x_nv][y_nv][z_nv].

All other functions are equivalent to the methods for rectilinear grids:

| void getPointCoordinates (int i, float *x_coord, int j, float *y_coord, int k, float *z_coord) | | |
|---|---|---|
| Description: | get the coordinates of selected point | |
| IN: | {i\|j\|k} | index in {x\|y\|z}-direction |
| OUT: | {x\|y\|z}_coord | Pointer to variable for requested cordinate |

| void getGridSize(int *x_dim, int *y_dim, int *z_dim) | | |
|---|---|---|
| Description: | get field dimensions | |
| OUT: | {x\|y\|z}_dim | Pointer to variable for size in {x\|y\|z}-direction |

| void getAddresses(float **x_start, float **y_start, float **z_start) | | |
|---|---|---|
| Description | get coordinate array pointers | |
| OUT: | {x\|y\|z}_start | Pointer to (float *) var to return starting address of vertex {x\|y\|z} coordinate array |

COVISE's structured grid type corresponds to "fields" in AVS which covers "uniform", "rectilinear" and "irregular" "AVS-grids". Referring to the IBM Data Explorer the structured grid type corresponds to "regular", "deformed regular", and "irregular" grids.

## 3.7 Unstructured Grid Types

Unstructured grids are grids composed of explicitly described basic elements. Such kind of grids is often used in Computational Fluid Dynamics (CFD) and Structural Analysis with Finite Elements Methods (FEM).

The basic elements for COVISE's unstructured grids are:

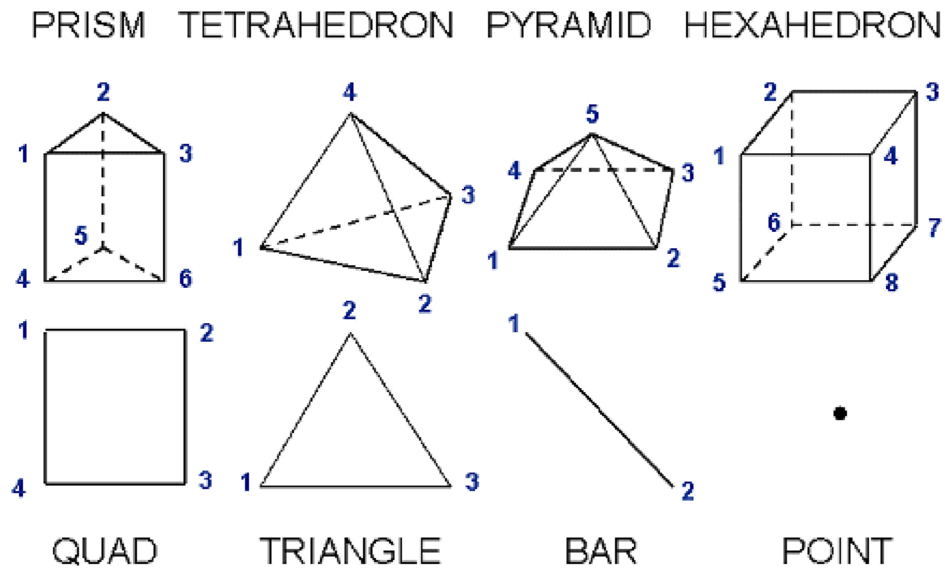



**Figure 3.4: Unstructured Grid Base Element**

All unstructured grids are collections of these basic elements. To describe the grid, multiple lists are maintained:

- **Type list**: It contains the type of a certain element. This cannot be determined implicitly, since tetrahedra and quads have the same number of vertices.
- **Element list**: It contains the index into the connectivity list, where the description of a certain element begins.
- **Connectivity list**: It lists the indices of the vertices belonging to each element.
- **X|Y|Z Coordinate list**: It contains the vertex coordinates.

If the type list does not exist, the types are assumed by the relation between the sizes of the connectivity and the element list. Not all modules correctly implement the non-typed unstructured grids, so using these is strongly discouraged.

The correlation of the different lists is shown in the following drawing:

An example for an unstructured Grid with three cells:

An unstructured grid is usually constructed using the following constructor :

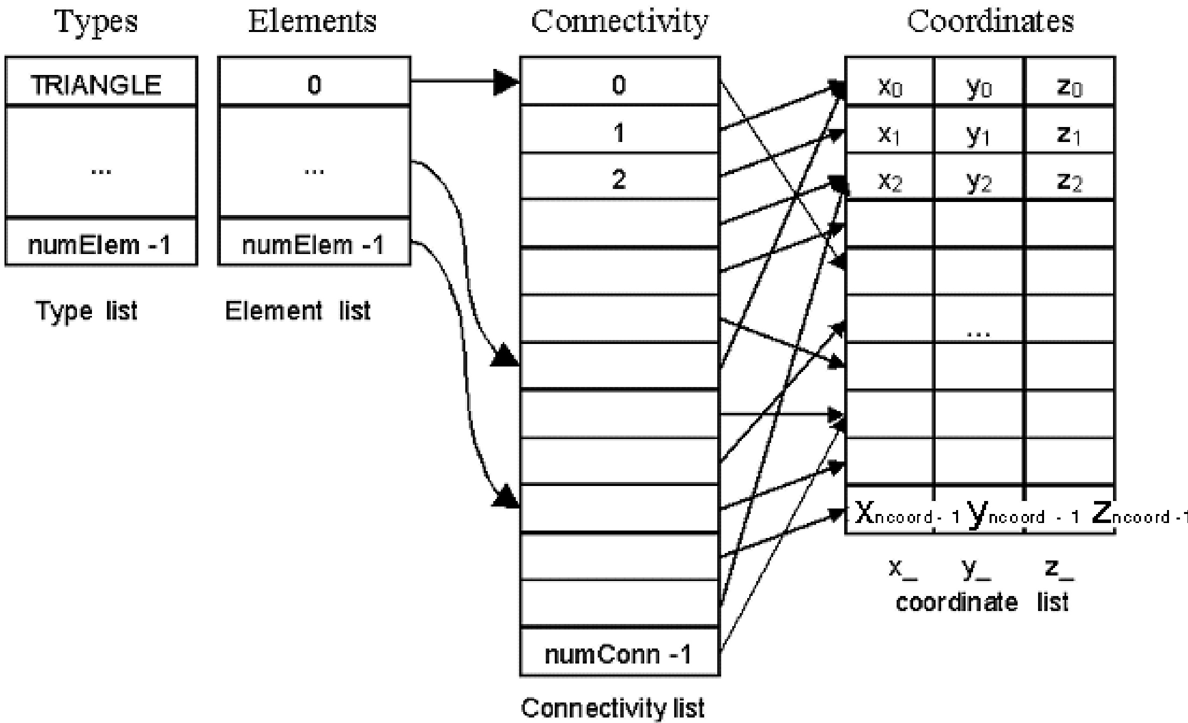



Figure 3.5: Unstructured Grid format

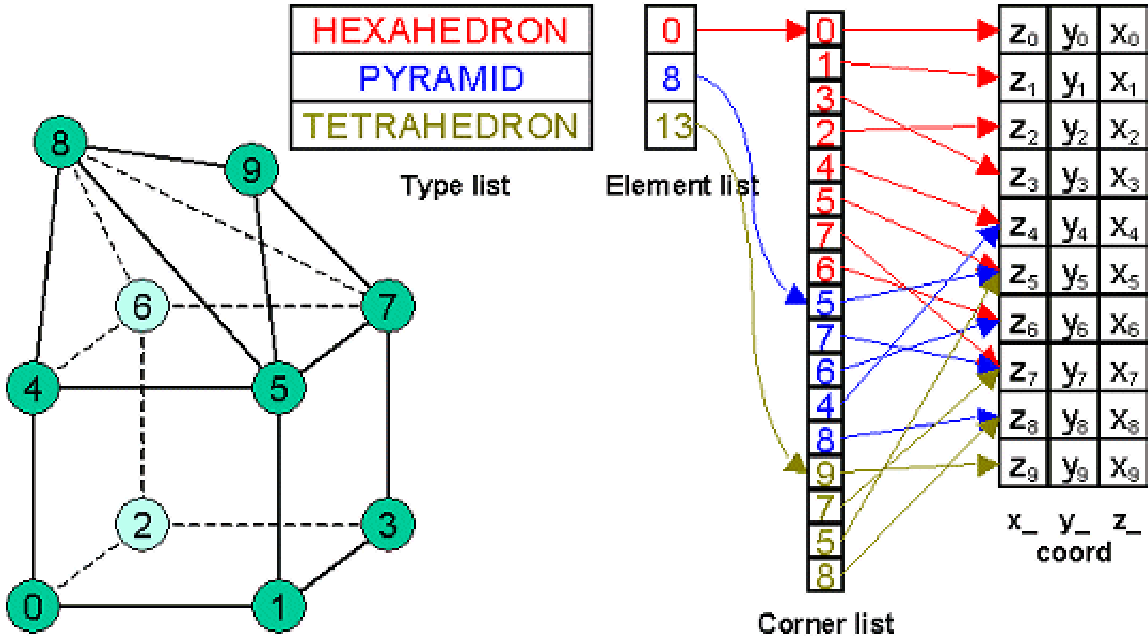



Figure 3.6: Unstructured Grid example

| **coDoUnstructuredGrid<br>(const char *name, int nelem, int nconn, int ncoord, int ht)** | | |
|---|---|---|
| Description: | Create an unstructured grid | |
| IN: | name | name of USG object |
| IN: | nelem | number of elements |
| IN: | nconn | number of connectivities |
| IN: | ncoord | number of coordinates |
| IN: | ht | whether the type list exists |

After constructing the object, pointers to the arrays are retrieved using the `getAddresses` method and the lists are then filled directly in shared memory.

The other constructor allows to create the object with already prepared fields. This is usually a waste of memory, since a copy has to be held which is not necessary if the USG is allocated before.

| **coDoUnstructuredGrid(const char *name,<br>int nelem, int nconn, int ncoord, int *el, int *cl,<br>float *x_coord, float *y_coord, float *z_coord,<br>int *tl)** | | |
|---|---|---|
| Description: | Create an unstructured grid | |
| IN: | name | name of USG object |
| IN: | nelem | number of elements |
| IN: | nconn | number of connectivities |
| IN: | ncoord | number of coordinates |
| IN: | el | element list |
| IN: | cl | coordinate list |
| IN: | tl | type list |
| IN: | {x\|y\|z}_coord | array of {x\|y\|z} coordinates |

There are additional constructors for unstructured grids, which can be found in the header file, but they will not be supported for future use.

The following functions retrieve all information from the USG and give access to its internal fields.

| **void getGridSize(int *numEl, int *numConn, int *numCoord)** | | |
|---|---|---|
| Description: | get field dimensions | |
| OUT: | numEl | number of elements |
| OUT: | numConn | number of connectivity list entries |
| OUT: | numCoord | number of coordinates |

| **void getAddresses(int **elem, int **conn,<br>float **x_coord, float **y_coord, float **z_coord)** | | |
|---|---|---|
| Description: | Get pointers to internal lists of USG type | |
| OUT: | elem | element list |
| OUT: | conn | connectivity list |
| OUT: | {x\|y\|z}_list | {x\|y\|z}-coordinate arrays |

| **void getTypeList(int **tList)** | | |
|---|---|---|
| Description: | Get pointers to internal lists od USG type | |
| OUT: | tList | type list |

| **int hasTypeList()** | |
|---|---|
| Description: | Check whether a type list exists |
| Return value: | =0 : no typelist, else typelist exists |

The following command creates a coordinate-to-cell mapping. *Caution: The fields allocated by this procedure have to be deleted by the user with* `delete []` *command.*

| **void getNeighborList(int *n, int **elemList, int **vStart)** | | |
|---|---|---|
| Description: | get backward mapping | |
| OUT: | n | number of entries in elemList |
| OUT: | elemList | [n] : elements containing specific vertex |
| OUT: | vStart | [numvert+1] :<br>Entries in elemList start at<br>vStart[vertexNo] and end<br>before vStart[vertexNo+1] |

E.g.the cells containing Point #7 are `*elemList[vStart[7]...(vStart[8]-1)]`

Example:

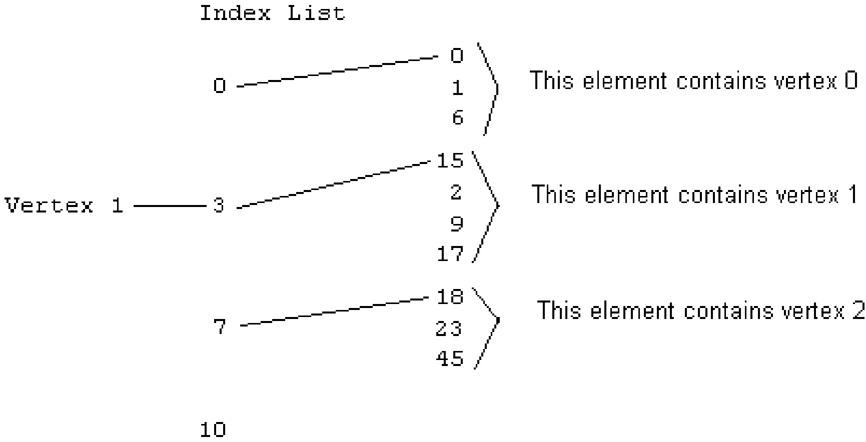



**Figure 3.7: Element Neighbor List**

## 3.8 Data Types

The mapping between computational and coordinate space is either direct and implicit or explicit. In COVISE explicit data mapping has been chosen. That means in the three dimensional case, all three coordinates at each grid point are stored plus possibly additional scalar values at each of these grid points. A vector, e.g. a three dimensional vector, will be treated and stored as three scalars at each grid point.

There are several different data types available: scalar data, 2D and 3D vector data, tensor data, and packed RGBA data. All are quite similar, except that the 2D and 3D vector types contain 2 or 3 arrays respectively instead of 1 array. The splitting into several different fields instead of 1 large field with x-y-z packed data allows larger fields on systems with restricted shared memory chunk sizes.

Data contains no information of the underlying grid. Instead, a linear array is saved.

The user must compare the number of elements in the list, which can be requested in all sub-types of unstructured data, and then compare against the number of cells and coordinates to decide, whether cell- or point-based data was given.

### 3.8.1 Scalar Data

| **coDoFloat(const char *name, int num_values)** | | |
|---|---|---|
| **coDoFloat(const char *name, int num_values, float *scalar_data)** | | |
| Description: | Create an object for scalar data without and with setting elements | |
| IN: | name | name of data object |
| IN: | num_values | length of data array |
| IN: | scalar_data | data values |

As for all other objects, it is recommended to create an empty object and fill in data to prevent double storage.

| **void getAddress(float **start)** | | |
|---|---|---|
| Description: | get pointer to object data | |
| OUT: | start | pointer to starting address |

| **void getPointValue(int pos, float *value)** | | |
|---|---|---|
| Description: | get single data value | |
| IN: | pos | index |
| OUT: | value | data value |

| **int getNumPoints()** | |
|---|---|
| Description: | get number of data values saved |
| Return value: | number of elements |

### 3.8.2 2D Vector Data

<table>
<tr><td colspan="3">coDoVec2(const char *name, int num_values)</td></tr>
<tr><td colspan="3">coDoVec2(const char *name, int num_values,<br>float *data1, float *data2)</td></tr>
<tr><td>Description:</td><td colspan="2">Create an object for 2D vector data without and with setting elements</td></tr>
<tr><td>IN:</td><td>name</td><td>name of data object</td></tr>
<tr><td>IN:</td><td>num_values</td><td>length of data array</td></tr>
<tr><td>IN:</td><td>data{1|2}</td><td>data value arrays</td></tr>
</table>

<table>
<tr><td colspan="3">void getAddresses(float **data1, float **data2)</td></tr>
<tr><td>Description:</td><td colspan="2">get pointers to object data fields</td></tr>
<tr><td>OUT:</td><td>data{1|2}</td><td>pointer to starting address in shared memory</td></tr>
</table>

<table>
<tr><td colspan="3">void getPointValue(int pos, float *val1, float *val2)</td></tr>
<tr><td>Description:</td><td colspan="2">get single data value</td></tr>
<tr><td>IN:</td><td>pos</td><td>index</td></tr>
<tr><td>OUT:</td><td>val{1|2}</td><td>data values</td></tr>
</table>

### 3.8.3 3D Vector Data

<table>
<tr><td colspan="3">coDoVec3(const char *name, int num_values)</td></tr>
<tr><td colspan="3">coDoVec3(const char *name, int num_values,<br>float *data1, float *data2, float *data2)</td></tr>
<tr><td>Description:</td><td colspan="2">Create an object for 3D vector data without and with setting elements</td></tr>
<tr><td>IN:</td><td>name</td><td>name of data object</td></tr>
<tr><td>IN:</td><td>num_values</td><td>length of data array</td></tr>
<tr><td>IN:</td><td>data{1|2|3}</td><td>data value arrays</td></tr>
</table>

<table>
<tr><td colspan="3">void getAddresses(float **data1, float **data2, float **data3)</td></tr>
<tr><td>Description:</td><td colspan="2">get pointers to object data fields</td></tr>
<tr><td>OUT:</td><td>data{1|2|3}</td><td>pointer to starting address in shared memory</td></tr>
</table>

<table>
<tr><td colspan="3">void getPointValue(int pos, float *val1, float *val2, float *val3)</td></tr>
<tr><td>Description:</td><td colspan="2">get single data value</td></tr>
<tr><td>IN:</td><td>pos</td><td>index</td></tr>
<tr><td>OUT:</td><td>val{1|2|3}</td><td>data values</td></tr>
</table>

### 3.8.4 Tensor Data

<table>
<tr><td colspan="3">coDoTensor(const char *name, int num_values, TensorType ttype)</td></tr>
<tr><td colspan="3">coDoTensor(const char *name, int num_values, float *tensor_data, TensorType ttype)</td></tr>
<tr><td>Description:</td><td colspan="2">Create an object for tensor data without and with setting elements</td></tr>
<tr><td>IN:</td><td>name</td><td>name of data object</td></tr>
<tr><td>IN:</td><td>num_values</td><td>length of data array</td></tr>
<tr><td>IN:</td><td>tensor_data</td><td>data values</td></tr>
<tr><td>IN:</td><td>ttype</td><td>kind of tensor</td></tr>
</table>

As for all other objects, it is recommended to create an empty object and fill in data to prevent double storage. The kinds of tensors are summarised here:

| Tensor type | Space dimension | Sequence of independent components |
|---|---|---|
| S2D: | 2 | XX, YY, XY |
| F2D: | 2 | XX, XY, YX, YY |
| S3D: | 3 | XX, YY, ZZ, XY, YZ, ZX |
| F3D: | 3 | XX, XY, XZ, YX, YY, YZ, ZX, ZY, ZZ |

The numerical information for this object is kept in a single array with as many floats as the number of tensors (number of points in space for which a tensor is described) multiplied by the number of independent components for the tensor type at issue.

| **void getAddress(float **start)** | | |
|---|---|---|
| Description: | get pointer to object data | |
| OUT: | start | pointer to starting address |

| **void getPointValue(int pos, float *value)** | | |
|---|---|---|
| Description: | get tensor components for an index | |
| IN: | pos | index |
| OUT: | value | array with tensor components |

| **int getNumPoints()** | |
|---|---|
| Description: | get number of data values saved |
| Return value: | number of tensor elements |

| **coDoTensor::TensorType getTensorType()** | |
|---|---|
| Description: | get tensor type |
| Return value: | Either UNKNOWN, S2D, F2D, S3D or F3D |

### 3.8.5 Packed RGBA Data

Packed RGBA data consists of one 4-byte word per value, containing RGBA color and opacity values.

| **coDoRGBA (const char *name, int num_values)** | | |
|---|---|---|
| **coDoRGBA (const char *name, int num_values, int *packedColors)** | | |
| Description: | Create an object for packed RGBA values | |
| IN: | name | name of data object |
| IN: | num_values | length of data array |
| IN: | packedColors | data array |

| **void getAddress(int **color_field)** | | |
|---|---|---|
| Description: | get pointers to object data fields | |
| OUT: | color_field | pointer to starting address |

| void getPointValue(int pos, int *value) | | |
|---|---|---|
| Description: | access single element | |
| IN: | pos | index |
| OUT: | value | RGBA value casted to int |

| int setFloatRGBA(int pos, float r, float g, float b, float alpha) | | |
|---|---|---|
| int setIntRGBA(int pos, int r, int g, int b, int alpha) | | |
| Description: | set a single value | |
| IN: | pos | - |
| IN: | r,g,b | RGBA float [0..1] or int [0..255] |

| int getFloatRGBA(int pos, float *r, float *g, float *b, float *alpha) | | |
|---|---|---|
| int getIntRGBA(int pos, int *r, int *g, int *b, int *alpha) | | |
| Description: | get a single value | |
| IN: | pos | - |
| OUT: | r,g,b | RGBA value , either float [0..1] or int [0..255] |
| Return value: | nothing (always 1) | |

## 3.9 Geometry data types

Geometry data is the kind of data that can be displayed in Renderers. It describes a geometrical object consisting of points and connectivity information of the geometrical primitives.

