IFrIT

User Guide

Reference Guide

by

Nick Gnedin and IFrIT

IFrIT Version 3.0

Ifrits in common mythology are jinn spirits that embody fire. They consider themselves superior to all races because they supposedly "came first," and they resent deeply that humans have found magical ways to take control over them.
Even when tasked, they show an ironic and malicious attitude, tending to subvert their masters' orders every time they can.

(Wikipedia.org)

Table of Contents

1 User Guide....1
- 1.1 Overview....1
  - 1.1.1 What can IFrIT do?....1
  - 1.1.2 General Structure of IFrIT....2
  - 1.1.3 Components of the IFrIT core....3
- 1.2 Controlling IFrIT....4
  - 1.2.1 Overview....4
  - 1.2.2 Mouse and Keyboard Controls....4
  - 1.2.3 Environment Variables....5
  - 1.2.4 Command–line Options....6
  - 1.2.5 State File....6
- 1.3 File Formats....7
  - 1.3.1 Overview....7
  - 1.3.2 Uniform Scalars Data....9
  - 1.3.3 Uniform Vectors Data....10
  - 1.3.4 Uniform Tensors Data....10
  - 1.3.5 Basic Particles Data....10
- 1.4 IFrIT Palettes....11
  - 1.4.1 Overview....11
- 1.5 Animation Support....12
  - 1.5.1 Overview....12
  - 1.5.2 Animatable Files....13
- 1.6 Animation and Control Scripts....14
  - 1.6.1 Overview....14
  - 1.6.2 Expressions....14
  - 1.6.3 Common Statements for Animator and Control Scripts....15
  - 1.6.4 Specific Control Script Statements....17
  - 1.6.5 Specific Animator Script Statements....19
  - 1.6.6 Pre–defined Animator Script Variables....20

2 Shell Reference....23
- 2.1 Command–line Shell Reference....23
  - 2.1.1 Command–line Shell....23
- 2.2 GUI Shell Reference....24
  - 2.2.1 Graphical User Interface (GUI) Shell....24
  - 2.2.2 Animation Script Debugger....25
  - 2.2.3 Array Calculator....25
  - 2.2.4 Command Line....26
  - 2.2.5 Data Explorer....26
  - 2.2.6 Palette Editor....26
  - 2.2.7 File Set Explorer....27
  - 2.2.8 Image Composer....27
  - 2.2.9 Picker Window....28
  - 2.2.10 Parallel Controller....28
  - 2.2.11 Event Recorder....28

3 Object Reference....29
- 3.1 Overview....29
  - 3.1.1 Components of the IFrIT core....29
  - 3.1.2 ControlModule Requests....30

Table of Contents

3 Object Reference
3.2 Available objects....30
3.3 Animator object....31
3.4 Camera object....34
3.5 ColorBars object....36
3.6 ControlModule object....36
3.7 CrossSection object....37
3.8 Data−BasicParticles object....39
3.9 Data−UniformScalars object....39
3.10 Data−UniformTensors object....40
3.11 Data−UniformVectors object....41
3.12 DataReader object....42
3.13 ImageComposer object....44
3.14 Marker object....47
3.15 ParticleGroup object....49
3.16 Particles object....52
3.17 Picker object....54
3.18 Surface object....55
3.19 Tensor object....58
3.20 Vector object....59
3.21 ViewModule object....62
3.22 Volume object....68

A Appendices....70
A.1 Codes For Writing IFrIT Data Files....70
A.1.1 Code Examples....70
A.1.2 Fortran....70
A.1.3 C....72
A.1.4 IDL....76
A.2 License Agreement....78
A.2.1 Overview....78
A.2.2 GNU General Public License....79

1 User Guide

1.1 Overview

1.1.1 What can IFrIT do?

IFrIT can visualize four different classes of data:

- **Scalar Field** data: several scalar variables in 3D space.
- **Vector Field** data: a 3D field of vectors.
- **Tensor Field** data: a symmetric 3x3 tensor in 3D space.
- **Particle Set** data: a set of particles (points) with several optional attributes (numbers that distinguish particles from each other) per particle.

For the **Scalar Field** data the following visualization objects are available:

- A two−dimensional surface (either an isosurface of a given variable, or a fixed geometric surface: a plane or a sphere). Several instances (copies) of each surface may co−exist (for example, isosurfaces at different levels of the same variable). Surfaces can be colored on the outside or inside by a value of another scalar variable, translated into color through a palette.
- An orthogonal cross section of a data cube, again several of them can be shown at a time.
- Volume rendering of one scalar variable.

The **Vector Field** data can be represented either as

- a "vector glyph" − a line that starts at each mesh point, points in the direction of the vector, and has a length proportional to the vector magnitude (sorry, no arrows in 3D), or
- a set of streamlines − lines along which the fluid would flow if the Vector Field is assumed to be a velocity field of some imaginary fluid. Streamlines can be colored by Vector Field properties (like magnitude, vorticity, etc), or by scalar variables, if the Scalar Field file of the same dimensions is loaded too. Streamlines also can be represented as tubes, with the tube diameter inversely proportional to the vector magnitude, or by ribbons with two neighborning streamlines being the ribbon boundaries.