### 3.9.1 Geometry Container Class

The additional container object type `coDoGeometry` is used to combine colors, normals and textures into geometrical data types. Usually, this is done with the "Collect" module, which automatically combines all its input objects into the `coDoGeometry` container.

Users should normally not create `coDoGeometry` objects but instead create the single part objects and then let the "Collect" module combine it. This allows to add filters, e.g. cropping or simplification, to be attached after the module.

Each coDoGeometry container must contain a geometrical object, while all other parts of the container are optional.

| coDoGeometry (const char *name, coDistributedObject *geometry_object) | | |
|---|---|---|
| Description: | Create a container around the given geometrical object | |
| IN: | name | object name |
| IN: | geometry_object | geometrical object |

This routine attaches the given object (not a copy) to the Container. See under ‘Advanced features’ what to do for re-using received data objects.

All other parts of the Container are volatile and can be attached to the container after its creation. There are no default objects, so if nothing is attached, a null object will be returned.

These methods attach additional objects to the container:

| **void setColors(int attrib, coDistributedObject *colObj** | | |
|---|---|---|
| **void setNormals(int attrib, coDistributedObject *normObj)** | | |
| Description: | set a single value | |
| IN: | attrib | OVERALL, PER_FACE or PER_VERTEX how data is to be applied to geometrical objects |
| IN: | colObj | Colors: Either a RGAB object containing packed colors or a structured or vector data field containing RGB valued [0..1] |
| IN: | normObj | Normals: Either a structured or vector data field containing normal vectors |

| **void coDoGeometry::setTexture(int attrib, coDistributedObject *object)** | | |
|---|---|---|
| Description: | attach colors and normals to a geometry container | |
| IN: | attrib | for future use |
| IN: | object | Texture object containing texture |

The part objects can be retrieved from the container using:

| **coDistributedObject *getGeometry()** | |
|---|---|
| **coDistributedObject *getColors()** | |
| **coDistributedObject *getNormals()** | |
| **coDistributedObject *getTexture()** | |
| Description: | get the geometry object from the container |
| Return value: | pointer to requested object |

The attributes of the part objects are retrieved using:

| **int getColorAttributes() int getNormalAttributes() int getTextureAttributes()** | |
|---|---|
| Description: | get object attributes |
| Return value: | attribute |

### 3.9.2 Points

A point object is simply a list of triples of data in 3D space. The representation in the renderer is a single point. To emphasize points, the "Sphere" module can convert points to spheres consisting of multiple triangles together with the appropriate normals or to sphere impostors.

| **coDoPoints(const char *name, int numPoints)** | | |
|---|---|---|
| **coDoPoints(const char *name, int numPoints, float *x_coord, float *y_coord, float *z_coord)** | | |
| Description: | create a point object without or with initial values | |
| IN: | name | object name |
| IN: | numPoints | number of points |
| IN: | {x\|y\|z}_coord | point 3D coordinates |

| **void getAddresses(float **x_start, float **y_start, float**z_start)** | | |
|---|---|---|
| Description: | get access to coordinate fields in shared memory | |
| OUT: | {x\|y\|z}_start | Start of {x\|y\|z} cordinates field |

| int getNumPoints() | |
|---|---|
| Description: | get number of points in object |
| Return value: | number of points |

| void getPointValue(int pos, float &value[3]) | | |
|---|---|---|
| Description: | get single point | |
| IN: | pos | point index |
| OUT: | value | point coordinates |

Point objects are used e.g. for particle positions.

An attribute "POINTSIZE" can be attached to the object : a value of "0" results in the default pointsize in the renderer, while higher values than "1" can result in larger display points, if the hardware and the renderer support this.

### 3.9.3 Lines

A coDoLines object holds the points and the connectivity information for a set of line segments in 3D space.

The storage format of coDoLines is similar to coDoUnstructuredGrid. The connectivity information is stored in two lists: the first list holds the indices of the points that belong to each line segment, while the second list holds the index in the first list, where each segment starts.

This could also be done in one list by ending each segment with an index of -1, but the amount of storage needed for this would be nearly the same. Additionally the access to a line segment in the middle of the list will be much faster in the two-list-case.

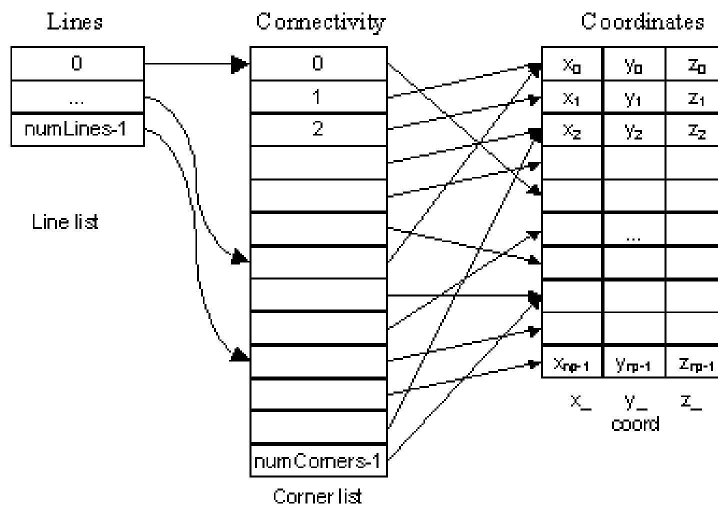



**Figure 3.8: Lists for defining a line structure**

To better understand the structure, an example is given, which constructs a coDoLines object with 3 lines using 10 points for 14 corners:

The resulting lists would be:

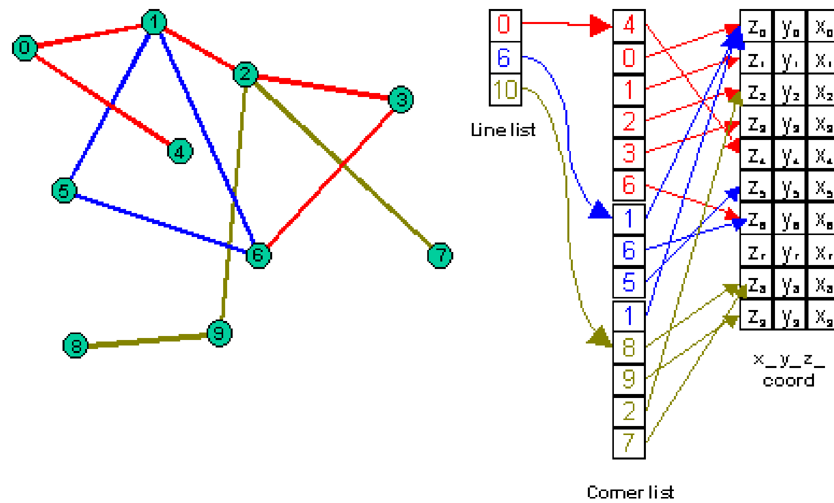



**Figure 3.9: Example for Lines**

| | | | | |
|---|---|---|---|---|
| num_lines | = 3 | Line list: | [ 0 6 10 ] | |
| num_corners | = 13 | Corner list: | [ 4 0 1 2 3 6 1 6 5 1 8 9 2 7 ] | |
| num_coord | = 10 | x_coord | [ $x_0$, $x_1$, ...$x_9$] | Point x-coordinates |
| | | y_coord | [ $y_0$, $y_1$, ...$y_9$] | Point y-coordinates |
| | | x_coord | [ $z_0$, $z_1$, ...$z_9$] | Point z-coordinates |

Notice that a closed line is generated by repeating the first point index at the end.

As for the unstructured grid, it is recommended to construct an empty Lines object of the appropriate size and then fill the list directly into shared memory. This both avoids double storage and the copying time.

The constructors for Lines objects are:

| **coDoLines(const char *name, int num_points, int num_corners, int num_lines)** | | |
|---|---|---|
| Description: | Construct a Lines object | |
| IN: | name | object name |
| IN: | num_points | Number of points used for the lines |
| IN: | num_corners | Sum of corners of all lines, including start and end point |
| IN: | num_lines | Number of distinct lines |

| **coDoLines(const char *name, int num_points, float *x_c, float *y_c, float *z_c, int num_corners, int *corner_list, int num_lines, int *line_list )** | | |
|---|---|---|
| Description: | Construct a Lines object and copy data from given fields | |
| IN: | name | object name |
| IN: | num_points | Number of points used for the lines |
| IN: | num_corners | sum of corners of all lines, including start and end point |
| IN: | num_lines | Number of distinct lines |
| IN: | {x\|y\|z}_c | point coordinates |
| IN: | corner_list | corner list |
| IN: | line_list | line list |

Access to the fields of the object is given by the following call:

*Caution: boundaries are not checked!*

| int getNumPoints() | |
|---|---|
| Description: | get number of points (length of coordinate arrays) |
| Return value: | number of points |

| int getNumVertices() | |
|---|---|
| Description: | get number corners (length of corner list) |
| Return value: | number of corners |

| int getNumLines() | |
|---|---|
| Description: | get number of lines (length of line list) |
| Return value: | number of lines |

### 3.9.4 Polygons

Polygonal objects are similar to lines, except that every set of corners which used to define a line now defines a closed polygon. Other than lines,polygons are closed automatically, so the last point is automatically connected with the first one.

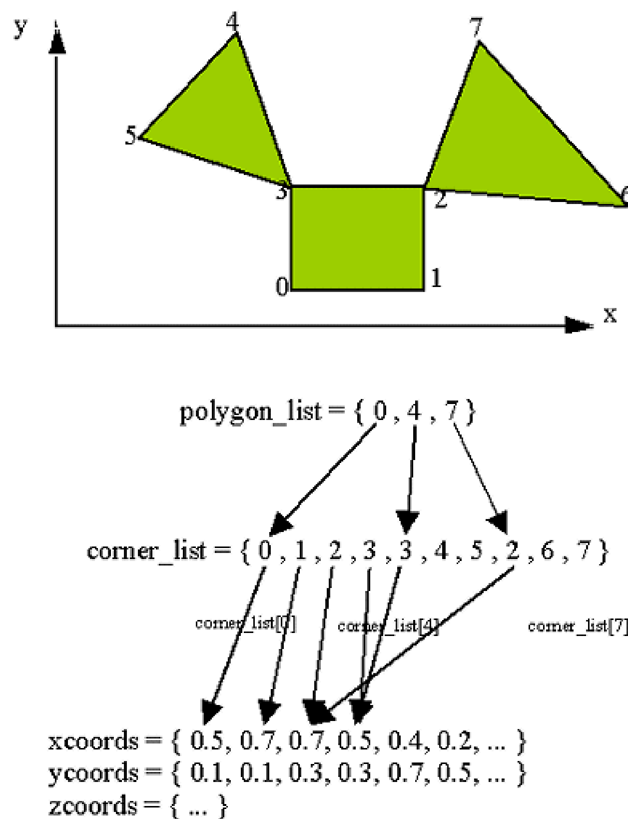



**Figure 3.10: Polygon Example**

| **coDoPolygons(const char *name, int num_points, int num_corners, int num_polygons)** | | |
|---|---|---|
| Description: | Construct a Lines object | |
| IN: | name | object name |
| IN: | num_points | Number of points used for the lines |
| IN: | num_corners | Sum of corners of all lines, including start and end point |
| IN: | num_polygons | Number of distinct polygons |

| **void getAddresses(float **x_start, float **y_start, float **z_start, int **corner_list, int **polygon_list)** | | |
|---|---|---|
| Description: | access fields in shared memory | |
| OUT: | {x\|y\|z}_start | Pointer to coordinate field |
| OUT: | corner_list | Pointer to corner list |
| OUT: | polygon_list | Pointer to polygon list |

As described in previous cases, the recommended way to construct an object is to initialize it empty, get the field pointers and then fill it. Nevertheless, a copying constructor is available.

| **coDoPolygons(const char *name, int num_points, float *x_coord, float *y_coord, float *z_coord, int num_corners, int *corner_list, int num_polygons, int *polygon_list)** | | |
|---|---|---|
| Description: | Construct a coDoPolygons object and copy data from given fields | |
| IN: | name | object name |
| IN: | num_points | Number of points used for the lines |
| IN: | num_corners | Sum of corners of all lines, including start and end point |
| IN: | num_polygons | Number of distinct polygons |
| IN: | {x\|y\|z}_c | point coordinates |
| IN: | corner_list | corner list |
| IN: | polygon_list | polygon list |

Previous versions of this manual used the term "vertex" either for points within unstructured grids or for corners in geometry. To avoid confusion, this was changed. Still, the geometrical library calls use the term "vertex" for corners.

| **int getNumPoints()** | |
|---|---|
| Description: | get number of points (length of coordinate arrays) |
| Return value: | number of points |

| **int getNumVertices()** | |
|---|---|
| Description: | get number corners (length of corner list) |
| Return value: | number of corners |

| **int getNumPolygons()** | |
|---|---|
| Description: | get number of polygons (length of polygon list) |
| Return value: | number of polygons |

| lp14cml | | **get_neighbor_list(int *** |
|---|---|---|
| Description: | returns list of polygons that have the same point in common | |
| OUT: | n | number of neigbors |
| OUT: | polygon_list[numconn] | polygons that have one point in common |
| OUT: | index_list[numccod+1] | begin of next group of polygons that have one point in common |

Example:

Polygons of vertex i: lnl[(lnli[i]...lnli[i+1]-1)]

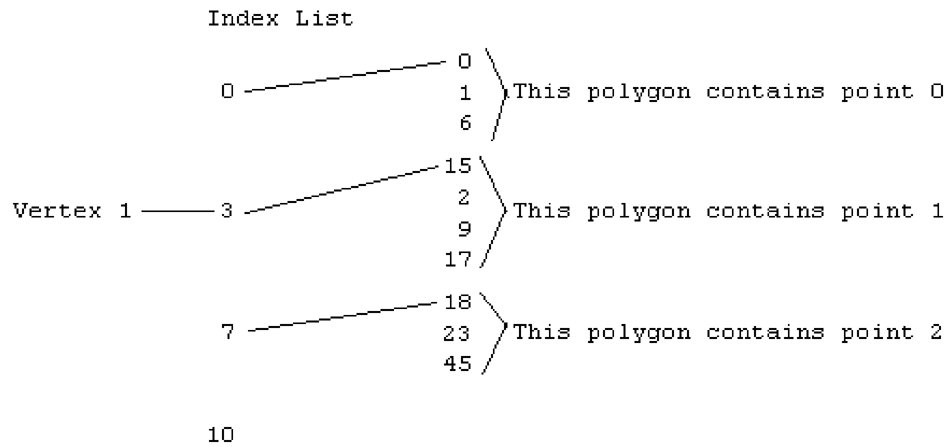



**Figure 3.11: Polygon Neighbor List**

### 3.9.5 Triangle Strips

Triangle strips are a special case of polygons and are rendered very efficiently on hardware accelerated graphics. Nevertheless, PER_VERTEX coloring is not available and leads to strange colored objects. A triangle strip is stored similar to lines and polygons, except that a line/polygon is now interpreted as a triangle strip.

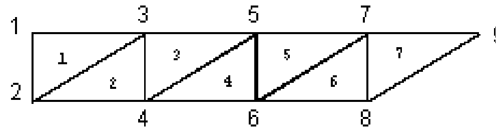



**Figure 3.12: Single triangle strip**

Just as the coDoPolygon object contains multiple polygons, the coDoTriangleStrips object contains multiple strips, with a strip list pointing into a corner list which holds the corner values.

| **coDoTriangleStrips(const char *name,<br>int num_points, int num_corners, int num_strips)** | | |
|---|---|---|
| Description: | Create a Triangle Strip Object | |
| IN: | name | name of triangle strips object |
| IN: | num_points | number of coordinates (points) |
| IN: | num_corners | length of corner list |
| IN: | num_strips | length of strip list |

| **coDoTriangleStrips(const char *name,<br>int num_points, float *x_c, float *y_c, float *z_c,<br>int num_corners, int *corner_list,<br>int num_strips, int *strips_list )** | | |
|---|---|---|
| Description: | Create a Triangle Strip Object | |
| IN: | name, num_* | as above |
| IN: | {x\|y\|z}_c | coordinates array |
| IN: | corner_list | corner list |
| IN: | strips_list | strip list |

The functions `getAddresses()`, `getNumPoints()`, `getNumVertices()` and `getNumStrips()` work exactly like their coDoPolygon counterparts.

Optimal rendering performance already be reached with about 5 to 6 triangles per Strip.

Example:

## 3.10 Pixel Image Objects

Pixel images with different formats can be stored in the PixelImage type. Warning: the data saved in the pixel buffer of this type is transferred in binary between machines and not converted.

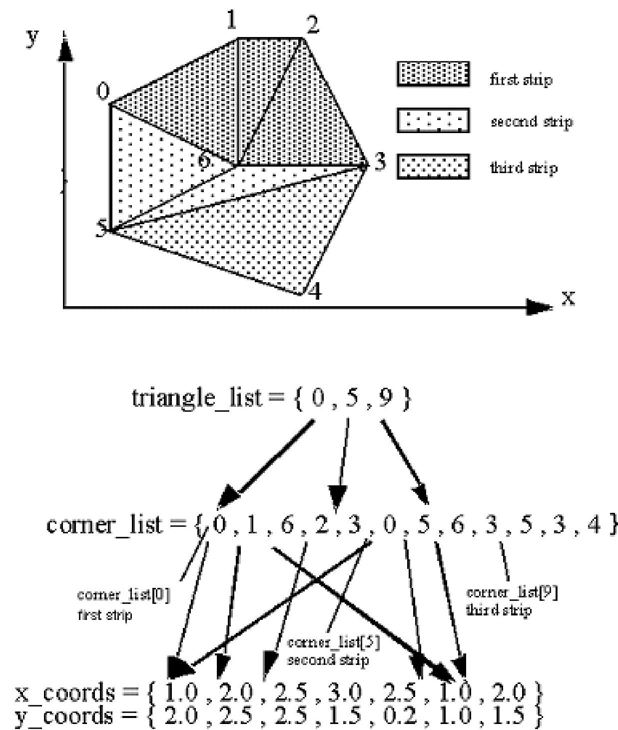



**Figure 3.13: Triangle Strip example**

| **coDoPixelImage(const char *name, int width, int height, unsigned form, short psize)** | | |
|---|---|---|
| Description: | create a Pixel Image | |
| IN: | width | Image width |
| IN: | heigth | Image heigth |
| IN: | form | format ID |
| IN: | psize | number of bytes per pixel |

| **char* getPixels()** | |
|---|---|
| Description: | get a pointer to the storage area |
| Return value: | pointer to buffer area |

All other parameters can be requested using member functions:

| | |
|---|---|
| `int getWidth()` | delivers image width |
| `int getHeight()` | delivers image height |
| `int getPixelsize()` | delivers image pixel size |
| `int getormat()` | delivers format ID |

### 3.10.1 2D Textures

Textures are, if supported in hardware, a fast method to apply complex coloring to geometrical objects.

A Texture object is created from a PixelImage, a list of vertex indices, a 2D field of texture coordinates and some additional information specific to texture usage in OpenGL.

| **coDoTexture(const char *name, coDoPixelImage* image, int b, int c, int l, int nv,int* vi, int nc, float** coords);** | | |
|---|---|---|
| Description: | create a texture object | |
| IN: | image | texture buffer |
| IN: | b | number of border pixels |

## 3.11 Text Objects

Text objects can be used to transmit binary data between modules. Furthermore the Inventor renderer understands a special kind of text object directly: OpenInventor file format (or VRML 1.0 format).

| **coDoText(const char *name, int numBytes)** | | |
|---|---|---|
| Description: | create a binary object | |
| IN: | name: | name of text object |
| IN: | numBytes | allocated size in bytes |

| **coDoText(const char *name, const char *value)** | | |
|---|---|---|
| Description: | create a binary objectfrom a given string and copy string to it | |
| IN: | name: | name of text object |
| IN: | value | NONZERO terminated string |

| **int getTextLength()** | |
|---|---|
| Description: | request length of storage area |
| Return value: | length of buffer in bytes |

| **void getAddress(char **text_ptr)** | | |
|---|---|---|
| Description: | get buffer pointer | |
| OUT: | text_ptr | buffer pointer |

Text objects should not be used for generating geometry for the renderer. Instead the COVISE geometry object classes are recommended. In this case other existing modules may be able to also handle data objects.

Using Text objects for anything else than ASCII text is generally discouraged, since neither type conversion nor field size adjustment is done on this data.

## 3.12 Integer Arrays

Integer arrays can be used to send indices or other integer fields. Fluid dynamics codes often save cell types, materials and other properties per cell. These properties can be useful to allow selection of parts, e.g. with the SelectUsg module. The class IntArr defines an integer array with an arbitrary number of dimensions. It defined them in a one dimension field, saving the sizes per dimension, and the data field as one large field.

| **coDoIntArr(const char *objName, int numDim, const int *dimArray, const int *initdata=NULL)** | | |
|---|---|---|
| Description: | | |
| IN: | numDim | number of dimensions |
| IN: | dimArray | integer array holding the sizes for numDim dimensions |
| IN: | initdata | if given and non-zero: initialization field |

| **int getNumDimensions()** | |
|---|---|
| Description: | get number of dimensions |
| Return value: | number of dimensions |

| **int getDimension(int i)** | |
|---|---|
| Description: | get size in i-dimension (i=[0..numDim-1] |
| Return value: | size in i-dimension |

| **int getSize()** | |
|---|---|
| Description: | get overall size |
| Return value: | product of all sizes |

| **int *getAddress()** | |
|---|---|
| Description: | get base address |
| Return value: | pointer to table in memory |

## 3.13 Container Class coDoSet

The class coDoSet is a container for a certain number of objects of the same type. These objects can also be sets, thus creating hierarchies of objects. Such hierarchies should be used cautiously since every access to a data object means a task-task communication with the CRB. Thus excessive usage of set hierarchies can lead to significant performance penalties.

There are two cases, where sets are typically used:

- The real-world data is structured in a hierarchical system: typically FEM data comes from CAD models, which have a strong hierarchical structure.
- The data is time-dependent: in this case there is one object per time step. Time dependent data is created by attaching an attribute to the set declaring this set to be a time series. A time set is simply any set with an attribute "TIMESTEP" and the value "0 <last number>" attached to the container object.

The creation of a coDoSet containing n objects consists of the following steps:

1. Creation of an array for the object pointers: *one element larger for termination* `coDistributedObject **objects = new coDistributedObject* [n+1]`
2. Creation of n objects of any kind and assigning it to `objects[i]`. The class `coDistributedObject` is the base class of all object classes, thus it can be assigned without casting. All object names *must* be different. As a convention, the name returned from the port is taken and '_i' is added using the current index i.
3. Assignment of `objects[n] = NULL` to terminate the list.
4. Creation of the set object with `objects` as the parameter.
5. Deletion of all part objects with `delete objects[i]`.
6. Deletion of the array using `delete [] objects`.

For the final level, the set object is assigned to the port, otherwise it might be integrated into another set and deleted after creation of the higher level set. As a rule, any object created with `'new *'` must either be deleted or assigned to a port.

There are several constructors for coDoSet, but only the following is supported:

| **coDoSet(const char *name, coDistributedObject **elements)** | | |
|---|---|---|
| Description: | Create a set | |
| IN: | name | name of the set container to fill |
| IN: | elements | null-terminated array of COVISE data objects |

To retrieve data from a set, there are two possibilities:

| **coDistributedObject * const *getAllElements(int *no)** | | |
|---|---|---|
| Description: | get all elements of a set | |
| OUT: | no | number of elements in the set |
| Return value: | Array of object pointers, must be deleted with delete[] | |

| **coDistributedObject *getElement(int no)** | | |
|---|---|---|
| Description: | get all elements of a set | |
| IN: | no | number of elements in the set |
| Return value: | pointer to set object | |

# 4 Simulation Library

## 4.1 Usage models

### Interactive simulation

The primary focus of the Simulation Library is to allow an easy implementation of interactive simulations. An interactive simulation is a simulation code which is started directly from COVISE, immediately visualizes its results and allows parameter changes during ongoing simulation. It is assumed, that the simulation code is not directly integrated as subroutine into a COVISE module but a client-server coupling to a remote machine is used.

### Control Visualization

Another application of simulation coupling is checking simulations with long execution times. In this case, the simulation start-up is not done by COVISE, but COVISE can connect to the running simulation and display the intermediate results. This allows an engineer to check them and terminate erroneous simulations early, avoiding unnecessary computing costs. Control visualization commands with repeated attachment to and detachment from the simulation is not yet implemented.

## 4.2 Basic structure

The simulation library is split into two parts:

- The simulation server is the COVISE-sided part of the simulation coupling. It is integrated into a module and takes care of the flow control, receives the parameters from the Map-Editor and puts the data from the simulation into the COVISE administered shared memory.
- The simulation client is the simulation-sided part of the coupling. It connects the COVISE module to the simulation code, exchanges messages with the module to request parameter values and sends results to the server for visualization.

## 4.3 Language bindings

The simulation client is written in ANSI C and calls only functions of the standard C library, which is automatically linked to every code on UNIX operating systems. This is important, because some simulation codes do not allow additional libraries to be added to the linker line but only object files.

The implementation language of COVISE is C++, which also applies to the COVISE-sided part of CoSimLib used by the simulation server. Thus it is possible to use all features of the development system.

The simulation library is accessible from multiple languages via appropriate language bindings. At the moment, 4 languages are supported:

1. C as the internally-used ‘natural’ language binding for the client

2. FORTRAN77 as the most common language used for simulations
3. Fortran 90 by using backward-compatibility with FORTRAN77
4. C++ by using header-definitions to map C++ language calls to C language calls

Additional language bindings will be supplied if necessary.

## 4.4 Simulation Server Commands

The Simulation Server is the COVISE-sided part of the simulation coupling. When using interactive simulations, the module has to start the simulation code, for control visualization the module has to connect and authenticate to a running simulation. In both cases a network connection has to be created for control and data flow between the simulation and COVISE.

### Simulation start-up

The simulation start-up is highly dependent on the circumstances in which the simulation takes place. To ensure sufficient flexibility for the module programmer as well as for the module user, the start-up procedure can be configured in the covise.config file. The following lines show an excerpt from covise.config.

```
MiniSim
{
   STARTUP SGI    CO_SIMLIB_CONN=%e ; export CO_SIMLIB_CONN; cd %0; ./miniSim
   STARTUP MANUAL echo "set CO_SIMLIB_CONN=%e and start MiniSim"
   PORTS    31000-31999
   SERVER   Module
   TIMEOUT  90
   VERBOSE  1
}
```

For each simulation coupling module, a section with the name of the module executable is searched. This section must include at least one STARTUP line, since no default start-up can be defined. The first word after STARTUP defines a label, the rest is executed by a shell in the active directory where COVISE was started in. If the module is started on a remote host, it is executed in the directory the user enters on login, usually the home directory. It has to define an (exported) environment variable CO_SIMLIB_CONN in the shell and then start the simulation job itself. The following variables can be used in the STARTUP string:

```
%%        the % character itself
%e        the value CO_SIMLIB_CONN has to be set to
%h        The host IP of the simulation host in "1.2.3.4" fashion (user-defined)
%0 ... %9 User-definable strings
```

Every SimLib module automatically creates a choice parameter for selecting the startup line from the possibilities given in the config file.

Examples:

```
STARTUP LOCAL  CO_SIMLIB_CONN=%e ; export CO_SIMLIB_CONN; cd %0; ./miniSim
```

If the user chooses "LOCAL" on the startup method choice, the environment variable is set, then changes to the module-defined directory (possibly from a string parameter of the module) and starts the program ‘miniSim’.

```
STARTUP REMOTE xterm -geometry 135x35 -e rsh -l %0 %h 'cd %1;
                        export CO_SIMLIB_CONN=%s; echo y | star' &
```

This has to be written in one line. Here the user creates a new terminal window, starts a remote shell on a given host with a user-supplied login, changes to a user-supplied directory, sets the environment variable, then changes to the module-defined directory (possibly from a string parameter of the module) and then starts the program 'star', piping a 'y' answer to a 'are you sure' question.

All other parameters in the file are optional:

```
PORTS    <minPort>-<maxPort>
```

Sets the range of allowed port numbers for the network connection. For a single port, minimum and maximum has to be set to the same value. If no port range is given, the range 31000 ... 31999 is used. The explicit setting of port numbers is only required if firewalls are between the simulation host and the COVISE module host. Consult your Network administrators for help with firewalls and port numbers.

```
SERVER   Module|Client
```

Defines, which side of the client/server pair should open the TCP connection. As a default, the server is opened by the module. Most firewalls require the server 'outside' of the secure area. Consult your Network administrators for help with TCP socket opening directions.

```
TIMEOUT  90
```

Time in seconds to wait for a connection before giving up. Default is 90 sec. Increase the value if the network is extremely slow or if the simulation start-up time before starting the network client is long.

```
VERBOSE  1
```

Verbose level: when set to values $>$ 0, CoSimLib commands create debug output to stderr. Currently, values 1...3 are supported.

```
LOCAL  <local machine name>
```

The machine name for the simulation to connect to. By default this is set to the result of a nameserver lookup performed by a gethostname system call. This is typically the default network interface. The local machine name can be changed either to use a faster network connection or to connect through firewalls via IP-masquerading. Consult your network administrator for help with firewall connections and IP addresses. The IP number of the local host can also be set by the program.

The start-up is triggered by the `startupSim()` command:

| **int coSimLib::startupSim()** | |
|---|---|
| Description: | 1. Analyses the parameters given in the covise.config<br>2. Sets verbose level<br>3. Sets % arguments in STARTUP line<br>4. Calls shell with STARTUP command<br>5. Tries to connect with simulation |
| COVISE states: | all |
| Return value: | -1 on error, 0 on success |

If the user arguments (%0 ... %9) are used in the covise.config START line, they have to be set before the

startupSim() command is called.

## Status request

All coSimLib commands check for errors, which are usually reported by returning (-1) as result. For further checking, the internal status variable can be accessed:

| **int coSimLib::status()** | |
|---|---|
| Description: | Return the error code of the last failed operation |
| COVISE states: | all |
| Return value: | = 0 no error,<br>$>$ 0: UNIX errno variable value,<br>$<$ 0: unknown Err |

## Setting start-up parameters

Both the local and target host IP addresses can be set explicitly. As a default, the local interface to be used is set to the value of the LOCAL variable if given, otherwise the host's default interface is used.

Both functions call the nameserver if the host is not given in dot notation and check whether the requested host name exists. If the name is given in dot notation, no nameserver lookup or host existence testing is performed.

| **int coSimLib::setTargetHost(const char *hostname)** | | |
|---|---|---|
| Description: | | |
| COVISE states: | all, only before coSimLib::startupSim() | |
| IN: | hostname | name of target host |
| Return value: | 0 = no error, -1 = host unknown | |

| **int coSimLib::setLocalHost(const char *hostname)** | | |
|---|---|---|
| Description: | Set the local hostname | |
| COVISE states: | all, only before coSimLib::startupSim() | |
| IN: | hostname | name of local host |
| Return value: | 0 = no error, -1 = host unknown | |

The user arguments %0 ... %9 must be set explicitly before starting the simulation. This allows simulation specific parameters, e.g. starting directory or startup parameters, to be set.

| **int coSimLib::setUsrArg (int num, const char *argVal)** | | |
|---|---|---|
| Description: | Set one of the user arguments | |
| COVISE states: | all, only before coSimLib::startupSim() | |
| IN: | num | which argument to set: 0...9 |
| IN: | argVal | value of the argument |
| Return value: | 0 = no error, -1 = unknown | |

### Starting the Server mode

Whenever this command is called, the module waits for the simulation to send client commands until a 'finish' command is sent. This may only be done from the compute callback.

| **int coSimLib::serverMode()** | |
|---|---|
| Description: | set server mode: wait for client commands and perform them until a finish command is received |
| COVISE states: | Compute |
| Return value: | 0 = returned by finish command,<br>-1 = returned because of error |

Parameter requests are also handled outside of the server mode, but data creation is only allowed when the simulation server module is in the compute() callback and in server mode.

### Requesting and setting verbose mode

The verbose level can be set and requested by the user. It is recommended to request the verbose mode when creating own debug output in simulation coupling modules.

| **void coSimLib::setVerbose(int level)** | | |
|---|---|---|
| Description: | set verbose level | |
| IN: | level | 0=no, 4=max. verbose level |

| **int coSimLib::getVerboseLevel()** | |
|---|---|
| Description: | get verbose level |
| Return value: | verbose level: 0=no, 4=max. verbose level |

## 4.5 Simulation Client Commands

### Start-up the connection with COVISE

This command checks the environment for the variable CO_SIMLIB_CONN, analyses it and tries to open a connection with the COVISE module. Depending on the setting in the covise.config file, this can be either a client or a server connection on any port in the specified direction. The command tries to open the connection for the period specified in the configuration file and returns with an error message if the connection was not established.

| **C:** | **coInitConnect()** |
|---|---|
| **FORTRAN:** | **COVINI()** |
| Description: | Start connection with COVISE |
| Return value: | 0 = successfully connected, -1 = not connected |

## Check whether COVISE is connected

This call checks whether the connection to COVISE is still active. It does not check, whether the module is able to handle data now or whether it is busy.

| **C:** | **coNotConnected()** |
|---|---|
| **FORTRAN:** | **CONOCO()** |
| Description: | Check connection with COVISE |
| Return value: | 0 = connected, -1 = not connected |

## Retrieving parameter values

All parameter calls can be submitted both in the compute and main-loop state of the COVISE coupling module. One should be aware that only immediate mode parameters are updated at the module immediately while non-immediate parameters are only updated before entering the compute() callback.

All parameter requests have the parameter name as input and the requested parameter(s) as output. Returned is an error code: 0 for proper function, -1, if the parameter is not yet available, -2 for an unknown parameter name.

Not all parameter types are currently implemented in the simulation library.

| **int coGetParaSlider(const char *name,float *min, float *max, float *value)** | | |
|---|---|---|
| **COGPSL(name,min,max,val)** | | |
| Description: | Gets values from a COVISE slider parameter: minimum, maximum and value of the slider parameter. | |
| IN: | name | name of parameter |
| OUT: | min, max, value | Slider parameters) |
| Return value: | 0 = ok, -1 = not yet available, -2 = unknown | |

| **int coGetParaFloat(const char *name, float *value)** | | |
|---|---|---|
| **COGPFL(name,value)** | | |
| Description: | Gets values from a COVISE float parameter | |
| IN: | name | name of parameter |
| OUT: | value | parameter value (output) |
| Return value: | 0 = ok, -1 = not yet available, -2 = unknown | |

| **int coGetParaInt(const char *name, int *value)** | | |
|---|---|---|
| **COGPIN(name, value)** | | |
| Description: | Gets values from a COVISE int parameter | |
| IN: | name | name of parameter |
| OUT: | value | parameter value (output) |
| Return value: | 0 = ok, -1 = not yet available, -2 = unknown | |

| **int coGetParaChoice(const char *name, int *value)** | | |
|---|---|---|
| **COGPCH(name, ivalue)** | | |
| Description: | Gets values from a COVISE choice parameter | |
| IN: | name | name of parameter |
| OUT: | value | parameter value (output) |
| Return value: | 0 = ok, -1 = not yet available, -2 = unknown | |

| **int coGetParaBool(const char *name, int *value)** | | |
|---|---|---|
| **COGPBO(name, ivalue)** | | |
| Description: | Gets values from a COVISE choice parameter | |
| IN: | name | name of parameter |
| OUT: | value | parameter value (output) |
| Return value: | 0 = ok, -1 = not yet available, -2 = unknown | |

| **int coGetParaFile(const char *name, char *filepath)** | | |
|---|---|---|
| **COGPFI(name, path)** | | |
| Description: | Gets values from a COVISE file-browser parameter | |
| IN: | name | name of parameter |
| OUT: | filepath | selected file (output) |
| Return value: | 0 = ok, -1 = not yet available, -2 = unknown | |

| **int coGetParaText(const char *name, char *text)** | | |
|---|---|---|
| **COGPTX(name, text)** | | |
| Description: | Gets values from a COVISE file-browser parameter | |
| IN: | name | name of parameter |
| OUT: | value | parameter value (output) |
| Return value: | 0 = ok, -1 = not yet available, -2 = unknown | |

## Creating data objects

Currently only the creation of unstructured scalar and vector data fields (`coDoFloat` and `coDoFloat`) is supported.

| **int coSend1Data(const char *name, int num, float *data)** | | |
|---|---|---|
| **COSU1D(name, num, data)** | | |
| Description: | Create a 1D unstructured data object at a port and copy the data to the shared memory | |
| State: | Compute | |
| IN: | name | name of output port |
| IN: | num | number of data elements |
| IN: | data | Pointer to array containing data |
| Return value: | =0: ok , otherwise error | |

| **int coSend3Data(const char *name, int num, float *x, float *y, float *z)** | | |
|---|---|---|
| **COSU3D(name, num, x,y,z)** | | |
| Description: | Create a 3D unstructured data object at a port and copy the data to the shared memory | |
| State: | Compute | |
| IN: | name | name of output port |
| IN: | num | number of data elements |
| IN: | x,y,z | Pointer to arrays containing data components |
| Return value: | =0: ok , otherwise error | |

Other types will be implemented in the future, especially for sending unstructured grids and structured data sets.

## 4.6 Handling data on parallel machines

Parallelization of simulations is usually done by domain decomposition: parts of the grid are decomposed to several processors. For these kinds of simulations, only one node is connected to COVISE and has to perform the data and parameter propagation.

### Initialisation

The simulation library supports distributed data by supplying a set of additional calls, which prepare and perform the gathering operations. Therefore, two different index mappings can be defined for both cell indices and vertex indices. To set these tables, all nodes send their local-to-global mapping field (array size of local data size, containing global data indices).

| **int coParallelInit(int numParts, int numPorts)** | | |
|---|---|---|
| **COPAIN(numprocs,numports)** | | |
| Description: | initialise parallel data sending | |
| State: | All | |
| IN: | numParts | number of domains |
| IN: | numPorts | number of ports creating parallel data |
| Return value: | =0: ok, otherwise error | |

| **int coParallelPort(const char *portname, int isCellData);** | | |
|---|---|---|
| **COPAPO(portname, isCellData)** | | |
| Description: | Declare the named port as parallel data port | |
| State: | All | |
| IN: | Portname | name of the output data port |
| IN: | IsCellData | whether this is<br>(1)cell- or<br>(0)vertex-based data |
| Return value: | =0: ok, otherwise error | |

There can be two different tables for mapping local fields into global fields: a cell map and a vertex map. For every node, the cell map gives for each local cell the cell numbers in the global field. Accordingly, the vertex table gives the global coordinate index for every local coordinate.

| **int coParallelCellMap(int node, int numCells, int *locToGlob);** | | |
|---|---|---|
| **int coParallelVertexMap(int node, int numVert, int *locToGlob);** | | |
| **COPACM(node, numCells, locToGlob)** | | |
| **COPAVM(node, numVert, locToGlob)** | | |
| Description: | Set index mapping for a node's cell/vertex data | |
| State: | All | |
| IN: | Node | node ID (starting with node 0) |
| IN: | NumCells | number of cells on this node |
| IN: | NumVert | number of vertices on this node |
| IN: | LocToGlob | global index for each local index |
| Return value: | =0: ok, otherwise error | |

## Data creation

Data is created by telling the COVISE server, which node sends, and then using the standard data object creation calls, which automatically gather the data according to the mapping set during the initialization.

| **int coParallelNode(int node);** | | |
|---|---|---|
| **COPANO(node)** | | |
| Description: | set current processor number | |
| State: | All | |
| IN: | node | node ID (starting with node 0) |
| Return value: | =0: ok, otherwise error | |

# 5 Other Features

## 5.1 Error, Warning and Info Messages

The base class coModule delivers a set of methods to send messages to the users. Info messages are displayed in the info box at the bottom of the Map Editor, warnings and errors in a pop-up window can be configured to pop up a window. The difference between warnings and errors is that after a module has issued an error, no other modules will execute automatically, while warnings only pop up without any impact on the flow control.

The programming interface of the message commands only differs in the endings:

| **Function names** | **Meaning** |
|---|---|
| coModule::sendInfo | Info message |
| coModule::sendWarning | Warning message |
| coModule::sendError | Error message |

| **static void coModule::sendWarning(const char *fmt, . . . );** | | |
|---|---|---|
| Description: | Send messages to the user interface | |
| IN: | fmt | Message format string |

All of the above message methods accept `printf` style arguments.

## 5.2 Explicit flow control commands

The control message passing is done internally in the module base class, usually none of the messages has to be sent by the user himself. Nevertheless, users might send own messages for explicit flow control to achieve effects that extend the classical data flow paradigm. Since these explicit calls violate the basic structure of the COVISE system, it is discouraged to use these calls. Explicit flow control calls can both influence stability and performance of the COVISE system and should be used by experienced users only and with special care for internal racing conditions.