The **Tensor Field** data are hard to visualize. At the moment, the only supported visualization mode is the "tensor glyphs" − ellipsiods with orientation and dimensions proportional to three tensor eigenvalues, placed at some of the vertices of the uniform mesh.

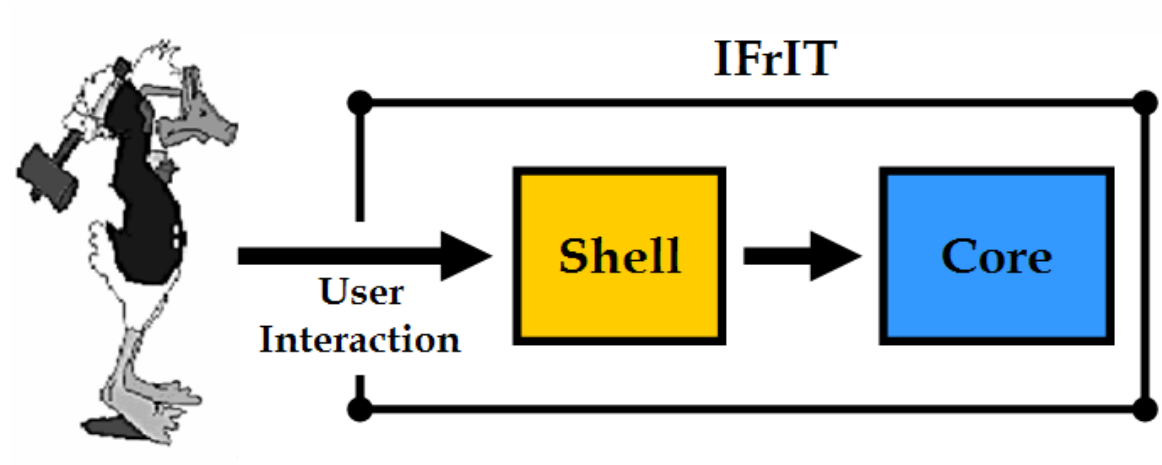
The **Particle Set** data can be split into individual groups, and particles in each group can have various representations (dots, spheres, clouds of dots, etc), can be colored by the value of one of the attributes, and can be sized with an arbittary sizing function by the value of another attribute. Particles belonging to one group can be connected by a line − this is useful for, say, plotting trajectories.

Different modes of visualization are coexistent, they are activated/de−activated independently of each other. Several visualization windows can exist at the same time, each one having a full set of visualization objects. Some visualization windows can share the data between them, while other windows can be fully independent. Images from several visualization windows can be combined into one image file on the disk, tiling some windows together, and inserting reduced versions of some windows into larger other windows.

A large array of nifty features is also available, including highly advanced animation capabilities, a complex set of lights, markers to label various points in space, a capability to "pick" a point in the scene and retrive information about the data at this location, two scripting languages, etc.

1.1.2 General Structure of IFrIT

IFrIT consists of a set of components called **Objects**. Each object is designed to perform a certain function: just like in a computer various components − a processor, memory, a video card, etc − are designed to perform certain tasks. Various objects have diverse relations to each other: some objects are independent, while other objects may belong to different objects. You can think of the general structure of IFrIT as a directory hierarchy: objects may have "sub−objects" in them, these, in turn, may have "sub−sub−objects", etc. The diagram below shows the main structure of IFrIT:



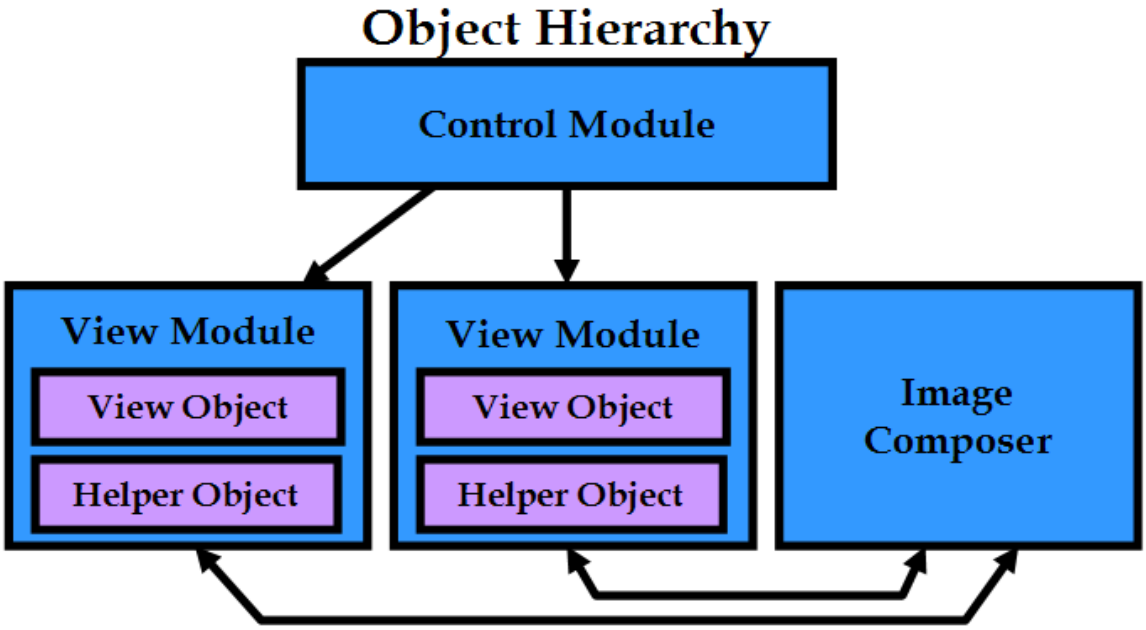
General structure of IFrIT

All the objects together form a **core** of IFrIT (blue parts in the chart − like a motherboard in a computer, or the full file system in a directory analogy). The core can be manipulated by a user via a **Shell** (yellow block in the chart). A shell acts like a keyboard in a computer analogy. Currently, IFrIT supports several shells. The simplest one is a **Command−line Shell**, that uses a command prompt to take typed commands from the user and forward them to the core. Because typing commands is slow, this shell is only useful for driving IFrIT over a remote slow connection, when graphical user interface does not work well. Other shells use GUI (Graphical User Interface) to provide a user with a fast way of forwarding requests to the core. Because of licensing issues, several GUI shells are available.

Each invocation of the executable uses a specific shell, but more than one shell can be compiled into the executable, so that different shells can be used without re−compiling IFrIT. A specific shell can be invoked either by using Command−line Options, or if it specified as a default one. If the default shell is not specified, and none is requested in the command line, and more than one shell is compiled into the code, IFrIT will ask a user to choose a shell at start−up. Only shells that have been compiled into IFrIT during the installation will be accessible, of course.

1.1.3 Components of the IFrIT core

The core of IFrIT consists of objects. Depending on their use, objects are divided into two categories: **modules** and **View Objects**. A hierarchy of IFrIT objects is shown in the figure below.



IFrIT object hierarchy

Modules form the highest level of object hierarchy. A **Control Module** object plays a role of the root of the hierarchy, its main function is to provide coordination between various **View Module**s. A **View Module** appears as an IFrIT visualization window on the screen. All **View Objects** belong to one of the **View Module**s, with the single exception of **ImageComposer** module, which is responsible for composing a snapshot or an animation image from several **View Module**s.

Each **View Module** is an independent part of IFrIT that has its own visualization window on the computer screen, its own set of other objects and, often, its own data. Some of **View Module**s may share the data with another **View Module** – in that case the owner of the data is called a "parent" **View Module**, and sharing **View Module**s are called "clones". Irrespectively of whether a given **View Module** is a clone or a parent, it has its own set of **View Objects**. **View Module**s are directly responsible for a general setup of the visualization scene, such as lighting, camera properties, bounding box, clipping plane, and various accessories (such as rulers, labels, measuring boxes, etc). They "outsource" operations on various components of the visualization scene and manipulations on the data to their own View Objects.

Some **View Objects** are responsible for maintaining various parts of the visualization scene. For example the **Surface** object represents a two–dimensional surface within the visualization scene that samples the three–dimensional scalar data (either as an isosurface of a particular scalar variable, or a specified geometric shape like a sphere or a plane), while the **Particles** object represents a set of particles (points).

Other **View Objects** (also called **Helper Objects**) perform various functions that are not directly represented in the visualization scene. For example, the **Data Reader** object is responsible for loading data files into IFrIT, while the **Animator** object creates animations of your visualization scene in a diverse variety of ways.

Object are manipulated by special requests, which invoke specialized parts of each object called **properties**. You can think of a property as of a special variable: as soon as you assign a new value to the variable, something happens. For example, if you assign a value 1 (which is equivalent to `true`) to the property **ViewModule:BoundingBox**, the bounding box will appear in the visualization scene. Assign 0 (`false`) to the same property, and the bounding box disappears. In the **Command–line Shell** these requests are typed explicitly in the command prompt, but in GUI shells commands are issued by GUI widgets, and a user does not need to know the specific form of every command.

1.2 Controlling IFrIT

1.2.1 Overview

The main method of controlling IFrIT is by using a shell to manipulate various objects. In addition, IFrIT can be controlled by using

- Mouse clicks and motions in the visualization window,
- Environment variables,
- Command–line options, and
- External files.

1.2.2 Mouse and Keyboard Controls

IFrIT reaction to mouse clicks and movements depends on which of three major mouse interaction modes it is in.