### Explicitly stopping the pipeline

The function `void Covise::send_stop_pipeline()` explicitly demands that no other modules be started automatically by the controller. It is discouraged to use this function: `sendError` accomplishes the same effect and also submits an error message. In all other than error cases the pipeline should run based on object creation and module connections.

### Implicitly stopping pipeline execution

A module can prevent down-stream modules from being activated by not creating an output object. Any module is supposed to stop if a connected port receives no data. Using the classic module interface from

earlier versions, this was the case when an object name was received, but no object could be retrieved based on this name. All modules created based on coModule automatically implement this behavior.

### Self-execution

Any module callback (except compute) can tell the module that it should 'execute' after the callback has been left. To achieve this, the function calls `void coModule::selfExec()`. Multiple calls within a callback will only trigger a single start. *Warning: Executing a module again while the succeeding module is still running may cause COVISE to crash.*

To prevent 'overrun' crashes, the module waits one second after the selfExec call has been submitted. This period can be configured by placing an entry 'execGracePeriod' (measured in seconds) into the 'System.HostInfo' section of the configuration files or with the call `coModule::setExecGracePeriod(float gracePeriod)`.

## 5.3 Re-using objects

Usually, COVISE objects have an automatic life period: They are created by a module and are destroyed when they are not needed any more. Nevertheless, COVISE makes some assumptions about the object usage:

- Objects read once at an input port are not used any further
- An object created in a module is either put ONCE into an output port or put ONCE into a Set or Geometry container

If a user wants to violate these assumption, he must tell the object that it is used more than once by incrementing its reference counter. This can be useful to

- "hand through" an object from an input to an output port by adding it to a container object
- add an input object to a container and give this to the output
- add an object more than once to a container - e.g. for a constant part within a time series.

To increment the reference counter, call:

| **coDistributedObject::incRefCount()** | |
|---|---|
| Description: | add a reference to a distributed object |
| Return value | number of references |

## 5.4 Overriding parameter values

You may use the class coHideParam in order to override the values of some module parameters. For instance, you may wish to adjust the behaviour of the module not by directly modifying the module params, but rather let the module read their values from a string that the module may get for instance through a Text object at an input port, or that may be alternatively accessible as the value of some attribute given to one of the input objects. The example below should make it clear how this coHideParam class is used:

myModule.h

```
class myModule
{
   private:

      coBooleanParam *p_OneBooleanParam_;
      coFloatParam *p_OneFloatScalarParam_;
      coFloatVectorParam *p_OneIntVectorParam_;

      coHideParam *h_distance_of_plane_;
      coHideParam *h_OneFloatScalarParam_;
      coHideParam *h_OneIntVectorParam_;
      std::vector<coHideParam *> hparams_; // a useful container
                                          // for the previous
                                          // coHideParam pointers

}
```

myModule.cpp

```
void
myModule::postInst()
{
   // create the hiding objects here
   // and keep pointers in the container
   hparams_.push_back(h_OneBooleanParam_ =
                      new coHideParam(p_OneBooleanParam_));
   hparams_.push_back(h_OneFloatScalarParam_ =
                      new coHideParam(p_OneFloatScalarParam_));
   hparams_.push_back(h_OneIntVectorParam_ =
                      new coHideParam(p_OneIntVectorParam_));
}


myModule::myModule()   // ... build ports and parameters as usual
{
  // create params here
  p_OneBooleanParam_ = addBooleanParameter("OneBooleanParamName",
                                           "a bool param");
  p_OneFloatScalarParam_ =
         addFloatParameter("OneFloatScalarParamName",
                                         "a float scalar param");
  p_OneIntVectorParam_ =
         addFloatParameter("OneIntVectorParamName",
                                         "an int vector param");

  // and set default values
  ....
}

#include <stringstream>

myModule::compute(const char *)
{
  // RESETTING HIDING OBJECTS
```

```
   // first of all reset all hparams_
   for(int param = 0;param<hparams_.size();++param){
      hparams_[param]->reset();
   }
   ...

   char *values;
   ...

   // OVERRIDE PARAMETER VALUES (LOADING coHideParam OBJECTS)
   // assume that after the previous lines
   // 'values' points to the string:
   // "OneBooleanParamName 0\nOneFloatScalarParamName 3.47\n
   //  OneIntVectorParamName 2 5 3\n"
   char *oneValue = new char[strlen(values)+1];
   istringstream pvalues(values);
   while(pvalues.getline(oneValue,strlen(values)+1)){
      for(int param=0;param<hparams_.size();++param){
         hparams_[param]->load(value);
      }
   }
   delete [] value;
   ...
   ...

   // AND NOW WE USE THE coHideParam OBJECTS
   // INSTEAD OF THE PARAMETERS
   // whenever you would normally use a port, ...
   //     float notThisWay =
   //                  p_OneFloatScalarParam_->getValue();
   // ...you use now instead the pertinent
   // coHideParam object
   float ratherThisWay = h_OneFloatScalarParam_->getFValue(); // 3.47
   int intArray[3];
   intArray[0] = h_OneIntVectorParam_->getIValue(0); // 2
   intArray[1] = h_OneIntVectorParam_->getIValue(1); // 5
   intArray[2] = h_OneIntVectorParam_->getIValue(2); // 3
   if(h_OneBooleanParam_->getIValue()){ // the returned value is 0
      ...
   }
}
```

Some remarks are here in order. Using the reset() member function for all coHideParam objects should always be done before using load(). After resetting, the object forgets previously assigned values, and if a value is requested using one of the getIValue or getFValue functions, the requested value will be that of the hidden parameter, i. e. the parameter that was indicated in the constructor of the coHideParam object (see myModule::postInst()).

Hiding parameter values is achieved when using the load() member function as used in the example. When the string *values* contains no substring for a given parameter, the corresponding coHideParam object will not hide that parameter. Otherwise, it will take the values that follow the parameter name.

If you are deriving myModule from coSimpleModule instead of coModule, you may prefer performing the resetting and loading of coHideParam objects in preHandleObjects and not in the compute function.

The last part concerns requesting values from these objects. For this purpose you have the following member

functions at your disposal:

```
class coHideParam{
public:
   ...

   // get the float value
   // when hiding coFloatParam or coFloatSliderParam
   float getFValue();

   // get the int value
   // when hiding coChoiceParam, coBooleanParam,
   // coInt32Param or coIntSliderParam
   int getIValue();

   // get the float value
   // when hiding coFloatVectorParam
   float getFValue(int component);

   // get int values
   // when hiding coFloatVectorParam
   int  getIValue(int);

   // get float values when hiding
   // coFloatVectorParam (assuming 3 components)
   void getFValue(float &data0, float &data1, float &data2);
}
```

# 6 Example Modules

## 6.1 Hello

The "Hello" module shows just the minimal framework needed to build a module. It only defines the basic routines without implementing any functionality.

Hello.h:

**Make sure we never include the header twice**

```
#ifndef HELLO_H
#define HELLO_H

// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// ++                                                  (C)2000 RUS  ++
// ++ Description: "Hello, world!" in COVISE API                    ++
// ++ Author:                                                       ++
// ++                         Andreas Werner                        ++
// ++            Computing Center University of Stuttgart           ++
// ++                         Allmandring 30                        ++
// ++                        70550 Stuttgart                        ++
// ++                                                               ++
// ++ Date:  10.01.2000  V2.0                                       ++
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

**Basic module header file for API and library calls**

```
#include <api/coModule.h>

class Hello : public coModule
{
   private:
```

**The compute callback reacts on the execute messages**

```
     /// this module has only the compute callback
      virtual void compute(const char *);

   public:
```

**Construct the module**

```
      /// this is the Constructor
      Hello(int argc, char *argv[]);

};
```

```
#endif
```

Hello.cpp:

```
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// ++ Description: "Hello, world!" in COVISE API        (C)2000 RUS ++
// ++ Author:                                                    ++
// ++                        Andreas Werner                      ++
// ++            Computing Center University of Stuttgart        ++
// ++                        Allmandring 30                      ++
// ++                       70550 Stuttgart                      ++
// ++ Date:  10.01.2000  V2.0                                    ++
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

///// this includes our own class's headers
#include "Hello.h"

///// Constructor : This will set up module port structure
Hello::Hello(int argc, char *argv[])
```

**Set a module description**

```
      :coModule(argc, argv, "Hello, world! program")
```

**This is a really primitive module**

```
{
      // no parameters, no ports...
}

///// compute() is called once for every EXECUTE message
void Hello::compute(const char *)
```

**Just send an INFO message**

```
{
      sendInfo("Hello, COVISE!");
}

// ++++  What's left to do for the Main program:
// ++++  create the module and start it

\hline
   {\bf Create the module} \\
   {\bf And start it ...} \\
\hline

MODULE_MAIN(Examples, Hello)
```

## 6.2 Filter

The second example implements a scaling of an unstructured grid: The structure information (element list, type list and connectivity) are simply copied while the corner coordinates are scaled by a constant factor.

By convention, we name the implementation class and the main source and header file like the module name, in this case "Enlarge". We also always name the ports and parameters "p_..."

Enlarge.h:

```
#ifndef ENLARGE_H
#define ENLARGE_H

// +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// ++ Description: Filter program in COVISE API        (C)2000 RUS  ++
// ++ Author:                                                       ++
// ++                        Andreas Werner                         ++
// ++            Computer Center University of Stuttgart            ++
// ++                        Allmandring 30                         ++
// ++                       70550 Stuttgart                         ++
// ++ Date:  10.01.2000  V2.0                                       ++
// +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

#include <api/coModule.h>

class Enlarge : public coModule
{
   private:


      /// this module has only the compute call-back
      virtual void compute(const char *);
```

**This will be the slider for the enlargement factor**

```
      ////////// parameters
      coFloatSliderParam *p_scale;
```

**An input and an output port. By convention, all ports start with "p_"**

```
      ////////// ports
      coInputPort  *p_inPort;
      coOutputPort *p_outPort;

   public:

      Enlarge(int argc, char *argv[]);
};

#endif
```

Enlarge.cpp:

```
// +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
// ++ Description: Filter program in COVISE API        (C)2000 RUS  ++
// ++ Author:                                                       ++
// ++                         Andreas Werner                        ++
// ++             Computer Center University of Stuttgart           ++
// ++                         Allmandring 30                        ++
// ++                        70550 Stuttgart                        ++
// ++ Date:  10.01.2000  V2.0                                       ++
// +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

#include "Enlarge.h"

// Construct the module

Enlarge::Enlarge(int argc, char *argv[])
```

**The module's description**

```
      :coModule(argc, argv, "Enlarge, world! program")

{
   // Parameters

   // create the parameter
```

**The parameter is not created with its constructor, but by calling a member of coModule to register it at the module**

**The parameter's name must be unique, the description doesn't**

```
   p_scale = addFloatSliderParam("scale","Scale factor for Grid");
```

**Default values for the enlargement factor: between 0.1 and 5.0, current value 2.0**

```
   // set the default values

   p_scale->setValue(0.1,5.0,2.0);
```

**Input and output ports: Unique name, Type list (here only one type), and description**

```
// Ports
   p_inPort  =  addInPort("inPort", "UnstructuredGrid",
                                         "Grid input");
   p_outPort = addOutputPort("outPort","UnstructuredGrid",
                                        "Grid output");

}


// compute callback: called when new input data arrived
```

```
void Enlarge::compute(const char *)
{
   int i;
```

**Retrieve Object from Port**

```
   coDistributedObject *obj = p_inPort->getCurrentObject();
```

**Check whether we got an object**

```
     // we should have an object
   if (!obj)
   {
      sendError("Did not receive object at port %s",
                 p_inPort->getName());
      return;
   }
```

**Check the type: This is not guaranteed by the port types!**

```
  // it should be the correct type
   coDoUnstructuredGrid *inGrid = dynamic_cast<coDoUnstructuredGrid *>(obj);
   if ( !inGrid )

   {
      sendError("Received illegal type at port '%s'",
                 p_inPort->getName());
      return;
   }
```

**We can safely cast up: we checked the type**

```
   // So, this is an unstructured grid

   // retrieve the size and the list pointers
   int   numElem,numConn,numCoord,hasTypes;
   int   *inElemList,*inConnList,*inTypeList;
   float *inXCoord,*inYCoord,*inZCoord;
```

**This call retrieves the field sizes from the object**

```
   inGrid->getGridSize(&numElem,&numConn,&numCoord);
```

**Do we have a type list?**

```
   hasTypes = inGrid->hasTypeList();
```

**This gives us the pointers to the data in shared memory**

```
inGrid->getAddresses(&inElemList,&inConnList,
                     &inXCoord,&inYCoord,&inZCoord);
```

**If we have a type list, we must copy it, too.**

```
if (hasTypes)
   inGrid->getTypeList(&inTypeList);
```

**Now allocate a new, empty USG object**

```
// allocate new Unstructured grid of same size
coDoUnstructuredGrid *outGrid
            = new coDoUnstructuredGrid(p_outPort->getObjName(),
                                       hasTypes);
```

**Oops, something went wrong here**

```
if (!outGrid->obj_ok())
{
   sendError("Failed to create object '%s' for port '%s'",
             p_outPort->getObjName(),p_outPort->getName());

   return;
}

int   *outElemList,*outConnList,*outTypeList;
float *outXCoord,*outYCoord,*outZCoord;
```

**This gives us access to the data space in shared memory. Beware of overwriting any fields**

```
outGrid->getAddresses(&outElemList,&outConnList,
                      &outXCoord,&outYCoord,&outZCoord);
```

**We have a type list. So here it is:**

```
if (hasTypes)
   outGrid->getTypeList(&outTypeList);
```

**We do not change the grid, so we copy all fields except the coordinates**

```
/// copy element list
for (i=0;i<numElem;i++)
   outElemList[i] = inElemList[i];

/// if we have one, copy the types list
if (hasTypes)
```

```
      for (i=0;i<numElem;i++)
         outTypeList[i] = inTypeList[i];

   /// copy connectivity list
   for (i=0;i<numConn;i++)
      outConnList[i] = inConnList[i];
```

**Get the current value from the slider**

```
   /// retrieve parameter
   float scale = p_scale->getValue();
```

**Negative or zero values make no sense: correct value and correct parameter on the user interface**

```
   if (scale<=0)
```

**Now put scaled coordinates into new object's memory**

```
   {
      scale = 1.0;
      p_scale->setValue(1.0)
   }

   /// copy coordinates and scale them
   for (i=0;i<numCoord;i++)
   {

      outXCoord[i] = inXCoord[i] * scale;
      outYCoord[i] = inYCoord[i] * scale;
      outZCoord[i] = inZCoord[i] * scale;
   }
```

**This is the output object for the port**

```
   // finally, assign object to port
   p_outPort->setCurrentObject(outGrid);

}
```

**Important: Never delete the objects you assigned to a port!**

```
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// ++++  What's left to do for the Main program:
// ++++                        create the module and start it
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

MODULE_MAIN(Examples, Enlarge)
```

Try to enhance the module: Different scales for each axis...

# 7 OpenCOVER Plugin Programming

The virtual reality Renderer module OpenCOVER is a COVISE module with support for virtual reality (VR) input devices such as Polhemus FASTRAK, Ascension Motionstar or A.R.T. tracking devices and backprojection displays with stereo projection like CAVE or workbench. OpenCOVER can also be started independently of COVISE and be used as a virtual reality viewer for 3D geometry.

The functionality of OpenCOVER can be extended through *plugins*. A plugin is a dynamic library with a certain interface. During execution, OpenCOVER executes the functions provided by each such library.

Plugins can either be used to implement interaction with other COVISE modules (also called "feedback"), for example for steering a simulation module, or for extending the functionality of OpenCOVER as a VR viewer.

The plugin programmer needs to derive from the class coVRPlugin and reimplement some of its virtual methods as well as its constructor and destructor. These virtual methods are called at the appropriate times while OpenCOVER is running. This class has to be advertised to OpenCOVER with the macro COVERPLUGIN.

A simple skeleton for an OpenCOVER plugin is available in covise/src/template/plugin.

## 7.1 Background on Dynamic Libraries

In UNIX systems dlopen makes a dynamic library available to a running process. dlsym returns the address of a symbol in the dynamic library (returns a pointer to a function in the dynamic library), dlerror returns diagnostic information and dlclose closes the dynamic library. The class coVRDynLib encapsulates this in a platform independent way.

If OpenCOVER notices that it has to open a plugin it creates an instance of the class coVRPlugin. The class coVRPlugin handles a plugin in OpenCOVER. It has virtual methods which correspond to the functions in a plugin, e. g. preFrame, postFrame, addObject, addInteractor, removeObject, . . . Then OpenCOVER opens the dynamic library with

```
handle=dlopen(name, mode)
```

and extracts the function with the name coVRPluginInit with

```
dlsym(handle, "coVRPluginInit")
```

which creates a new instance of coVRPlugin. All this is handled by the COVERPLUGIN macro.

During execution of OpenCOVER for each loaded plugin the methods are called at the appropriate time, e. g. coVRPlugin::preFrame is executed before OpenCOVER calls osgViewer::Viewer::frame and for each plugin coVRPlugin::postFrame is executed after.

## 7.2 OpenSceneGraph

OpenCOVER uses the scene graph library OpenSceneGraph. The plugin programmer should be familiar with the basic concepts of OpenSceneGraph such as the scene graph with its different types of nodes, the rendering loop and its main concepts. You can consult the OpenSceneGraph Quickstart Guide by Paul Martz (available as hardcopya or a free download from `http://www.lulu.com/content/767629`) and

the OpenSceneGraph Doxygen documentation (available online at http://www.openscenegraph.org/projects/osg/wiki/Support/ReferenceGuides) for more information.

OpenCOVER can be configured to use multiple threads for rendering.

The scene graph of OpenCOVER which is relevant to the plugin programmer looks like this:

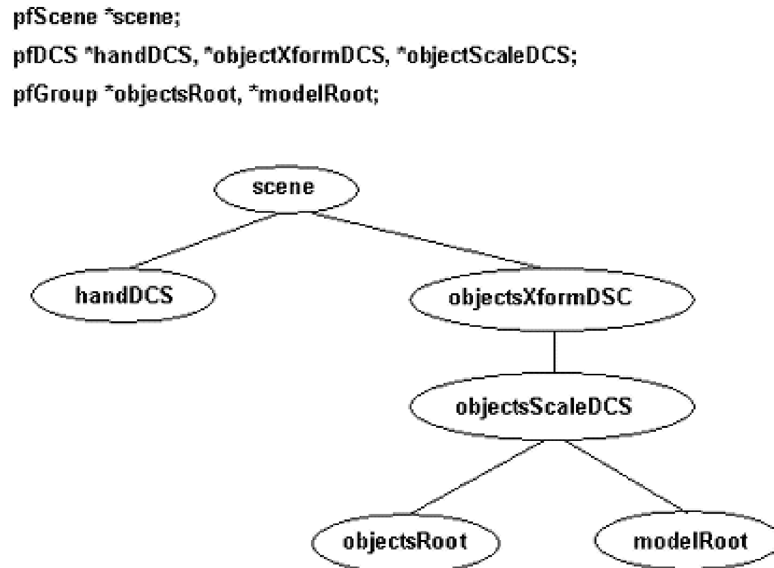



**Figure 7.1: OpenCOVER scene graph**

The matrix of handTransform comes from the 3D input device of the hand e. g. the Polhemus Stylus or the Ascension 6DOF Mouse. Under handTransform the interactor geometry for the hand is located. This is for e. g. laser sword or the magnifying glass. For the action "xform" the matrix of the handTransform is applied also to the objectXformTransform, for the action "scale" only the x component of the position is used to compute a scale matrix and is applied to the objectScaleTransform. COVISE geometry and other 3D geometry (in IV, OBJ etc. format) is appended under the group node cover->getObjectsRoot().

## 7.3 OpenCOVER as a COVISE module

As a COVISE module, OpenCOVER loads the COVISE plugin and uses the library *coAppl* for interfacing to COVISE. This library is similar to the newer *coApi* (which is used by non-renderer modules) and provides functions to open a socket connection to the controller and data manager and to create COVISE data objects. OpenCOVER can also be used stand-alone. Depending on the number of command line parameters the program decides if a connection to COVISE is needed.

As OpenSceneGraph has its own main loop, OpenCOVER checks for COVISE "ADD OBJECT" or "REMOVE OBJECT" messages from COVISE every frame. If it receives an object, OpenCOVER creates the appropriate OpenSceneGraph objects and appends them to the scene graph under the node cover->getObjectsRoot(). If feedback information was appended to the object (see 7.8), OpenCOVER loads the appropriate plugin automatically.

Section 7.5.2 describes the methods your plugin can implement for interfacing with COVISE.

## 7.4 OpenCOVER as a VR viewer for 3D geometry

If OpenCOVER is started outside of COVISE, it interprets the argument as a file which should be loaded. If you want to load a plugin, this has to be specified in your COVISE configuration in the section COVER.Plugin. A minimal configuration file to enable OpenCOVER's Vrml97 plugin would look like this:

```
<?xml version="1.0"?>

<COCONFIG version="1" >
  <GLOBAL>
    <COVER>
      <Plugin>
        <Vrml97 value="on" />
      </Plugin>
    </COVER>
  </GLOBAL>
</COCONFIG>
```

## 7.5 The Plugin Interface

During execution OpenCOVER calls the functions provided by the plugin at certain points in the rendering loop. The functions are declared in cover/coVRPlugin.h.

None of these methods have to implemented, all are optional. However, you should make sure that all resources acquired by your plugin are released in its desctructor.

### 7.5.1 Plugin Life Cycle

The following methods are called during the different phases of the plugin life time.

| **coVRPlugin()** | |
|---|---|
| Description: | The plugin is constructed when it is loaded. There are four methods to load a plugin:<br>1. The plugin is specified in the configuration file. Then it is initialized in OpenCOVER before the main loop starts. In this case, the OpenCOVER scene graph is not yet fully initialized.<br>2. OpenCOVER gets a COVISE object with had a coFeedback object applied.<br>3. The user requests that a plugin be loaded from Tablet User Interface.<br>4. A plugin starts another plugin.<br>The plugin constructor should be used to do early plugin initialization that does not depend on OpenCOVER data structures. |

| **bool init()** | |
|---|---|
| Description: | This method is called when the plugin is loaded and when OpenCOVER has completed start-up, all its data structures are established. This is the place where you can insert objects into the scene graph, add new menu entries, . . . |

| **bool destroy()** | |
|---|---|
| Description: | This method is called to announce that OpenCOVER wants to remove your plugin. Reimplement it in order to do early clean-up work. Return false to prevent your module from being removed. |

| **˜coVRPlugin()** | |
|---|---|
| Description: | This function is called before the plugin is removed from OpenCOVER.<br>It is important that you remove all your menu entries, remove your objects from the scene graph and release all resources acquired by your plugin. |

### 7.5.2 COVISE Interface

The following functions deal with COVISE data objects. A plugin can be informed if OpenCOVER gets ADD_OBJECT or DELETE_OBJECT messages:

Within OpenCOVER all COVISE data objects, i. e. objects derived from coDistributedObject, are represented as RenderObjects. These are replicated between all nodes if OpenCOVER runs on a render cluster for e. g. a CAVE. All data and attributes are copied to the corresponding fields of the RenderObject.

| **void addObject(RenderObject *baseObj, RenderObject *geomObj, RenderObject *normObj, RenderObject *colorObj, RenderObject *texObj, const char *parentName, int numCol, int colorBinding, int colorPacking, float *r, float *g, float *b, int *packedCol, int numNormals, int normalBinding, float *xn, float *yn, float *zn, float transparency)** | | |
|---|---|---|
| Description: | If implemented, this function is called whenever OpenCOVER receives a COVISE object. | |
| IN: | baseObj | The container object, i. e. the representation of the coDoSet object the newly added object is part of. |
| IN: | geomObj | The geometry container object consisting of a geometry object, a color object, a normal object and a texture object.<br>geomContainer is NULL if there is no container object but only a geometry object (e. g. polygons without colors) |
| IN: | normObj | The representation of the associated normals data. |
| IN: | colorObj | The representation of the associated colors data. |
| IN: | texObj | The representation of the associated texture data. |
| IN: | parentName | Name of the parent COVISE object, it is only set for set elements, for other data objects it is NULL. |
| IN: | numCol | Number of colors |
| IN: | colorBinding | Color binding type: PER_PRIMITIVE, PER_FACE, PER_VERTEX, NONE |
| IN: | colorPacking | Non-zero if colors are packed colors instead of RGB values. |
| IN: | r | Red color values. |
| IN: | g | Green color values. |
| IN: | b | Blue color values. |
| IN: | packedCol: | Packed RGBA colors. |

| | | |
|---|---|---|
| IN: | numNormals | Number of normals. |
| IN: | normalBinding | Normal binding type: PER_PRIMITIVE, PER_FACE, PER_VERTEX, NONE |
| IN: | xn | List of normals x values. |
| IN: | yn | List of normal y values. |
| IN: | zn | List of normal z values. |
| IN: | transparency | Transparency value. |

| **void removeObject(const char *objName, bool replace)** | | |
|---|---|---|
| Description: | If implemented, this function is called whenever OpenCOVER receives a DELETE_OBJECT message. | |
| IN: | objName | The name of the COVISE object to delete. |
| IN: | replace | true, if the COVISE object is replaced (this happens if the user a module executes again), false if it really has to be deleted (this happens, if the user removes a module or the connection between the module and OpenCOVER). |

| **void newInteractor(RenderObject *container, coInteractor *feedback)** | | |
|---|---|---|
| Description: | If implemented, this function is called whenever OpenCOVER receives a COVISE object with feedback information appended. | |
| IN: | container | The geometry container object consisting of a geometry object, a color object, a normal object and a texture object.<br>container is NULL if there is no container object but only a geometry object (e. g. polygons without colors). |
| IN: | feedback | The class coInteractor handles feedback in OpenCOVER. |

| **void coviseError(const char *errorMsg)** | | |
|---|---|---|
| Description: | If implemented, this function is called whenever OpenCOVER receives an error message. Error messages are created by modules with sendError. They are displayed in the Map Editor message window. | |
| IN: | errorMsg | The COVISE error message. |

### 7.5.3 Scene Graph Management

These functions are called when OpenCOVER appends or removes a node in the scene graph:

| **void addNode(osg::Node *node, RenderObject *obj)** | | |
|---|---|---|
| Description: | If implemented, this function is called when OpenCOVER adds a new COVISE object to the scenegraph or if other plugins insert a node into the scene graph. | |
| IN: | node | A pointer to the scene graph node. |
| IN: | obj | A pointer to the RenderObject generated from the COVISE object,<br>NULL if not corresponding to a COVISE object |

| **void removeNode(osg::Node *node)** | | |
|---|---|---|
| Description: | If implemented, this function is called if a node is removed from the scene graph. | |
| IN: | node | A pointer to the scene graph node. |

### 7.5.4 Render Loop

These functions are called at certain points during the execution of OpenCOVER.

| **void preFrame()** | |
|---|---|
| Description: | If implemented, this function is called from the main thread immediately before osgViewer::Viewer::frame() is called. |

| **void postFrame()** | |
|---|---|
| Description: | If implemented, this function is called from the main thread immediately after osgViewer::Viewer::frame() was called. |

| **void preSwapBuffers()** | |
|---|---|
| Description: | This method is called by the draw thread immediately before GL swap buffers. |

### 7.5.5 Other Events

All animations in OpenCOVER are synchronised. If a plugin needs to synchronise its interaction to the global animation, it should implement setTimestep to become informed whenever OpenCOVER displays another timestep.

| **void setTimestep(int timestep)** | | |
|---|---|---|
| Description: | If implemented, this function is called when OpenCOVER switches to a new timestep. | |
| IN: | timestep | The number of the current timestep. |

Plugins can be notified when the user presses keys on its keyboard.

| void key(int type, int keySym, int modifiers) | | |
|---|---|---|
| Description: | If implemented, this function is called when a key is pressed or released. | |
| IN: | type | Type of event: osgGA::GUIEventAdapter::KEYDOWN or KEYUP. |
| IN: | keySym | The key symbol, see osgGA/GUIEventAdapter. |
| IN: | modifiers | The state of the Shift and other modifier keys, also see osgGA/GUIEventAdapter. |

Plugins can send messages to each other, see 7.6.6.

| void message(int type, int len, const void *buf) | | |
|---|---|---|
| Description: | If implemented, this function is called when messages from other plugins arrive. | |
| IN: | type | Integer representing the message type. |
| IN: | len | Message length. |
| IN: | buf | A pointer to the message. |

## 7.6 Accessing OpenCOVER from a plugin

The class coVRPluginSupport gives the plugin programmer access to the OpenCOVER scene graph if he wants to add own geometry, or needs access to the menu to add new menu items, . . . This class is a singleton, the static instance of coVRPluginSupport named cover is declared in cover/coVRPluginSupport.h.

Example:

```
osg::Group *s = cover.getScene();
```

### 7.6.1 Access to the scene graph and its transformations

With the following methods you can access parts of the OpenCOVER scene graph (see Figure 12: OpenCOVER Scene Graph) or get/set transformations in the osg::MatrixTransform nodes.

| osg::Group* getScene(); | |
|---|---|
| Description: | Returns a pointer to the scene node |
| Return value: | pointer to the scene node |

| osg::ClipNode* getObjectsRoot(); | |
|---|---|
| Description: | Returns a pointer to the group node objectsRoot |
| Return value: | pointer to the group node |

| osg::MatrixTransform * getPointer(); | |
|---|---|
| Description: | Returns a pointer to the transform node for the hand which contains the transformation matrix of the 3D input device. |
| Return value: | pointer to the transform node for the hand/pointer device |

| **osg::MatrixTransform * getObjectsXform();** | |
|---|---|
| Description: | Returns a pointer to the objectsXform node. |
| Return value: | pointer to the transform node objectsXform |

| **osg::MatrixTransform * getObjectsScale();** | |
|---|---|
| Description: | Returns a pointer to the objectsXform node. |
| Return value: | pointer to the transform node objectsXform |

| **osg::Matrix &getBaseMat();** | |
|---|---|
| Description: | transformation matrix from object coordinates to world coordinates (objectScaleMat*objectsXformMat) |
| Return value: | transformation matrix |

| **osg::Matrix &getInvBaseMat();** | |
|---|---|
| Description: | transformation from world coordinates to object coordinates<br>1 / (objectScaleMat*objectsXformMat)<br>use this cached value instead of computing it on your own |
| Return value: | transformation matrix |

| **osg::Matrix &getPointerMat();** | |
|---|---|
| Description: | get matrix of handTransform<br>(same as getPointer()->getMatrix()) |
| Return value: | transformation matrix |

| **osg::Matrix &getXformMat();** | |
|---|---|
| Description: | get the matrix of objectsXform<br>(same as getObjectXforms()->getMatrix()) |
| Return value: | transformation matrix |

| **void setXformMat(osg::Matrix &transformMatrix);** | | |
|---|---|---|
| Description: | set the matrix of objectsXform<br>(same as getObjectXform()->setMatrix()) | |
| IN: | transformMatrix | transformation matrix |

| **float getScale();** | |
|---|---|
| Description: | get the scale factor of the scale node, scale factor is the same for all directions |
| Return value: | scale factor |

| **void setScale(float s);** | | |
|---|---|---|
| Description: | set the scale factor of the matrix of the scale node | |
| IN: | s | scale factor |

| **float getSceneSize();** | |
|---|---|
| Description: | get the scene size defined in covise.config |
| Return value: | the scene size in [mm] |

### 7.6.2 Access to the camera

You get the viewing matrix with:

| **osg::Matrix &getViewerMat();** | |
|---|---|
| Description: | get the transformation matrix of the viewer. The matrix contains the position and orientation of the users head. If headtracking is on, the viewer matrix changes every frame. |
| Return value: | transformation matrix |

### 7.6.3 Loading 3D Geometry

The following methods simplify the loading of files into the plugin:

| **const char *getname(const char *file);** | | |
|---|---|---|
| Description: | get the full name for the file (with absolute path) | |
| IN: | file | short file name |
| Return value: | full name | |

| **osg::Node *loadFile(const char *file);** | | |
|---|---|---|
| Description: | load a file in any format supported by OpenSceneGraph or a VRML97 file | |
| IN: | file | file name |
| Return value: | the root node of the scene graph created for this file | |

| **void loadVRML(const char *url);** | | |
|---|---|---|
| Description: | Load the specified VRML file into the OpenSceneGraph scene graph and appends it under objectsRoot | |
| IN: | url | url containing a vrml97 file |

| **void reloadVRML();** | |
|---|---|
| Description: | reload the previously loaded VRML file. If your VRML file is on a remote machine and you changed the model there you can update it in OpenCOVER by using this method |

| **void removeVRML();** | |
|---|---|
| Description: | removes the current VRML file from the scene |

| **osg::Node *loadIcon(const char *filename);** | |
|---|---|
| Description: | Load an icon file, similar to loadFile, but this method looks first in covise/icons/$LookAndFeel, then covise/icons |
| Return value: | the root node of the scene graph created for this file , or NULL, if the file was not found |

### 7.6.4 Access to the Buttons of the Input Device

| **coPointerButton *getButton();** | |
|---|---|
| Description: | returns a pointer to a pointerButton object |
| Return value: | class coPointerButton handles the button state |

### 7.6.5 Interactions

A plugin needs to know if the input device is already used for another interaction or if it is free for own interaction. If a plugin uses the input device it has to announce this to other plugins and to OpenCOVER.

### 7.6.6 Load and Unload Other Plugins and Communicate with them

| **coVRPlugin *addPlugin(const char *name);** | | |
|---|---|---|
| Description: | Load a plugin | |
| IN: | name | The name of the plugin. If the name of the shared object is libXxxPlugin.so, then the name is XxxPlugin |
| Return value: | Object of type coVRPlugin. Class coVRPlugin represents a plugin. | |

| **int removePlugin(const char *name);** | | |
|---|---|---|
| Description: | Unload a plugin | |
| IN: | name | The name of the plugin to be unloaded. |
| Return value: | true, if successfully unloaded | |

| **void unload(coVRPlugin *plugin);** | | |
|---|---|---|
| Description: | Unload a plugin | |
| IN: | plugin | The plugin object |

| **void sendMessage(coVRPlugin *sender, int toWhom, int type, int len, const void *buf);** | | |
|---|---|---|
| Description: | send a message to other plugins | |
| IN: | sender | the name of the plugin which sends the message |
| IN: | toWhom | destination types are:<br>TO_ALL,<br>TO_ALL_OTHERS,<br>TO_SAME,<br>TO_SAME_OTHERS,<br>NUM_TYPES |
| IN: | type | a user defined number. The sender can identify the message by means of that number. For messages sent to all plugins this should be unique. |
| IN: | len | length of the message |
| IN: | buf | the message |

| **void sendMessage(coVRPlugin *sender, const char *destination, int type, int len, const void *buf);** | | |
|---|---|---|
| Description: | send a message to a plugin | |
| IN: | sender | the name of the plugin which sends the message |
| IN: | destination | The name of the destination plugin |
| IN: | type | a user defined number. The sender can identify the message by means of that number. For messages sent to all plugins this should be unique. |
| IN: | len | length of the message |
| IN: | buf | the message |

| **void addedNode(osg::Node *node, coVRPlugin *myPlugin);** | | |
|---|---|---|
| Description: | informs other plugins that this plugin extended the scene graph | |
| IN: | node | pointer to the OpenSceneGraph node which was added |
| IN: | myPlugin | the plugin which added the node |

### 7.6.7 Append Buttons to the Pinboard

The plugin programmer has full access to the Virtual Reality User Interface library (OpenVRUI), which is used by OpenCOVER to generate the pinboard and its submenus. With this library simple user interface elements like labels and icons and more complex elements like menu items can be created. The OpenVRUI header files are located in covise/src/OpenCOVER/OpenVRUI. A small example how to use the library is available in the Cube plugin source code under covise/src/OpenCOVER/plugins/examples/Cube.

Sometimes a plugin needs to create own menu items or submenus and needs to add them to the main pinboard. The method getMenu returns a pointer to the pinboard menu. The plugin Cube is a small example which create an own submenu and add it to the pinboard.

| **coMenu *getMenu();** | |
|---|---|
| Description: | Get a pointer to the Pinboard. |
| Return Value: | pointer to the pinboard menu |

For compatibility with older COVISE Versions (< 5.0) the following functions for menu item generation are available:

| **void addToggleButton(const char *buttonName, const char *parentMenuName, int state, void *callback, void *classPtr, void *userData);** | | |
|---|---|---|
| Description: | append a switch button to a menu | |
| IN: | buttonName | the name of the button which is also the text on the button. Button names have to be unique. |
| IN: | parentMenuName | the name of the menu to which this button will be appended, codeNULL menas to append it to the main Pinboard |
| IN: | state | The state of the switch button, false=off, true=on |
| IN: | callback | function which is called on press/release |
| IN: | classPtr | ptr to the class which calls the callback |
| IN: | userData | ptr to data which are handed over to the callback |

| **void addGroupButton(const char *buttonName, const char *parentMenuName, int state, void *callback, int groupId, void *classPtr, void *userData);** | | |
|---|---|---|
| Description: | append a switch button to a menu | |
| IN: | buttonName | the name of the button which is also the text on the button. Button names have to be unique. |
| IN: | parentMenuName | the name of the menu to which this button will be appended, NULL means to append it to the main Pinboard |
| IN: | state | The state of the switch button, false=off, true=on |
| IN: | callback | function which is called on press/release |
| IN: | groupId | if a button is in the same group with others it is automatically switched off if another button is pressed. The group id of the navigation functions in the main Pinboard is zero. To create a new group id, use the method uniqueButtonGroup |
| IN: | classPtr | ptr to the class which calls the callback |
| IN: | userData | ptr to data which are handed over to the callback |

| **void addSubmenuButton(const char *buttonName, const char *parentMenuName, const char *subMenuName, int state, void *callback, int groupId, void *classPtr);** | | |
|---|---|---|
| Description: | append a switch button to a menu | |
| IN: | buttonName | the name of the button which is also the text on the button. Button names have to be unique. |
| IN: | parentMenuName | the name of the menu to which this button will be appended, NULL means to append it to the main Pinboard |
| IN: | subMenuName | name of the submenu which is also the header text of the submenu. Name has to be unique? |
| IN: | state | The state of the switch button, false=off, true=on |
| IN: | callback | function which is called on press/release |
| IN: | groupId | if a button is in the same group with others it is automatically switched off if another button is pressed. The group id of the navigation functions in the main Pinboard is zero. To create a new group id, use the method uniqueButtonGroup |
| IN: | classPtr | ptr to the class which calls the callback |

| **void setButtonState(const char *buttonName, int state);** | | |
|---|---|---|
| Description: | set the state of a switch or submenu button | |
| IN: | buttonName | the name of the button |
| IN: | state | The state of the button<br>switch button:<br>false=off, true=on<br>submenu:<br>false=closed, true=open |

| **addFunctionButton(const char *buttonName, const char *parentMenuName, *callback, void *classPtr);** | | |
|---|---|---|
| Description: | add a function button, a function button just calls the callback when pressed | |
| IN: | buttonName | the name of the button which is also the text on the button. Button names have to be unique. |
| IN: | parentMenuName | the name of the menu to which this button will be appended, NULL means to append it to the main Pinboard |
| IN: | callback | function which is called on press/release |
| IN: | classPtr | ptr to the class which calls the callback |

| **void addSliderButton(const char *buttonName, const char *parentMenuName,float min, float max, float value, void *callback, void *classPtr);** | | |
|---|---|---|
| Description: | add a slider to a menu | |
| IN: | buttonName | the name of the button which is also the text on the button. Button names have to be unique. |
| IN: | parentMenuName | the name of the menu to which this button will be appended, NULL means to append it to the main Pinboard |
| IN: | min | minimum slider value |
| IN: | max | maximum slider value |
| IN: | value | current slider value |
| IN: | callback | function which is if slider value changes |
| IN: | classPtr | ptr to the class which calls the callback |

| **void setSliderValue(const char *buttonName, float value);** | | |
|---|---|---|
| Description: | set slider value | |
| IN: | buttonName | the name of the button |
| IN: | value | current slider value |

| **void removeButton(const char *buttonName, const char *parentMenuName);** | | |
|---|---|---|
| Description: | remove a button<br>parentMenuName=NULL: remove it from main menu | |
| IN: | buttonName | the name of the button which is also the text on the button. Button names have to be unique. |
| IN: | parentMenuName | the name of the menu to which this button will be appended, NULL = append it to the main Pinboard |

## 7.7 Use OpenCOVER built-in interaction without the Pinboard

A plugin programmer may want to user OpenCOVER interaction although the pinboard is not visible, for example if the plugin implements a custom menu, but wants to use the navigation functions like XFORM, or the plugin wants to set the state of a button without using the menu.In this case the corresponding function needs to be configured in the section COVERPinboard. The Pinboard can be visible or invisible.

OpenCOVER supports the following functionality (see the usersguide for details):

Navigation:

- XFORM
- SCALE
- VIEW_ALL
- FREEZE
- COLLIDE

- FLY
- WALK
- DRIVE
- DRIVE_SPEED

Part Manipulation:

- SNAP
- REMOVE
- UNDO
- MOVE_PART

View Options:

- COORD_AXIS
- SPECULAR
- SPOTLIGHT
- STEREO_SEP

Collaborative Working:

- LOOSE_COUPLING
- TIGHT_COUPLING
- MASTERSLAVE_COUPLING
- SHOW_AVATAR

Scene Graph:

- STORE
- RELOAD_VRML

COVISE:

- EXECUTE
- CUTTINGSURFACE
- ISOSURFACEP
- TRACERUSG, STRACER, BLOCKSTRACER, TETRATRACE, MAGTRACER, MAGBLOCK-TRACER, CELLTRACER

Animation:

- FORWARD
- BACKWARD,
- ANIM_SPEED
- STEADYCAM

| **int isFunction(const char *functionName);** | | |
|---|---|---|
| Description: | Test if a function is configured and therefore in the pinboard | |
| IN: | functionName | The name of the function, for example "XFORM" |
| Return value: | true, if function is in pinboard | |

| **void setFunctionState(const char *functionName, int state);** | | |
|---|---|---|
| Description: | Set the state of a toggle, group, or submenu button and call the callback | |
| IN: | functionName | The name of the function, for example "FREEZE" |
| IN: | state | true/false |

| **void setFunctionValue(const char *functionName, float val);** | | |
|---|---|---|
| Description: | Set the value of a slider button and call the callback | |
| IN: | functionName | The name of the function, for example "DRIVE_SPEED" |
| IN: | value | value between min and max |

| **void doFunctionState(const char *functionName);** | | |
|---|---|---|
| Description: | Call the callback of a function button | |
| IN: | functionName | The name of the function, for example "XFORM" |

| **int getFunctionState(const char *functionName, int *state);** | | |
|---|---|---|
| Description: | Get the state of a toggle, group or submenu button | |
| IN: | functionName | The name of the function, for example "FREEZE" |
| IN: | state | provide a pointer for getting the current state |
| Return value: | true, if the function is in pinboard, false if not configured | |

| **int getFunctionValue(const char *functionName, float *value);** | | |
|---|---|---|
| Description: | Get the state of slider button | |
| IN: | functionName | The name of the function, for example "DRIVE_SPEED" |
| IN: | state | provide a pointer for getting the current value |
| Return value: | true, if the function is in pinboard, false if not configured | |

## 7.8 Controlling a COVISE module from within OpenCOVER

Module parameters are usually adjusted through the COVISE Map Editor or its Control Panel. If parameters should also be controllable from within OpenCOVER,

1. the module has to append the parameter information to the data objects and
2. OpenCOVER has to be extended by a plugin.

### 7.8.1 Class coFeedback

Feedback information is appended to data objects as attributes with a certain keyword. The class coFeedback handles the creation of such feedback-attributes:

In the constructor of coFeedback the name of the plugin has to be provided. When OpenCOVER receives this feedback information, it loads the appropriate plugin, if not already loaded (note that plugins can be loaded at starting time through the keyword MODULE in the section CoverConfig in the file covise.config).

| **coFeedback::coFeedback(const char *pluginName)** | | |
|---|---|---|
| Description: | Create a feedback object | |
| IN: | PluginName | The name of the plugin which should be loaded by OpenCOVER |

Parameters can now be added to such a feedback object with:

| **void coFeedback::addPara(coUifPara *parameter);** | | |
|---|---|---|
| Description: | add a parameter | |
| IN: | Parameter | The parameter which should be steerable from within the plugin |

In case that the module programmer needs to provide other information than parameters to the plugin he can use:

| **void coFeedback::addString(const char *userString);** | | |
|---|---|---|
| Description: | add a string | |
| IN: | UserString | Any string which is needed by the plugin |

The feedback object has to be applied to a COVISE data object. This data object can be one which is created by the module anyway or you can create a dummy output object, for example a single point and append it to this object.

| **void coFeedback:: apply(coDistributedObject *obj** | | |
|---|---|---|
| Description: | apply feedback object | |
| IN: | Obj | The COVISE data object to which the feedback is appended |

The feedback object can be deleted after it is appended:

| **coFeedback:: coFeedback(const char *pluginName)** | |
|---|---|
| Description: | Delete a feedback object |

### 7.8.2 Other functions

If an OpenCOVER plugin has the functions addObject and removeObject implemented, it is informed if OpenCOVER receives "ADD OBJECT" or "REMOVE OBJECT". If it has the function newInteractor implemented it is also informed, when the COVISE object has feedback attributes attached.

With the coFeedback::getXXX methods, the plugin can access the values of the module parameters, here one example for retreiving the values of a slider parameter:

| **void coInteractor::getFloatSliderParam(int paraNo, float min, float max, float val);** | | |
|---|---|---|
| Description: | Get the values of a slider parameter | |
| IN: | paraNo | The index of the parameter. The parameters are attached to the covise object with coFeedback::addPara(). paraNo is the index of the parameter in the list of the parameters which are added as feedback. it is not the parameter index in the module info window or the control panel. |
| IN: | min | The minimum value of the slider parameter |
| IN: | max | The maximum value of the slider parameter |
| IN: | min | The current value of the slider parameter |

The plugin can set module parameters with the coInteractor::setXXX methods:

| **void coInteractor::setSliderParam(const char *name,float min,float max, float value);** | | |
|---|---|---|
| Description: | Set the values of a slider parameter | |
| IN: | name | The parameter name. It needs to be exactly the name of the parameter of the module |
| IN: | min | The minimum value of the slider parameter |
| IN: | max | The maximum value of the slider parameter |
| IN: | min | The current value of the slider parameter |

Please look at covise/src/OpenCOVER/cover/coInteractor.h for the methods to receive and set also boolean, scalar, vector, string and choice parameters.

With newInteractor, addObject and removeObject a plugin is informed about any object which is added to OpenCOVER or removed from OpenCOVER. There are a few functions which can be used to find out from which module the coInteractor comes from or for which plugin it is intended:

| **const char *coInteractor::getPluginName();** | |
|---|---|
| Description: | Get the name of the plugin for which this coInteractor is intended |
| Return value: | The name of the plugin. |

| **const char *coInteractor::getModuleName();** | |
|---|---|
| Description: | Get the name of the module from which this coInteractor is generated |
| Return value: | The name of the module. |

| **int coInteractor::getModuleInstance();** | |
|---|---|
| Description: | Get the instance of the module from which this coInteractor is generated |
| Return value: | The instance of the module. |

### 7.8.3 A Plugin Programming Example

Have a look at the programming example in the directory covise/src/OpenCOVER/plugins/examples/Cube.

The plugin Cube implements interaction with the COVISE example module Cube. The Cube module generates a solid cube. The size and center of the cube can be manipulated through the module parameters.

The Cube plugin main class CubePlugin implements constructor and destructor and the virtual methods init, newInteractor, removeObject and preFrame. In init, OpenSceneGraph nodes are prepared to be added to the scene graph later. Through newInteractor the plugin can access the current value of the module parameters

and the current cube COVISE object. In removeObject the plugin is informed, when an object is deleted. In preFrame direct manipulation of a wireframe cube is implemented. We need this additional representation of the cube for direct manipulation because we cannot manipulate the parameters of the Cube module in realtime (the communication between OpenCOVER and the module is not fast enough). As soon as the manipulation is finished, the current size and center of this wireframe cube is sent back to the module.

### 7.8.4 Module Example

(from covise/src/application/examples/Cube)