- **Display mode** is the default mouse interaction mode. In this mode the following binding of mouse and keyboard are available:

  Left button: rotates the camera around its focal point by moving the mouse.
  [Ctrl] key + **Left button**: rotates the camera around the Z–axis (axis perpendicular to the screen).
  Middle button: pans (moves in space) the camera.
  Right button: zooms the scene in and out.
  3 key: toggles into and out of stereo mode.
  P key: performs a pick operation (i.e. selects a point in the scene the mouse cursor is pointing to, and retrives information about the data at this location).
  R key: resets the camera view along the current view direction.
  S key: modifies the representation of all actors so that they are surfaces.
  W key: modifies the representation of all actors so that they are wireframe.

- **Fly–by mode** emulates flying an airplane through the current visualization scene.

  Left button: moves forward.
  Right button: moves backward.
  [Shift] key: accelerator in mouse and key modes.
  [Ctrl]+[Shift] keys: causes sidestep instead of steering.

+/– keys: increases/deacreases normal speed.

- **Measuring box mode** becomes active when a measuring box is shown in the current visualization window. The box can be used to measure sizes of different features in the scene.

  Left button: rotates the measuring box around its origin.
  Middle button: translates the measuring box in/out of screen.
  Right button: translates the measuring box along scene axes.
  A/Z keys: scales the measuring box down/up.
  S/X keys: adjusts box opacity.
  C key: shifts the camera focal point to the box center.

- **Keyboard Interactor mode** uses keyboard to emulate all mouse interactions. The keyboard mode is slow, but precise and reproducible. In this mode the following keys are available:

  Left/Right (or **H/L**) keys: rotates the camera horizontaly around its focal point.
  Down/Up (or **J/K**) keys: rotates the camera vertically around its focal point.
  +/– keys: zoom the scene in/out.
  A/Z keys: slow down/speed up the interaction.
  Other key binding are the same as in the **Display** mode.

- **Common to all modes**:

  F key: switches in and out of the full screen mode (will not work in all shells).
  U key: dumps the current view of the scene into an image file on disk.

1.2.3 Environment Variables

IFrIT understands the following environment variables (all in capitals):

- **`IFRIT_DIR`**: The main IFrIT directory where you keep the state file(s) **ifrit.ini** and possibly other configuration files. If this variable is not set, IFrIT will check the environment variable HOME. If it exists, it will make $HOME/.ifrit the main directory (and will create that directory if it does not exist). If the environment variable HOME is not defined, IFrIT will try to create a main directory at /.ifrit. If everything else fails, IFrIT assumes that current directory is the main directory.
- **`IFRIT_SCRIPT_DIR`**: The directory where IFrIT keeps animation and control scripts. If not set, IFrIT will put scripts into its main directory.
- **`IFRIT_PALETTE_DIR`**: The directory where IFrIT keeps custom palettes. If not set, IFrIT will put palettes into its main directory.
- **`IFRIT_IMAGE_DIR`**: The directory where IFrIT keeps image and animation files. If not set, IFrIT will put image files into the current directory.
- **`IFRIT_DATA_DIR`**: The default directory for the data files. If not set, IFrIT will start searching for the data files in the current directory.
- **`IFRIT_SCALAR_FIELD_DATA_DIR`**: The default directory for the **Scalar Field data** files. If not set, IFrIT will start searching for the Scalar Field data in the default data directory.
- **`IFRIT_VECTOR_FIELD_DATA_DIR`**: The default directory for the **Vector Field data** files. If not set, IFrIT will start searching for the Vector Field data in the default data directory.

- **`IFRIT_TENSOR_FIELD_DATA_DIR`**: The default directory for the **Tensor Field data** files. If not set, IFrIT will start searching for the Tensor Field data in the default data directory.
- **`IFRIT_PARTICLE_SET_DATA_DIR`**: The default directory for the **Particle Set data** files. If not set, IFrIT will start searching for the Particle Set data in the default data directory.

1.2.4 Command–line Options

Several options can be specified in the command line when invoking IFrIT. Options begin with the dash (–) symbol, but the first option is special: it invokes IFrIT with a specific shell, as follows:

```
ifrit -<shell-specification>
```

where `<shell-specification>` is one of the following:

- **`cl`** for a command–line shell,
- **`qt`** for a GUI shell based on the Qt Graphical User Interface toolkit, and
- **`fx`** for a GUI shell based on the FOX Graphical User Interface toolkit.

Only shells that have been compiled into IFrIT during the installation will be accessible. If you do not specify the shell in the command line, a default shell will be used. The default shell is the first compiled–in shell from the folowing order: **qt**, **fx**, **cl**.

Different shells support different options, so for the full list of available options you need to refer to the documentation on the specific shell. However, all shells support the basic subset of command–line options:

- **`-h`**: shows the list of available command–line options, including those specific to a particular shell.
- **`-np`** *number*: sets the number of processors to *number* for parallel execution.
- **`-i`** *filename*: loads the full internal state from the previously saved state file *filename*.

If the last entry on the command line does not form any option, it is taken to be the name of directory; IFrIT will use this directory as his default DATA directory, overwriting the environment variable.