```
#include "coFeedback.h"

// the ports and parameters of the module
coOutputPort *p_polyOut;
coFloatVectorParam *p_center;
coFloatParam *p_cusize;

// in the compute callback we create an output object
polygonObj = new DO_Polygons(polygonObjName, 8, xCoords, yCoords, zCoords, 24,
vertexList, 6, polygonList);

// interaction info for OpenCOVER
coFeedback feedback("CubePlugin");
feedback.addPara(p_center);
feedback.addPara(p_cusize);
feedback.addString("Test the user string as well");
feedback.apply(polygonObj);

// apply the object to the port
p_polyOut->setObj(polygonObj);
```

## 7.9 Utility Classes

### 7.9.1 The classes coIntersection and coAction

The classes coIntersection and coAction provide intersection testing of a node in your scene graph with the pointer ray.

| **virtual int coAction::hit(osg::Vec3 &hitPoint, osgUtil::Hit *hit)** | | |
|---|---|---|
| Description: | hit is called whenever the node or any node with this action is intersected | |
| IN: | hitPoint | intersection point in world coordinates |
| IN: | hit | see man osgUtil::Hit |
| Return value: | return ACTION_CALL_ON_MISS if you want miss to be called, otherwise return ACTION_DONE | |

| **virtual void coAction::miss()** | |
|---|---|
| Description: | miss is called once after a hit if the node is not intersected any more |
| Return value: | return ACTION_CALL_ON_MISS if you want miss to be called, otherwise return ACTION_DONE |

Append the node which you want to have tested for intersection to the global intersection list:

| **void coIntersection::add(osg::Node *node, coAction *action)** | | |
|---|---|---|
| Description: | you can add a node to the global intersection list **intersector** | |
| IN: | node | the node you want to have tested for intersection |
| IN: | action | your class which is derived from coAction and which contains the functions hit and miss |

| **void coIntersection::remove(osg::Node *node)** | | |
|---|---|---|
| Description: | remove a node from the global intersection list **intersector** | |
| IN: | node | the node you want to have tested for intersection |

## Example:

```
#include <coInteraction.h>

class myClass: public coAction
{
private:
   osg::MatrixTransform *transform;

public:
   myClass()
{
       //...
       intersector.add(transform, this);
       //...
};
   ~myClass()
{
       //...
       interscetor.remove(transform);
       //...
};
   virtual int hit(osg::Vec3 &hitPoint, osgUtil::Hit *hit)
{
       //...
       fprintf(stderr, "my transform was hit\n");
       return ACTION_CALL_ON_MISS;
};

   virtual void miss()
{
       //...
       fprintf(stderr, "my transform was missed\n");
};
}
```

### 7.9.2 OpenCOVER in a clustered environment

### 7.9.3 opencover::coVRPluginSupport Class Reference

```
#include <cover/coVRPluginSupport.h>
```

**Public Member Functions**

- bool debugLevel (int level) const
- void initUI ()
- std::ostream & notify (Notify::NotificationLevel level=Notify::Info) const
- std::ostream & notify (Notify::NotificationLevel level, const char ∗format,...) const
- INTERNAL int getNumClipPlanes ()
- osg::ClipPlane ∗ getClipPlane (int num)
- void preparePluginUnload ()
- bool isClippingOn () const
- int getActiveClippingPlane () const
- void setActiveClippingPlane (int plane)
- osg::Group ∗ getScene () const
- osg::ClipNode ∗ getObjectsRoot () const
- osg::MatrixTransform ∗ getPointer () const
- const osg::Matrix & getPointerMat () const
- const osg::Matrix & getMouseMat () const
- const osg::Matrix & getRelativeMat () const
- osg::MatrixTransform ∗ getObjectsXform () const
- const osg::Matrix & getXformMat () const
- void setXformMat (const osg::Matrix &mat)
- osg::MatrixTransform ∗ getObjectsScale () const
- void setScale (float s)
- float getScale () const
- const osg::Matrix & getBaseMat () const
- const osg::Matrix & getInvBaseMat () const
- void watchFileDescriptor (int fd)
- void unwatchFileDescriptor (int fd)
- vrui::coUpdateManager ∗ getUpdateManager () const
- float getSceneSize () const
- bool isHighQuality () const
- bool isVRBconnected ()
- bool sendVrbMessage (const covise::MessageBase ∗msg) const
- const osg::Matrix & getViewerMat () const
- void setNodesIsectable (osg::Node ∗n, bool isect)
- coPointerButton ∗ getPointerButton () const
- coPointerButton ∗ getMouseButton () const
- coPointerButton ∗ getRelativeButton () const
- vrui::coMenu ∗ getMenu ()
- osg::Group ∗ getMenuGroup () const
- coVRPlugin ∗ addPlugin (const char ∗name)
- coVRPlugin ∗ getPlugin (const char ∗name)
- void removePlugin (coVRPlugin ∗)
- int removePlugin (const char ∗name)
- void addedNode (osg::Node ∗node, coVRPlugin ∗myPlugin)
- bool removeNode (osg::Node ∗node, bool isGroup=false)

- void sendMessage (coVRPlugin *sender, int toWhom, int type, int len, const void *buf)
- void sendMessage (coVRPlugin *sender, const char *destination, int type, int len, const void *buf, bool localonly=false)
- bool grabKeyboard (coVRPlugin *)
- void releaseKeyboard (coVRPlugin *)
- bool isKeyboardGrabbed ()
- bool grabViewer (coVRPlugin *)
- void releaseViewer (coVRPlugin *)
- bool isViewerGrabbed () const
- void protectScenegraph ()
- double frameTime () const
- double frameDuration () const
- double frameRealTime () const
- osgViewer::GraphicsWindow::MouseCursor getCurrentCursor () const
- void setCurrentCursor (osgViewer::GraphicsWindow::MouseCursor type)
- void setCursorVisible (bool visible)
- osg::Node * getIntersectedNode () const
- const osg::NodePath & getIntersectedNodePath () const
- const osg::Vec3 & getIntersectionHitPointWorld () const
- const osg::Vec3 & getIntersectionHitPointWorldNormal () const
- osg::Matrix updateInteractorTransform (osg::Matrix mat, bool usePointer) const
- INTERNAL INTERNAL vrui::coToolboxMenu * getToolBar (bool create=false)
- void setToolBar (vrui::coToolboxMenu *tb)
- void setFrameTime (double ft)
- void setRenderStrategy (osg::Drawable *draw, bool dynamic=false)
- opencover::coVRMessageSender * getSender ()
- bool sendGrMessage (const grmsg::coGRMsg &grmsg, int msgType=covise::COVISE_MESSAGE_UI) const

**Static Public Member Functions**

- static double currentTime ()

**Friends**

- class OpenCOVER
- class fasi
- class fasi2
- class mopla
- class coVRMSController
- class coIntersection

**Detailed Description**

Provide a stable interface and a single entry point to the most import OpenCOVER functions

**Member Function Documentation**

**addedNode()** `void opencover::coVRPluginSupport::addedNode (`
`osg::Node * node,`
`coVRPlugin * myPlugin )`

informs other plugins that this plugin extended the scene graph

**addPlugin()** `coVRPlugin* opencover::coVRPluginSupport::addPlugin (`
`const char * name )`

load a new plugin

**currentTime()** `static double opencover::coVRPluginSupport::currentTime ( ) [static]`

returns the current time in seconds since Jan. 1, 1970, if possible, use frameTime() as it does not require a system call

Returns

number of seconds since Jan. 1, 1970

**debugLevel()** `bool opencover::coVRPluginSupport::debugLevel (`
`int level ) const`

returns true if level <= debugLevel

debug levels should be used like this: 0 no output, 1 covise.config entries, coVRInit, 2 constructors, destructors, 3 all functions which are not called continously, 4, 5 all functions which are called continously

**frameDuration()** `double opencover::coVRPluginSupport::frameDuration ( ) const`

returns the duration of the last frame in seconds

Returns

render time of the last frame in seconds

**frameRealTime()** `double opencover::coVRPluginSupport::frameRealTime ( ) const`

returns the time in seconds since Jan. 1, 1970 at the beginning of this frame, use this function if you can since it is faster than currentTime(),

**frameTime()** `double opencover::coVRPluginSupport::frameTime ( ) const`

returns the time in seconds since Jan. 1, 1970 at the beginning of this frame, use this function if you can since it is faster than currentTime(), this is the time for which the rendering should be correct, might differ from system time

Returns

number of seconds since Jan. 1, 1970 at the beginning of this frame

**getActiveClippingPlane()** `int opencover::coVRPluginSupport::getActiveClippingPlane ( ) const`

return number of clipping plane user is possibly interacting with

**getBaseMat()** `const osg::Matrix& opencover::coVRPluginSupport::getBaseMat ( ) const [inline]`

transformation matrix from object coordinates to world coordinates

multiplied matrices from scene node to objects root node

**getClipPlane()** `osg::ClipPlane* opencover::coVRPluginSupport::getClipPlane ( int num ) [inline]`

return pointer to a clipping plane

**getCurrentCursor()** `osgViewer::GraphicsWindow::MouseCursor opencover::coVRPluginSupport::getCurrentCursor ( ) const`

get the number of the active cursor shape

**getIntersectedNode()** `osg::Node* opencover::coVRPluginSupport::getIntersectedNode ( ) const`

get node currently intersected by pointer

**getIntersectedNodePath()** `const osg::NodePath& opencover::coVRPluginSupport::getIntersectedNodePath ( ) const`

get path to node currently intersected by pointer

**getIntersectionHitPointWorld()** `const osg::Vec3& opencover::coVRPluginSupport::getIntersectionHitPointWorld ( ) const`

get world coordinates of intersection hit point

**getIntersectionHitPointWorldNormal()** `const osg::Vec3& opencover::coVRPluginSupport::getIntersectionHitPointWorldNorma ( ) const`

get normal of intersection hit

**getInvBaseMat()** `const osg::Matrix& opencover::coVRPluginSupport::getInvBaseMat ( ) const`

transformation from world coordinates to object coordinates

use this cached value instead of inverting getBaseMat() yourself

**getMenu()** `vrui::coMenu* opencover::coVRPluginSupport::getMenu ( )`

returns the COVER Menu (Pinboard)

**getMenuGroup()** `osg::Group* opencover::coVRPluginSupport::getMenuGroup ( ) const`

return group node of menus

**getMouseButton()** `coPointerButton* opencover::coVRPluginSupport::getMouseButton ( ) const`
returns a pointer to a coPointerButton object representing the mouse buttons state

**getMouseMat()** `const osg::Matrix& opencover::coVRPluginSupport::getMouseMat ( ) const`
get matrix of current 2D mouse matrix (the same as getPointerMat for MOUSE tracking)

**getNumClipPlanes()** `INTERNAL int opencover::coVRPluginSupport::getNumClipPlanes ( )`
return the number of clipPlanes reserved for the kernel, others are available to VRML ClippingPlane Node

**getObjectsRoot()** `osg::ClipNode* opencover::coVRPluginSupport::getObjectsRoot ( ) const`
get the group node for all COVISE and model geometry

**getObjectsScale()** `osg::MatrixTransform* opencover::coVRPluginSupport::getObjectsScale ( ) const`
get the MatrixTransform for objects scaling

**getObjectsXform()** `osg::MatrixTransform* opencover::coVRPluginSupport::getObjectsXform ( ) const`
get the MatrixTransform for objects translation and rotation

**getPlugin()** `coVRPlugin* opencover::coVRPluginSupport::getPlugin ( const char * name )`
get plugin called name

**getPointer()** `osg::MatrixTransform* opencover::coVRPluginSupport::getPointer ( ) const`
get the MatrixTransform node of the hand

**getPointerButton()** `coPointerButton* opencover::coVRPluginSupport::getPointerButton ( ) const`
returns a pointer to a coPointerButton object for the main button device

**getPointerMat()** `const osg::Matrix& opencover::coVRPluginSupport::getPointerMat ( ) const`
get matrix of hand transform (same as getPointer()->getMatrix())

**getRelativeButton()** `coPointerButton* opencover::coVRPluginSupport::getRelativeButton ( ) const`
returns a pointer to a coPointerButton object representing the buttons state on the relative input device

**getRelativeMat()** `const osg::Matrix& opencover::coVRPluginSupport::getRelativeMat ( ) const`

get matrix for relative input (identity if no input)

**getScale()** `float opencover::coVRPluginSupport::getScale ( ) const`

get the scale factor of the scale node

**getScene()** `osg::Group* opencover::coVRPluginSupport::getScene ( ) const`

get scene group node

**getSceneSize()** `float opencover::coVRPluginSupport::getSceneSize ( ) const`

get the scene size defined in covise.config

**getSender()** `opencover::coVRMessageSender* opencover::coVRPluginSupport::getSender ( )`

**getToolBar()** `INTERNAL INTERNAL vrui::coToolboxMenu* opencover::coVRPluginSupport::getToolBar ( bool create = false )`

**getUpdateManager()** `vrui::coUpdateManager* opencover::coVRPluginSupport::getUpdateManager ( ) const`

**getViewerMat()** `const osg::Matrix& opencover::coVRPluginSupport::getViewerMat ( ) const`

get the position and orientation of the user in world coordinates

**getXformMat()** `const osg::Matrix& opencover::coVRPluginSupport::getXformMat ( ) const`

same as getObjectsXform()->getMatrix()

**grabKeyboard()** `bool opencover::coVRPluginSupport::grabKeyboard ( coVRPlugin * )`

grab keyboard input

other plugins will not get key event notifications, returns true if keyboard could be grabbed, returns false if keyboard is already grabbed by another plugin

**grabViewer()** `bool opencover::coVRPluginSupport::grabViewer ( coVRPlugin * )`

let plugin request control over viewer position

**initUI()** `void opencover::coVRPluginSupport::initUI ( )`

**isClippingOn()** `bool opencover::coVRPluginSupport::isClippingOn ( ) const`

returns true if clipping is on

**isHighQuality()** `bool opencover::coVRPluginSupport::isHighQuality ( ) const`

favor high-quality rendering instead of interactivity

**isKeyboardGrabbed()** `bool opencover::coVRPluginSupport::isKeyboardGrabbed ( )`

check if keyboard is grabbed

**isViewerGrabbed()** `bool opencover::coVRPluginSupport::isViewerGrabbed ( ) const`

whether a plugins controls viewer position

**isVRBconnected()** `bool opencover::coVRPluginSupport::isVRBconnected ( )`

**notify()** **[1/2]** `std::ostream& opencover::coVRPluginSupport::notify ( Notify::NotificationLevel` *`level = Notify::Info`* `) const`

**notify()** **[2/2]** `std::ostream& opencover::coVRPluginSupport::notify ( Notify::NotificationLevel` *`level,`* `const char *` *`format,`* `... ) const`

**preparePluginUnload()** `void opencover::coVRPluginSupport::preparePluginUnload ( )`

**protectScenegraph()** `void opencover::coVRPluginSupport::protectScenegraph ( )`

forbid saving of scenegraph

**releaseKeyboard()** `void opencover::coVRPluginSupport::releaseKeyboard ( coVRPlugin * )`

release keyboard input, all plugins will get key events

**releaseViewer()** `void opencover::coVRPluginSupport::releaseViewer ( coVRPlugin * )`

release control over viewer position

**removeNode()** `bool opencover::coVRPluginSupport::removeNode (`

```
osg::Node * node,
bool isGroup = false )
```

remove node from the scene graph,

use this method when removing nodes from the scene graph in order to update OpenCOVER's internal state

Returns

if a node was removed

**removePlugin() [1/2]** `void opencover::coVRPluginSupport::removePlugin (`

```
coVRPlugin * )
```

remove the plugin by pointer

**removePlugin() [2/2]** `int opencover::coVRPluginSupport::removePlugin (`

```
const char * name )
```

remove a plugin by name

**sendGrMessage()** `bool opencover::coVRPluginSupport::sendGrMessage (`

```
const grmsg::coGRMsg & grmsg,
int msgType = covise::COVISE_MESSAGE_UI ) const
```

**sendMessage() [1/2]** `void opencover::coVRPluginSupport::sendMessage (`

```
coVRPlugin * sender,
int toWhom,
int type,
int len,
const void * buf )
```

send a message to other plugins

**sendMessage() [2/2]** `void opencover::coVRPluginSupport::sendMessage (`

```
coVRPlugin * sender,
const char * destination,
int type,
int len,
const void * buf,
bool localonly = false )
```

send a message to a named plugins

**sendVrbMessage()** `bool opencover::coVRPluginSupport::sendVrbMessage (`

```
const covise::MessageBase * msg ) const
```

send a message either via COVISE connection or via tcp to VRB

**setActiveClippingPlane()** `void opencover::coVRPluginSupport::setActiveClippingPlane ( int plane )`

set number of clipping plane user is possibly interacting with

**setCurrentCursor()** `void opencover::coVRPluginSupport::setCurrentCursor ( osgViewer::GraphicsWindow::MouseCursor type )`

set cursor shape

Parameters

| | |
|---|---|
| *type* | number of cursor shape |

**setCursorVisible()** `void opencover::coVRPluginSupport::setCursorVisible ( bool visible )`

make the cursor visible or invisible

**setFrameTime()** `void opencover::coVRPluginSupport::setFrameTime ( double ft )`

use only during coVRPlugin::update()

**setNodesIsectable()** `void opencover::coVRPluginSupport::setNodesIsectable ( osg::Node * n, bool isect )`

search geodes under node and set Visible bit in node mask

**setRenderStrategy()** `void opencover::coVRPluginSupport::setRenderStrategy ( osg::Drawable * draw, bool dynamic = false )`

**setScale()** `void opencover::coVRPluginSupport::setScale ( float s )`

set the scale matrix of the scale node

**setToolBar()** `void opencover::coVRPluginSupport::setToolBar ( vrui::coToolboxMenu * tb )`

**setXformMat()** `void opencover::coVRPluginSupport::setXformMat ( const osg::Matrix & mat )`

same as getObjectsXform()->setMatrix()

**unwatchFileDescriptor()** `void opencover::coVRPluginSupport::unwatchFileDescriptor ( int fd )`

remove fd from filedescriptors to watch

**updateInteractorTransform()** `osg::Matrix opencover::coVRPluginSupport::updateInteractorTransform ( osg::Matrix mat, bool usePointer ) const`

update matrix of an interactor, honouring snapping, ...

**watchFileDescriptor()** `void opencover::coVRPluginSupport::watchFileDescriptor ( int fd )`

register filedescriptor fd for watching so that scene will be re-rendererd when it is ready

### Friends And Related Function Documentation

**coIntersection** `friend class coIntersection [friend]`

**coVRMSController** `friend class coVRMSController [friend]`

**fasi** `friend class fasi [friend]`

**fasi2** `friend class fasi2 [friend]`

**mopla** `friend class mopla [friend]`

**OpenCOVER** `friend class OpenCOVER [friend]`

## 7.9.4 opencover::coPointerButton Class Reference

`#include <cover/coVRPluginSupport.h>`

### Public Member Functions

- coPointerButton (const std::string &name)
- ∼coPointerButton ()
- unsigned int getState () const
- unsigned int oldState () const
- unsigned int wasPressed (unsigned int buttonMask=vrui::vruiButtons::ALL_BUTTONS) const
- unsigned int wasReleased (unsigned int buttonMask=vrui::vruiButtons::ALL_BUTTONS) const

- bool notPressed () const
- int getWheel (size_t idx=0) const
- void setWheel (size_t idx, int count)
- const std::string & name () const

### Friends

- class coVRPluginSupport
- class coVRMSController

### Detailed Description

Access to buttons and wheel of interaction devices

### Constructor & Destructor Documentation

**coPointerButton()** `opencover::coPointerButton::coPointerButton ( const std::string & name )`

**~coPointerButton()** `opencover::coPointerButton::~coPointerButton ( )`

### Member Function Documentation

**getState()** `unsigned int opencover::coPointerButton::getState ( ) const`

button state

Returns

button press mask

**getWheel()** `int opencover::coPointerButton::getWheel ( size_t idx = 0 ) const`

accumulated number of wheel events

**name()** `const std::string& opencover::coPointerButton::name ( ) const`

button name

**notPressed()** `bool opencover::coPointerButton::notPressed ( ) const`

is no button pressed

**oldState()** `unsigned int opencover::coPointerButton::oldState ( ) const`

previous button state

Returns

old button state

**setWheel()** `void opencover::coPointerButton::setWheel (`
`size_t idx,`
`int count )`

set number wheel events

**wasPressed()** `unsigned int opencover::coPointerButton::wasPressed (`
`unsigned int buttonMask = vrui::vruiButtons::ALL_BUTTONS ) const`

buttons pressed since last frame

**wasReleased()** `unsigned int opencover::coPointerButton::wasReleased (`
`unsigned int buttonMask = vrui::vruiButtons::ALL_BUTTONS ) const`

buttons released since last frame

#### Friends And Related Function Documentation

**coVRMSController** `friend class coVRMSController [friend]`

**coVRPluginSupport** `friend class coVRPluginSupport [friend]`

### 7.9.5 opencover::Isect Struct Reference

`#include <coVRPluginSupport.h>`

#### Public Types

- enum IntersectionBits {
  Collision = 1, Intersection = 2, Walk = 4, Touch = 8,
  Pick = 16, Visible = 32, NoMirror = 64, Left = 128,
  Right = 256, CastShadow = 512, ReceiveShadow = 1024, Update = 2048,
  OsgEarthSecondary = 0x80000000 }

#### Member Enumeration Documentation

**IntersectionBits** `enum opencover::Isect::IntersectionBits`

Enumerator

| | |
|---|---|
| Collision | |
| Intersection | |
| Walk | |
| Touch | |
| Pick | |
| Visible | |
| NoMirror | |
| Left | |
| Right | |
| CastShadow | |
| ReceiveShadow | |
| Update | |
| OsgEarthSecondary | |

# 8 Quick Reference

## 8.1 coModule

**Base class for all module programming**

```
coModule(const char *description);
virtual ~coModule();

virtual void start(int argc, char *argv[]);
virtual int compute(const char*);
virtual void param(const char *paramName);
virtual void sockData(int sockNo);
virtual void quit(void*);
virtual void postInst();
virtual void mainLoop();

coBooleanParam *addBooleanParam(const char *name, const char *desc);
coFileBrowserParam *addFileBrowserParam(const char *name, const char *desc);
coChoiceParam *addChoiceParam(const char *name, const char *desc);
coFloatParam *addFloatParam(const char *name,
                                        const char *desc);
coFloatSliderParam *addFloatSliderParam(const char *name,
                                        const char *desc);
coFloatVectorParam *addFloatVectorParam(const char *name,
                                        const char *desc);
coInt32Param *addInt32Param(const char *name,
                                    const char *desc);
coIntSliderParam *addIntSliderParam(const char *name,
                                    const char *desc);
coIntVectorParam *addInt32VectorParam(const char *name,
                                    const char *desc);
coStringParam *addStringParam(const char *name, const char *desc);
coInputPort *addInPort(const char *name, const char *types,
                                      const char *desc);
coOutputPort *addOutputPort(const char *name, const char *types,
                                        const char *desc);

coChoiceParam *paraSwitch(const char *name, const char *desc);
int paraEndSwitch();
int paraCase(const char *name);
int paraEndCase();

static void sendError(const char *format, \dots);
static void sendWarning(const char *format, \dots);
static void sendInfo(const char *format, \dots);

void selfExec();

void addSocket(int socket);
```

```
void removeSocket(int socket);
```

## 8.2 coSimpleModule

**This class overloads coModule to automatically recurse through coDoSet hierarchies.**

```
class coSimpleModule : public coModule

void setComputeTimesteps(const int v);
void setComputeMultiblock(const int v);
void copyAttributes(coDistributedObject *tgt, coDistributedObject *src);
void setCopyAttributes(const int v);
```

## 8.3 coSimLib

**This class overloads coModule to automatically recurse through coDoSet hierarchies.**

```
class coSimLib : public coModule

coSimLib(const char *moduleName);
int setTargetHost(const char *hostname);
int setLocalHost(const char *hostname);
int setUserArg(int num, const char *data);
int startSim();
int serverMode();
int isConnected();
int recvData(void *buffer, size_t length);
int sendData(const void *buffer, size_t length);
int coParallelInit(int numParts, int numPorts);
int coParallelPort(const char *portName, int isCellData);
int setParaMap(int isCell,int isFortran, int nodeNo, int length,
               int32 *nodeMap);
```

## 8.4 Simlib client

**This is the client-side connectivity for a coSimLib module: These are C, not C++ routines, with additional FORTRAN77 language binding, which have the same parameters but 6-char names. The file coSimClient.o from the covise/$ARCH/bin directory must be linked to the simulation.**

```
int coNotConnected()                                   F77: CONOCO

/* Logic send/receive calls ******/

int coGetParaSlider(const char *name,                  F77: COGPSL
                    float *min, float *max, float *val);
int coGetParaFloatScalar(const char *name, float *data);  F77: COGPFL
int coGetParaIntScalar(const char *name, int *data);F77:  F77: COGPIN
int coGetParaChoice(const char *name, int *data);      F77: COGPCH
int coGetParaBool(const char *name, int *data);        F77: COGPBO
int coGetParaText(const char *name, char *data);       F77: COGPTX
```

```
int coGetParaFile(const char *name, int *data);           F77: COGPFI
int coSend1Data(const char *portName,                     F77: COSU1D
                int numElem, float *data);
int coSend3Data(const char *portName,                     F77: COSU3D
                int numElem, float *data);
int coExecModule();                                       F77: COEXEC
int coFinished();                                         F77: COFINI
int coParallelInit(int numParts, int numPorts);           F77: COPAIN
int coParallelPort(const char *portname, int isCellData); F77: COPAPO
int coParallelCellMap(int node, int numCells,             F77: COPACM
                      const int *localToGlobal);
int coParallelVertexMap(int node, int numCells,           F77: COPAVM
                        const int *localToGlobal);
int coParallelNode(int node);                             F77: COPANO
int sendData(const void *buffer, size_t length);          F77: COSEND
int recvData(void *buffer, size_t length);                F77: CORECV
int getVerboseLevel();
                                                          F77: COVERB
```

## 8.5 coUifElem

**Base class for all ports, Parameters and switch groups nodes**

```
enum Kind { SWITCH, PARAM, INPORT, OUTPORT };
virtual void hide();
virtual void show();
virtual Kind kind();
virtual const char *getName();
```

## 8.6 coPort

**Base class for In-, Out- and Parameter ports**

```
class coPort : public coUifElem

virtual const char *getDesc() const;
```

## 8.7 coInputPort

**Input data port**

```
class coInputPort : public coPort

void setRequired(int isRequired);
coDistributedObject *getCurrentObject();
```

## 8.8 coOutputPort

**Output data port**

```
class coOutputPort : public coPort

void setRequired(int isRequired);
void setObj(coDistributedObject *obj);
coDistributedObject *getObj();
const char *getObjName();
```

## 8.9 coUifPara

**Base class for all parameter ports**

```
class coUifPara : public coPort

void setImmediate(int isImmediate);
void setActive(int isActive);
virtual void hide();
virtual void show();
void enable();
void disable();
int isActive() const;
```

## 8.10 coBooleanParam

**Boolean value parameter**

```
class coBooleanParam: public coUifPara

int setValue(int value);
int getValue() const;
```

## 8.11 coFileBrowserParam

**File browser parameter**

```
class coFileBrowserParam : public coUifPara

int setValue(const char *path, const char *value);
const char *getValue() const;
```

## 8.12 coChoiceParam

**Parameter to choose values from a list**

```
class coChoiceParam : public coUifPara

int setValue(int numChoices, const char * const* choice,
             int actChoice);
```

```
int updateValue(int numChoices, const char * const* choice,
             int actChoice);

int setValue(int actChoice);
int getValue() const;
```

## 8.13 coFloatParam

**single float parameter**

```
class coFloatParam : public coUifPara

int setValue(float val);
float getValue() const;
```

## 8.14 coFloatSliderParam

**float slider parameter**

```
class coFloatSliderParam : public coUifPara

int setValue(float min, float max, float value);
int setMin(float min);
int setMax(float max);
int setValue(float value);
void getValue(float &min, float &max, float &value) const;
float getMin() const;
float getMax() const;
float getValue() const;
```

## 8.15 coFloatVectorParam

**Multiple float parameters**

```
class coFloatVectorParam : public coUifPara

int setValue(int pos, float data);
int setValue(int size, const float *data);
int setValue(float data0, float data1, float data2);
float getValue(int pos) const;
int getValue(float &data0, float &data1, float &data2) const;
```

## 8.16 coInt32Param

**single integer parameter**

```
class coInt32Param : public coUifPara

int setValue(int val);
int getValue() const;
```

## 8.17 coIntSliderParam

**Integer slider parameter**

```
class coIntSliderParam : public coUifPara

int setValue(int min, int max, int value);
int setMin(int min);
int setMax(int max);
int setValue(int value);

void getValue(int &min, int &max, int &value) const;
int getMin() const;
int getMax() const;
int getValue() const;
```

## 8.18 coIntVectorParam

**Parameter for multiple integers**

```
class coIntVectorParam : public coUifPara

int setValue(int pos, int data);
int setValue(int size, const int *data);
int setValue(int data0, int data1, int data2);
int getValue(int pos) const;
```

## 8.19 coStringParam

**Implements string parameters**

```
class coStringParam : public coUifPara

int setValue(const char *val);
const char *getValue() const;
```

## 8.20 coDistributedObject

**Base class for all data objects**

```
void setAttribute(const char *, const char *);
void setAttributes(int, const char **, const char **);
const char *getAttribute(const char *);
int getAllAttributes(const char ***name,
                     const char ***content);
const char *getName() { return name; }
const char *getType() const;
int isType(const char *reqType);
char objectOk() const;
char *getName();
char *getType();
```

## 8.21 coDoUniformGrid

```
coDoUniformGrid(const char *name, int x, int y, int z,
               float xmin, float xmax, float ymin,
               float ymax, float zmin, float zmax);
void getGridSize(int *x, int *y, int *z);
void getPointCoordinates(int i, float *x_c,
                           int j, float *y_c,
                           int k, float *z_c);
void getDelta(float *dx, float *dy, float *dz);
void getMinMax(float *xmin, float *xmax,
                 float *ymin, float *ymax,
                 float *zmin, float *zmax);
```

## 8.22 coDoRectilinearGrid

```
coDoRectilinearGrid(const char *name, int x, int y, int z);
coDoRectilinearGrid(const char *name, int x, int y, int z,
                   float *xc, float *yc, float *zc);
void getGridSize(int *x, int *y, int *z);
void getPointCoordinates(int i, float *x_c,
                           int j, float *y_c,
                           int k, float *z_c);
void getAddresses(float **x_c, float **y_c, float **z_c);
```

## 8.23 coDoStructuredGrid

```
coDoStructuredGrid(const char *name, int x, int y, int z);
coDoStructuredGrid(const char *name, int x, int y, int z,
                     float *xc, float *yc, float *zc);
void getGridSize(int *x, int *y, int *z);
void getPointcoordinates(int i, float *x_c,
                           int j, float *y_c,
                           int k, float *z_c);
void getAddresses(float **x_c, float **y_c, float **z_c);
```

## 8.24 coDoUnstructuredGrid

```
coDoUnstructuredGrid(const char *name,
                    int nelem, int nconn, int ncoord, int ht);
coDoUnstructuredGrid(const char *name,
                    int nelem, int nconn, int ncoord,
                    int *el, int *cl,
                    float *xc, float *yc, float *zc,
                    int *tl);
void getGridSize(int *e, int *c, int *p);
void getAddresses(int **elem, int **conn,
                  float **x_c, float **y_c, float **z_c);
void getTypeList(int **l);
int hasTypeList();
void getNeighborList(int *n, int **l, int **li);
```

## 8.25 coDoFloat

```
coDoFloat(const char *n, int no, float *s);
coDoFloat(const char *n, int no);

int getNumPoints();
void getPointValue(int no, float *s);
void getAddress(float **data);
```

## 8.26 coDoVec2

```
coDoVec2(const char *n, int no, float *s, float *t);
coDoVec2(const char *n, int no);

int getNumPoints();    // returns gridsize
void getPointValue(int no, float *s, float *t);
void getAddresses(float **s_d, float **t_d);
```

## 8.27 coDoVec3

```
coDoVec3(const char *n, int no,
                        float *xc, float *yc, float *zc);
coDoVec3(const char *n, int no);

int getNumPoints();
void getPointValue(int no, float *s);
void getAddresses(float **u_v, float **v_v, float **w_v);
```

## 8.28 coDoTensor

```
coDoTensor(const char *n, int no,
                        float *data, TensorType ttype);
coDoTensor(const char *n, int no, TensorType ttype);

int getNumPoints();
coDoTensor::TensorType getTensorType();
void getPointValue(int no, float *s);
void getAddress(float **value);
```

## 8.29 coDoRGBA

```
coDoRGBA(const char *n, int no, int *pc);
coDoRGBA(const char *n, int no);

int getNumElements();    // returns gridsize
void getElementValue(int no, int *s);
void getAddress(int **pc);

int setFloatRGBA(int pos, float r, float g, float b, float a = 1.0);
```

```
int setIntRGBA(int pos, int r, int g, int b, int a = 255);
int getFloatRGBA(int pos, float *r, float *g, float *b, float *a);
int getIntRGBA(int pos, int *r, int *g, int *b, int *a);
```

## 8.30 coDoGeometry

```
coDoGeometry(const char *n, coDistributedObject *geo);

void setColor(int cattr, coDistributedObject *c);
void setNormal(int nattr, coDistributedObject *n);
void setTexture(int tattr, coDistributedObject *t) ;

coDistributedObject *getGeometry();
coDistributedObject *getColors();
coDistributedObject *getNormals();
coDistributedObject *getTexture();

int getGeometryType();
int getColorAttributes();
int getNormalAttributes();
int getTextureAttributes();

void setColorAttributes(int cattr);
void setNormalAttributes(int nattr);
void setTextureAttributes(int nattr);
```

## 8.31 coDoPoints

```
coDoPoints(char *n, int no);
coDoPoints(char *n, int no,
          float *x, float *y, float *z);

int getNumPoints();
void getPointCoordinates(int no, float *xc, float *yc, float *zc);
void getAddresses(float **x_c, float **y_c, float **z_c);
```

## 8.32 coDoLines