1.2.5 State File

IFrIT can remember its exact state and save it into the **ifrit.ini** file, which is placed into main IFrIT directory (see Environment Variables). This file is not intended to be modified by the user. When IFrIT starts, it looks for the **ifrit.ini** file in its main directory. If the file is found, IFrIT will read its internal configuration from it. Thus, if you spent a long time changing various settings and adjusting IFrIT to your needs, all you need to do is to save the state, and the next time you start IFrIT, all your settings will be restored automatically, and your hard work will not be wasted.

You can keep several of state files and use the appropriate one by specifying its name after **–i** option on the command line.

1.3 File Formats

1.3.1 Overview

IFrIT can visualize four different **classes** of data:

- **Scalar Field** data: several scalar variables in 3D space.
- **Vector Field** data: a 3D field of vectors.
- **Tensor Field** data: a symmetric 3x3 tensor in 3D space.
- **Particle Set** data: a set of particles (points) with several optional attributes (numbers that distinguish particles from each other) per particle.

Each class of data may contain more than one specific **type** of data. We use the word "type" here in the same sense as it is used to describe different data types in computer language (integer, boolean, real, etc). Thus, each data **type** is a unique representation of a specific layout of data in space, and is associated with a specific format of a data file. For example, **UniformScalars** data type describes several scalar variables on a uniform rectangular mesh in space, loaded with a specific file format. What data types available for each class depends on what extensions of IFrIT are included in your installation.

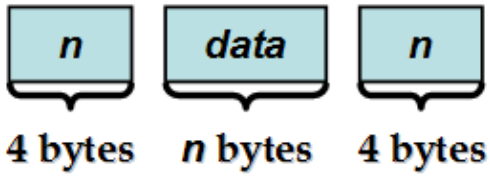
Each data **type** has one–to–one correspondence with a data object. The corresponding data object is named "Data–*DataType*", where *DataType* is the name of the data type. For example, the data object that corresponds to the **UniformScalars** data type is named **Data–UniformScalars**.

The standard edition of IFrIT includes four basic data types:

- **Uniform Scalars** data file: uniformly spaced 3–dimensional mesh of data values with several scalar variables per file.
- **Uniform Vectors** data file: a 3D Vector Field specified on a uniformly spaced mesh.
- **Uniform Tensors** data file: a symmetric 3x3 Tensor Field specified on a uniformly spaced mesh.
- **Basic Particles** data file: three positions and several optional attributes for a set of particles (or points).

Binary Files

IFrIT can load both the plain text (ASCII) and binary data files. The binary files use **Fortran unformatted** file formats – these files can be created in many programming languages, including Fortran, C, and IDL (code examples are given in Appendix A). Fortran writes binary data into a file in **records**. Each record contains a 4–byte header, a body of the record, and a 4–byte footer, as shown in the image below:



Structure of a Fortran record

The header and the footer are identical, and each contains a 4–byte integer number with the number of bytes in the body of the record. The body can contain any data. Because the length of the header and the footer are 4 bytes, one record cannot contain more bytes that can be described by a 4–byte signed integer number (2147483647). It is important that the header and the footer of the record be correct (i.e. contain the number of

bytes in the body). IFrIT uses the header and the footer to verify the integrity of the data file and to deduce the endiness of the data.

File Sets

IFrIT can load several data files of different formats simultaneously as a "set". For example, if you are visualizing the Uniform Scalars data and the Basic Particles data from a series of outputs from the same simulation, you can load files in pairs (or triplets if you add a Uniform Vectors, etc). Only Animatable Files can be loaded as sets. For example, if you load a Basic Particles file named part_1234.bin, and then load a Uniform Scalars file named scal_1234.bin (the same 4–digit record label), IFrIT will recognize these files as a set. The Basic Particles file becomes a **set leader**, and you will be able to load files in sets: if you load a new leader (another Basic Particles file, say, part_5678.bin), then the corresponding Uniform Scalars file (scal_5678.bin) will be loaded automatically. Sets can, of course, include data files of all 4 formats.

If one of the files in a set does not exist, or you load an individual file rather than a set, the set will be broken and IFrIT will treat files as unrelated.

Subdirectory `docs` of IFrIT source distribution contains three files: `writeIFRIT.f`, `writeIFRIT.c`, and `writeIFRIT.pro` that contain Fortran, C, and IDL code respectively for writing all four types of IFrIT data files. These files are also listed in Appendix A.

Cell vs Point Data

In order to understand the placement of the data in the scene, you need to know the difference between the cell and point data. Data can be specified on a uniform mesh in two distinct ways. If every cell of a mesh is represented as a cube, the data can be specified either at the center of a cube (**cell data**), or at the vertices of a cube (**point data**). Most of VTK classes can only handle point data, while most simulation codes place the data at cell centers. To handle this inconsistency, IFrIT allows to specify the placement of the data and to convert cell data into point data using CellToPointMode property of the **Data Reader** object.

If the data in the data file is the point data, then nothing needs to be done and IFrIT can use it directly. In that case, though, you have to make sure that the data descriptions in IFrIT and in the file are consistent. For example, if you specify periodic boundary conditions, the data values on the opposite sides of the data cube should be identical. If they are not, then extending data periodically beyond the bounding box will create visualization artifacts.

If your file contains the cell data, IFrIT can transform it into the point data in several ways: it can obtain a value at a point by taking 8 cell neighbors and computing an average, a median, a minimum, or a maximum value. All these methods cause the loss of information and effectively smooth the input data. Alternatively, IFrIT can project the cell value to a point (**CellToPointMode**=5). In that case the data become displaced by half of the cell size. This is acceptable if you work with scalar mesh data only, or if you have scalar, vector and tensor data on the same mesh. However, if you use non–commensurate mesh sizes for scalar, vector and tensor data, or if you use particles, then various data sets will be displaced relative to each other – which can bias the relative relation of different objects.

The most accurate method is called the "Fourier shift". It uses Fast Fourier Transform to project the cell data onto points. This method is lossless, and preserves location of the data, but it can be very slow for large mesh dimensions which are not powers of two. It also uses additional storage.

1.3.2 Uniform Scalars Data

IFrIT Uniform Scalars data file contains uniformly spaced 3–dimensional mesh of data values. Both, plain text (ASCII) and binary (Fortran–type binary of any endiness) files are accepted.

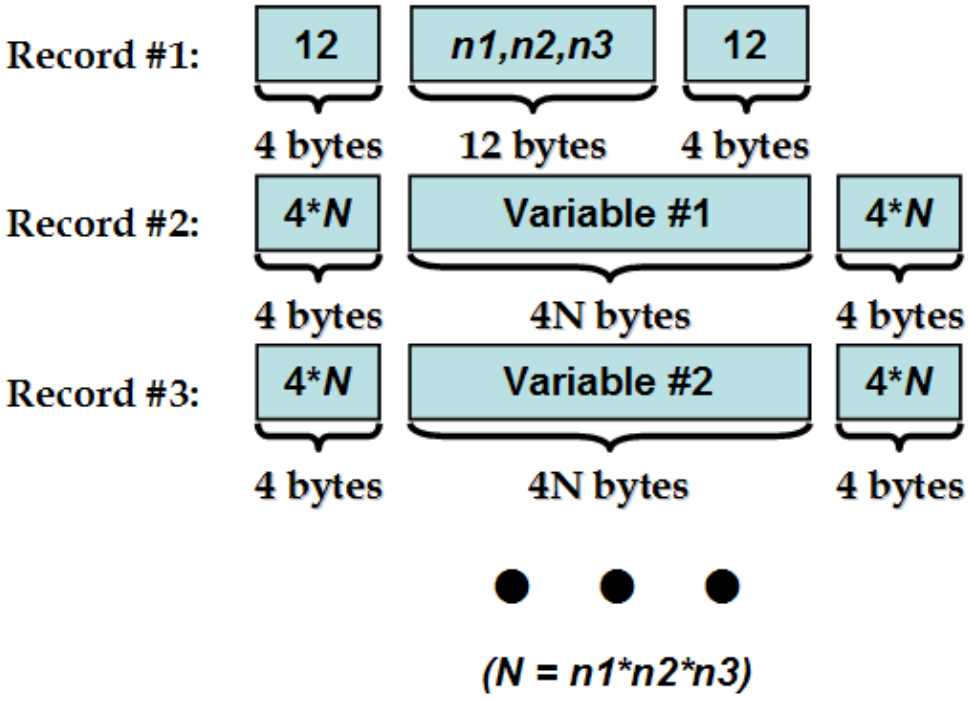
Plain text file. This file should have an extension "txt" (as in "myfile.txt", lower–case or upper–case does not matter) and should contain in its first line three integer numbers: the sizes of the mesh in X, Y, and Z directions. These dimensions do not have to be the same. Each line after the first one should contain up to 10 floating point numbers as values for the physical variables at each cell of the mesh. The first dimension changes the fastest. For example, the following file:

```
12 33 55
1.0456 4.56768 2.45e-30
0.9866 5.45890 3.07e-20
  ...
```

(12*33*55+1 lines altogether)

defines a 12 by 33 by 55 data mesh with three physical variables.

Binary file. This file should have an extension "bin" (as in "myfile.bin") and should be a **Fortran unformatted** binary file. The data in the file must be in **single precision** (Fortran `real*4`, C `float`). The file should contain at least 2 records. The first record should contain 12 bytes of data as 3 integer numbers: mesh sizes in three dimensions (`n1`, `n2`, and `n3`). The remaining records should contain `n1*n2*n3` floating point numbers each: one scalar variable per record.



Structure of the IFrIT Uniform Scalars data binary file

Examples of the code that can write such files is given in Appendix A.

1.3.3 Uniform Vectors Data

IFrIT Uniform Vectors file format is identical to the Uniform Scalars data file format with three vector components stored as three scalar variables.