```
coDoLines(char *n, int no_p, int no_v, int no_l);
coDoLines(char *n,
         int no_p, float *x_c, float *y_c, float *z_c,
         int no_v, int *v_l, int no_l, int *l_l);

int getNumLines();
int getNumVertices();
int getNumPoints();
void getAddresses(float **x_c, float **y_c, float **z_c,
                  int **v_l, int **l_l);
```

## 8.33 coDoPolygons

```
coDoPolygons(char *n, int no_p, int no_v, int no_l);
coDoPolygons(char *n,
            int no_p, float *x_c, float *y_c, float *z_c,
            int no_v, int *v_l, int no_pol, int *pol_l);

int getNumPolygons();
int getNumVertices();
int getNumPoints();
void getNeighborList(int *n, int **l, int **li);
void getAddresses(float **x_c, float **y_c, float **z_c,
                  int **v_l, int **l_l);
```

## 8.34 coDoTriangleStrips

```
coDoTriangleStrips(char *n, int no_p, int no_v, int no_l);
coDoTriangleStrips(char *n,
                  int no_p,float *x_c,float *y_c, float *z_c,
                  int no_v, int *v_l, int no_pol, int *pol_l);

int getNumStrips();
int getNumVertices();
int getNumPoints();
void getAddresses(float **x_c, float **y_c, float **z_c,
                  int **v_l, int **l_l) ;
```

## 8.35 coDoTexture

```
coDoTexture(const char *name,coDoPixelImage* texture,
           int border, int components, int level,
           int numVertices, int* vertexIndex,
           int numCoord, float** coords)

coDoPixelImage* getBuffer();
int            getBorder();
int            getComponents();
int            getLevel();
int            getWidth();
int            getHeight();
int*           getVertices();
int            getNumCoordinates();
float**        getCoordinates();
```

## 8.36 coDoPixelImage

```
coDoPixelImage(const char *name, int width, int height,
              unsigned format, short pixelsize, const char **buffer);
coDoPixelImage(const char *name, int width, int height,
              unsigned format, short pixelsize, const char *buffer);
```

```
coDoPixelImage(const char *name, int width, int height,
              unsigned format, short pixelsize);

int getWidth();
int getHeight();
short getPixelsize();
unsigned getFormat();
char* getPixels();
char& operator() (int x, int y);
char& operator[] (int i);
```

## 8.37 coDoText

```
coDoText(const char *n, int s);
coDoText(const char *n, int s, const char *d);
int getTextLength();
void getAddress(char **base);
```

## 8.38 coDoIntArr

```
coDoIntArr(const char *objName, int numDim, const int *dimArray,
          const int *initdata=NULL);
int getNumDimensions();
int getDimension(int i);
int getSize();
int *getAddress();
void getAddress(int **res);
```

## 8.39 coDoSet

```
coDoSet(const char *n, coDistributedObject * const *elem);
coDistributedObject * const *getAllElements(int *no);
coDistributedObject *getElement(int no);
```

## 8.40 coCoviseConfig

**This class makes the COVISE config-file accessible for the module.**

```
#include <config/CoviseConfig.h>

static const char *coCoviseConfig::getEntry(const char *entry);
static const char *coCoviseConfig::getEntry(const char *variable, const char *entry);
static int coCoviseConfig::getInt(const char *entry, int defaultValue, bool *exists=NULL);
static int coCoviseConfig::getInt(const char *variable, const char *entry, int defaultValue, bool
static long coCoviseConfig::getLong(const char *entry, long defaultValue, bool *exists=NULL);
static long coCoviseConfig::getLong(const char *variable, const char *entry, long defaultValue, bo
static float coCoviseConfig::getFloat(const char *entry, long defaultValue, float *exists=NULL);
static float coCoviseConfig::getFloat(const char *variable, const char *entry, float defaultValue,
static bool coCoviseConfig::isOn(const char *entry, bool defaultValue, float *exists=NULL);
```

```
static bool coCoviseConfig::isOn(const char *variable, const char *entry, bool defaultValue, bool
static char *coCoviseConfig::getScopeEntry(const char *scope, char *name);
static char **coCoviseConfig::getScopeEntries(const char *scope);
static char **coCoviseConfig::getScopeEntries(const char *scope, const char *name);
```

# Figures

1.1 Module and Dataflow network . . . . . . 7
1.2 Module execution scheme . . . . . . 7

3.1 Uniform Grid . . . . . . 34
3.2 Rectilinear Grid . . . . . . 35
3.3 Structured Grid . . . . . . 36
3.4 Unstructured Grid Base Element . . . . . . 38
3.5 Unstructured Grid format . . . . . . 39
3.6 Unstructured Grid example . . . . . . 39
3.7 Element Neighbor List . . . . . . 41
3.8 Lists for defining a line structure . . . . . . 47
3.9 Example for Lines . . . . . . 48
3.10 Polygon Example . . . . . . 49
3.11 Polygon Neighbor List . . . . . . 51
3.12 Single triangle strip . . . . . . 52
3.13 Triangle Strip example . . . . . . 53

7.1 OpenCOVER scene graph . . . . . . 82

# Index


∼coPointerButton
  opencover::coPointerButton, 111
2D Vector data, *see* coDoVec2
2D textures, *see* coDoTexture
3D Vector data, *see* coDoVec3

addPlugin
  opencover::coVRPluginSupport, 103
addedNode
  opencover::coVRPluginSupport, 102
Architecture suffix, 5
Arrays, *see* coDoIntArr

Basics, 5

CastShadow
  opencover::Isect, 113
Character data, *see* coDoText
CMake, 5
cmake
  template, 6
coIntersection
  opencover::coVRPluginSupport, 110
coPointerButton
  opencover::coPointerButton, 111
coVRMSController
  opencover::coPointerButton, 112
  opencover::coVRPluginSupport, 110
coVRPluginSupport
  opencover::coPointerButton, 112
coCoviseConfig, 27
  getEntry, 27
  getFloat, 28
  getInt, 27
  getLong, 28
  getScopeEntries, 28
  getScopeEntry, 28
  isOn, 28
coDistributedObject
  addAttribute, 32
  addAttributes, 32
  copyAllAttributes, 33
  getAllAttributes, 33
  getAttribute, 33
  getName, 30
  getType, 30
  incRefCount, 68
  isType, 30
  objectOk, 31
coDoFloat
  constructor, 42
  getAddress, 42
  getNumPoints, 42
  getPointValue, 42
coDoGeneralStructuredGrid
  getGridSize, 33
  getPointCoordinates, 33
coDoGeometry
  constructor, 45
  getColorAttributes, 46
  getColors, 46
  getGeometry, 46
  getNormalAttributes, 46
  getNormals, 46
  getTexture, 46
  getTextureAttributes, 46
  setColors, 46
  setNormals, 46
  setTexture, 46
coDoIntArr
  constructor, 54
  getAddress, 55
  getDimension, 55
  getNumDimensions, 54
  getSize, 55
coDoLines
  constructor, 48
  getAddresses, 48
  getNumLines, 49
  getNumPoints, 49
  getNumVertices, 49
coDoPixelImage
  constructor, 53
  getFormat, 53
  getHeight, 53
  getPixels, 53
  getPixelsize, 53
  getWidth, 53
coDoPoints
  constructor, 46
  getAddresses, 46
  getNumPoints, 47
  getPointValue, 47
coDoPolygons, 50
  constructor, 50
  get_neighbor_list, 51
  getAddresses, 50
  getNumLines, 50
  getNumPoints, 50
  getNumVertices, 50
coDoRectilinearGrid
  constructor, 35

getAddresses, 36
getGridSize, 36
getPointCoordinates, 36
coDoRGBA
constructor, 44
getAddress, 44
getFloatRGBA, 45
getIntRGBA, 45
getPointValue, 45
setFloatRGBA, 45
setIntRGBA, 45
coDoSet
constructor, 56
getAllElements, 56
getElement, 56
coDoStructuredGrid
constructor, 36
getAddresses, 37
getGridSize, 37
getPointCoordinates, 37
coDoTensor
constructor, 44
getAddress, 44
getNumPoints, 44
getPointValue, 44
getTensorType, 44
coDoText
constructor, 54
getAddress, 54
getTextLength, 54
coDoTexture
constructor, 53
getBorder, 53
getBuffer, 53
getComponents, 53
getCoordinates, 53
getHeight, 53
getLevel, 53
getNumCoordinates, 53
getVertices, 53
getWidth, 53
coDoTriangleStrips
constructor, 52
coDoUniformGrid
constructor, 34
getDelta, 34, 35
getGridSize, 34
getPointCoordinates, 34
coDoUnstructuredGrid
constructor, 38
getAddresses, 40
getGridSize, 40
getNeighborList, 41
getTypeList, 40
hasTypeList, 41
coDoVec2
constructor, 43
getAddresses, 43
getPointValue, 43
coDoVec3
constructor, 43
getAddresses, 43
getPointValue, 43
coInputPort
getCurrentObject, 14
setRequired, 14
Collision
opencover::Isect, 113
coModule, 9
addInputPort, 13
addOutputPort, 14
compute, 10
constructor, 9
paraCase, 26
paraEndCase, 27
paraEndSwitch, 26
param, 10
paraSwitch, 26
postInst, 10
selfExec, 68
sendError, 67
sendInfo, 67
sendWarning, 67
sockData, 10
start-up, 6
Configuration database, 27
Container, *see* coDoSet
coOutputPort
getObjName, 15
setCurrentObject, 15
setDependency, 14
coPort
setInfo, 15
coSimLib
getVerboseLevel, 61
serverMode, 61
setLocalHost, 60
setTargetHost, 60
setUsrArg, 61
setVerbose, 61
startupSim, 59
status, 60
coUifElem
getDesc, 13
getName, 13
print, 13
coUifPara
disable, 16

enable, 16
hide, 16
show, 15
COVER, *see* OpenCOVER
coVRPluginSupport
Access to the Buttons, 90
Access to the camera, 89
Access to the scene graph, 87
Append Buttons to the Pinboard, 91
Communicate with other Plugins, 90
Interactions, 90
Loading 3D Geometry, 89
currentTime
opencover::coVRPluginSupport, 103

Data flow, 6
Data Object, *see* coDistributedObject
Data Object Types, 29
Data Objects, 29, 42
debugLevel
opencover::coVRPluginSupport, 103
Distributed Object, 29, *see* coDistributedObject

Execution sequence, 6
Explicit flow control, 67

fasi
opencover::coVRPluginSupport, 110
fasi2
opencover::coVRPluginSupport, 110
frameDuration
opencover::coVRPluginSupport, 103
frameRealTime
opencover::coVRPluginSupport, 103
frameTime
opencover::coVRPluginSupport, 103

Geometry container, *see* coDoGeometry
Geometry data, 45
getActiveClippingPlane
opencover::coVRPluginSupport, 103
getBaseMat
opencover::coVRPluginSupport, 104
getClipPlane
opencover::coVRPluginSupport, 104
getCurrentCursor
opencover::coVRPluginSupport, 104
getIntersectedNode
opencover::coVRPluginSupport, 104
getIntersectedNodePath
opencover::coVRPluginSupport, 104
getIntersectionHitPointWorld
opencover::coVRPluginSupport, 104
getIntersectionHitPointWorldNormal
opencover::coVRPluginSupport, 104
getInvBaseMat
opencover::coVRPluginSupport, 104
getMenu
opencover::coVRPluginSupport, 104
getMenuGroup
opencover::coVRPluginSupport, 104
getMouseButton
opencover::coVRPluginSupport, 105
getMouseMat
opencover::coVRPluginSupport, 105
getNumClipPlanes
opencover::coVRPluginSupport, 105
getObjectsRoot
opencover::coVRPluginSupport, 105
getObjectsScale
opencover::coVRPluginSupport, 105
getObjectsXform
opencover::coVRPluginSupport, 105
getPlugin
opencover::coVRPluginSupport, 105
getPointer
opencover::coVRPluginSupport, 105
getPointerButton
opencover::coVRPluginSupport, 105
getPointerMat
opencover::coVRPluginSupport, 105
getRelativeButton
opencover::coVRPluginSupport, 105
getRelativeMat
opencover::coVRPluginSupport, 105
getScale
opencover::coVRPluginSupport, 106
getScene
opencover::coVRPluginSupport, 106
getSceneSize
opencover::coVRPluginSupport, 106
getSender
opencover::coVRPluginSupport, 106
getState
opencover::coPointerButton, 111
getToolBar
opencover::coVRPluginSupport, 106
getUpdateManager
opencover::coVRPluginSupport, 106
getViewerMat
opencover::coVRPluginSupport, 106
getWheel
opencover::coPointerButton, 111
getXformMat
opencover::coVRPluginSupport, 106
grabKeyboard
opencover::coVRPluginSupport, 106
grabViewer
opencover::coVRPluginSupport, 106

Image data, *see* coDoPixelImage
initUI
    opencover::coVRPluginSupport, 106
Integer arrays, *see* coDoIntArr
Intersection
    opencover::Isect, 113
IntersectionBits
    opencover::Isect, 112
isClippingOn
    opencover::coVRPluginSupport, 106
isHighQuality
    opencover::coVRPluginSupport, 107
isKeyboardGrabbed
    opencover::coVRPluginSupport, 107
isVRBconnected
    opencover::coVRPluginSupport, 107
isViewerGrabbed
    opencover::coVRPluginSupport, 107

Left
    opencover::Isect, 113
Libraries
    coApi, 6
    coAppl, 6
    coCore, 6
Line data, *see* coDoLines

Makefile, 5
module states, 6
mopla
    opencover::coVRPluginSupport, 110

name
    opencover::coPointerButton, 111
NoMirror
    opencover::Isect, 113
notPressed
    opencover::coPointerButton, 111
notify
    opencover::coVRPluginSupport, 107

oldState
    opencover::coPointerButton, 111
OpenCOVER
    opencover::coVRPluginSupport, 110
OpenCOVER
    Controlling a COVISE module, 96
    Dynamic Libraries, 81
    introduction, 81
    OpenCOVER as a COVISE module, 82
    OpenSceneGraph, 81
opencover::Isect
    CastShadow, 113
    Collision, 113
    Intersection, 113
    Left, 113
    NoMirror, 113
    OsgEarthSecondary, 113
    Pick, 113
    ReceiveShadow, 113
    Right, 113
    Touch, 113
    Update, 113
    Visible, 113
    Walk, 113
opencover::Isect, 112
    IntersectionBits, 112
opencover::coPointerButton, 110
    ~coPointerButton, 111
    coPointerButton, 111
    coVRMSController, 112
    coVRPluginSupport, 112
    getState, 111
    getWheel, 111
    name, 111
    notPressed, 111
    oldState, 111
    setWheel, 112
    wasPressed, 112
    wasReleased, 112
opencover::coVRPluginSupport, 101
    addPlugin, 103
    addedNode, 102
    coIntersection, 110
    coVRMSController, 110
    currentTime, 103
    debugLevel, 103
    fasi, 110
    fasi2, 110
    frameDuration, 103
    frameRealTime, 103
    frameTime, 103
    getActiveClippingPlane, 103
    getBaseMat, 104
    getClipPlane, 104
    getCurrentCursor, 104
    getIntersectedNode, 104
    getIntersectedNodePath, 104
    getIntersectionHitPointWorld, 104
    getIntersectionHitPointWorldNormal, 104
    getInvBaseMat, 104
    getMenu, 104
    getMenuGroup, 104
    getMouseButton, 105
    getMouseMat, 105
    getNumClipPlanes, 105
    getObjectsRoot, 105
    getObjectsScale, 105
    getObjectsXform, 105

getPlugin, 105
getPointer, 105
getPointerButton, 105
getPointerMat, 105
getRelativeButton, 105
getRelativeMat, 105
getScale, 106
getScene, 106
getSceneSize, 106
getSender, 106
getToolBar, 106
getUpdateManager, 106
getViewerMat, 106
getXformMat, 106
grabKeyboard, 106
grabViewer, 106
initUI, 106
isClippingOn, 106
isHighQuality, 107
isKeyboardGrabbed, 107
isVRBconnected, 107
isViewerGrabbed, 107
mopla, 110
notify, 107
OpenCOVER, 110
preparePluginUnload, 107
protectScenegraph, 107
releaseKeyboard, 107
releaseViewer, 107
removeNode, 107
removePlugin, 108
sendGrMessage, 108
sendMessage, 108
sendVrbMessage, 108
setActiveClippingPlane, 108
setCurrentCursor, 109
setCursorVisible, 109
setFrameTime, 109
setNodesIsectable, 109
setRenderStrategy, 109
setScale, 109
setToolBar, 109
setXformMat, 109
unwatchFileDescriptor, 110
updateInteractorTransform, 110
watchFileDescriptor, 110

OsgEarthSecondary
opencover::Isect, 113

Packed RGBA data, *see* coDoRGBA
Parameter
Boolean, 16
Choice, 23
Common functions, 15
File Browser, 22
Scalar, 17
Slider, 19
String, 22
Vector, 21
Parameter switching, 25
Parameters
common functions, 12
Pick
opencover::Isect, 113
Pixel image data, *see* coDoPixelImage
Plugin Interface
coVRPlugin(), 84
addNode, 85
addObject(), 84
constructor, 83
coviseError, 85
coVRPlugin(), 83
destroy(), 84
destructor, 84
init(), 83
Introduction, 81
key, 86
message, 87
newInteractor, 85
postFrame, 86
preFrame, 86
preSwapBuffers, 86
removeNode, 86
removeObject, 85
setTimestep, 86
Point data, *see* coDoPoints
Polygon data, *see* coDoPolygons
Ports
common functions, 12
preparePluginUnload
opencover::coVRPluginSupport, 107
Prerequisites, 5
protectScenegraph
opencover::coVRPluginSupport, 107

ReceiveShadow
opencover::Isect, 113
Rectilinear Grid, *see* coDoRectilinearGrid
releaseKeyboard
opencover::coVRPluginSupport, 107
releaseViewer
opencover::coVRPluginSupport, 107
removeNode
opencover::coVRPluginSupport, 107
removePlugin
opencover::coVRPluginSupport, 108
RGBA data, *see* coDoRGBA
Right

opencover::Isect, 113

Scalar data, *see* coDoFloat
Self-execution, 68
sendGrMessage
opencover::coVRPluginSupport, 108
sendMessage
opencover::coVRPluginSupport, 108
sendVrbMessage
opencover::coVRPluginSupport, 108
Set, *see* coDoSet
setActiveClippingPlane
opencover::coVRPluginSupport, 108
setCurrentCursor
opencover::coVRPluginSupport, 109
setCursorVisible
opencover::coVRPluginSupport, 109
setFrameTime
opencover::coVRPluginSupport, 109
setNodesIsectable
opencover::coVRPluginSupport, 109
setRenderStrategy
opencover::coVRPluginSupport, 109
setScale
opencover::coVRPluginSupport, 109
setToolBar
opencover::coVRPluginSupport, 109
setWheel
opencover::coPointerButton, 112
setXformMat
opencover::coVRPluginSupport, 109
SimClient
coGetParaBool, 63
coGetParaChoice, 63
coGetParaFile, 63
coGetParaFloat, 62
coGetParaInt, 62
coGetParaSlider, 62
coGetParaText, 63
COGPBO, 63
COGPCH, 63
COGPFI, 63
COGPFL, 62
COGPIN, 62
COGPSL, 62
COGPTX, 63
coInitConnect, 62
CONOCO, 62
coNotConnected, 62
COPACM, 65
COPAIN, 64
COPANO, 65
COPAPO, 64
coParallelCellMap, 65
coParallelInit, 64
coParallelNode, 65
coParallelPort, 64
coParallelVertexMap, 65
COPAVM, 65
coSend1Data, 63
coSend3Data, 64
COSU1D, 63
COSU3D, 64
COVINI, 62
Simulation
Check connection, 62
Client Commands, 61
Creating data objects, 63
data on parallel machines, 64
Language bindings, 57
Server Commands, 58
start-up, 58
start-up parameters, 60
Starting the Server mode, 61
Status request, 60
Simulation Library, 57
Singleton
coCoviseConfig, 27
String data, *see* coDoText
Structured Grid, 33, *see* coDoStructuredGrid
Abstract base class, *see* coDoGeneralStructuredGrid

Tensor data, *see* coDoTensor
Text data, *see* coDoText
Texture data, *see* coDoTexture
Touch
opencover::Isect, 113
Triangle strips, *see* coDoTriangleStrips

Uniform Grid, *see* coDoUniformGrid
Unstructured Grid, *see* coDoUnstructuredGrid
unwatchFileDescriptor
opencover::coVRPluginSupport, 110
Update
opencover::Isect, 113
updateInteractorTransform
opencover::coVRPluginSupport, 110

Visible
opencover::Isect, 113

Walk
opencover::Isect, 113
wasPressed
opencover::coPointerButton, 112
wasReleased
opencover::coPointerButton, 112
watchFileDescriptor
opencover::coVRPluginSupport, 110