1.3.4 Uniform Tensors Data

IFrIT (symmetric) Uniform Tensors data file format is identical to the Uniform Scalars data file format with 6 tensor components stored as 6 scalar variables in the following order: 11, 12, 13, 22, 23, 33.

1.3.5 Basic Particles Data

IFrIT Basic Particles data file contains three positions and, optionally, several attributes for a set of particles. Both, plain text (ASCII) and binary (Fortran−type binary of any endiness) files are accepted.

Plain ASCII file. This file should have an extension "txt" (as in "myfile.txt", lower−case or upper−case does not matter) and should contain in its first line one integer number: the total number of particles in the file. The second line should contain 6 numbers that determine how the particle positions relate to the bounding box. The first three numbers are X, Y, and Z coordinates of the lower−left−back corner of the bounding box in the units in which particle coordinates are given. The last three numbers are X, Y, and Z coordinates of the upper−right−front corner. For example, if you ran a simulation with a cubic box of 5 meters in size, and your particle positions are given in meters, and the left−lower−back corner of your box has coordinates of (0,0,0) meters, then the second line should be: `0.0 0.0 0.0 5.0 5.0 5.0 X-label Y-label Z-label`. Optionally, three strings are allowed at the end of the line to use as axis labels if the bounding box is displayed as 3−dimensional axes. This is useful for making 3D scatter plots.

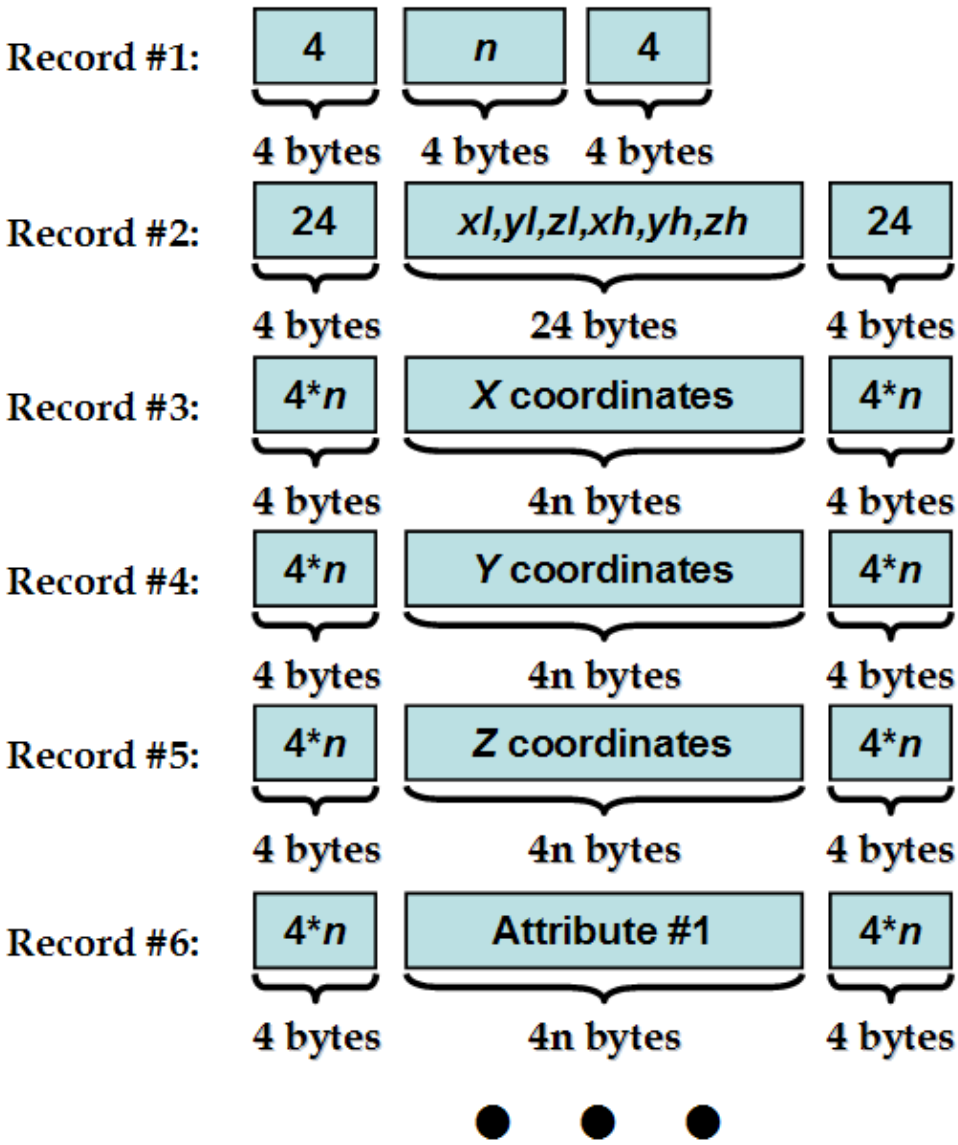
Each line after the second one should contain three floating point numbers as values for the three coordinates for each particle and, optionally, up to 10 more numbers as values of attributes. The attributes can be used to distinguish particles in a set. For example, the following file:

```
120
0.0 0.0 0.0 5.0 5.0 5.0
1.0456 4.56768 3.05678 2.45e-30 1.11e+10 0.555
0.9866 5.45890 -2.0568 3.07e-20 2.44e+11 -0.34
  ...
```

(120+2 lines altogether)

defines a set of 120 particles with three attribute fields defined. Notice that the second particle is located outside the bounding box − there is nothing wrong with that.

Binary file. This file should have an extension "bin" (as in "myfile.bin") and should be a **Fortran unformatted** binary file. The file should contain at least 5 records. The first record should contain 4 bytes of data as 1 integer number: the number of particles `n`. The second record should contain 24 bytes as 6 floating point numbers for 6 values of the bounding box. Records from 3 to 5 contain `n` single or double precision floating point numbers (4*`n` or 8*`n` bytes) each, which are x, y, and z coordinates of particles (i.e. all x coordinates are stored in record 3, all y coordinates are stored in record 4, etc). Optional remaining records contain `n` single precision floating point numbers with particle attributes (scalar values that characterize individual particles).



Structure of the IFrIT Basic Particles data binary file

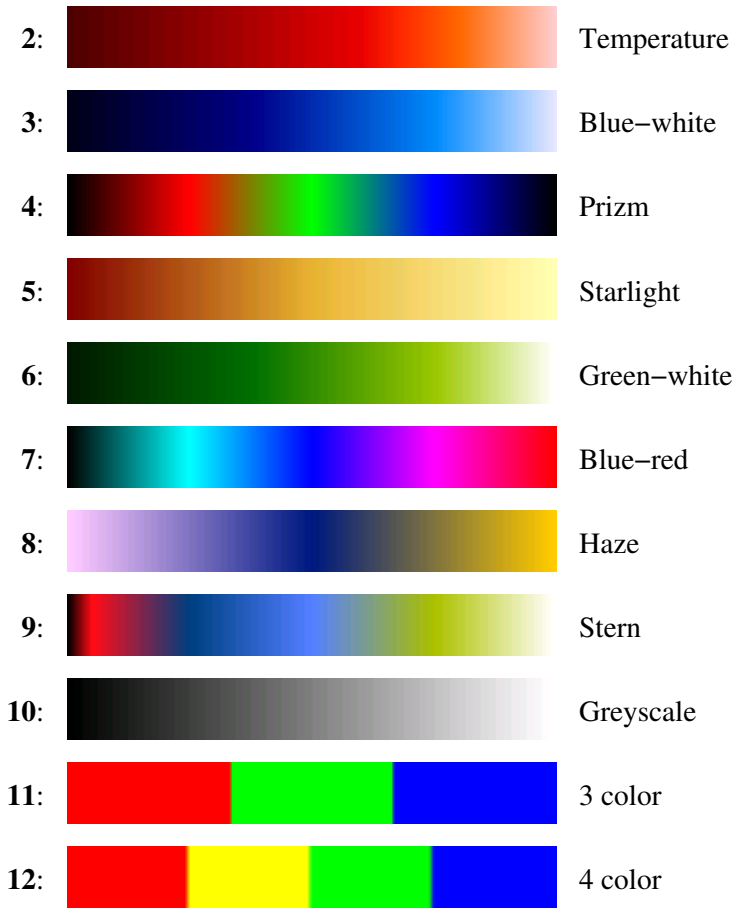
Examples of code that can write such files is given in Appendix A.

1.4 IFrIT Palettes

1.4.1 Overview

IFrIT includes a set of pre−defined *palettes* (or *color−maps*), i.e. a one−dimensional sequences of colors that are used to map scalar values to a color of a point in the scene. Palettes identified by their numbers:

1: Rainbow



Each palette can be used in its original form or "reversed", i.e. with colors changing from right to left. In addition, new palettes can be created and loaded in GUI shells.

Some of **View Objects** also use a special palette called "brightness", which includes shades of different intensity for the current object color. For example, if you the color of the object is red, then using the brightness palette will color the object with shades of red of varied brightness. This palette can be useful, for example, for coloring particles with varied intensity of yellow color, to represent stars of different magnitudes.

1.5 Animation Support

1.5.1 Overview

In the Animation mode IFrIT works automatically to generate an animation of your scene. You cannot control IFrIT interactively in this mode and you cannot change the scene – although IFrIT can change the scene for you automatically. By default, the output of an animation is a series of image files that show consequitive snapshots of the visualization scene. Animation images have **ifrit–an** as the root of their names and they are distinguished by a 4–digit sequential number, i.e. the results of an animation will be called:

```
ifrit-an-00001.jpg
ifrit-an-00002.jpg
```

```
ifrit-an-00003.jpg
...
```

(up to 99999 images can be created). In this case it remains your task to convert the set of images into an animation format of your choice (many image viewers will also have a slide show feature, so you can simply display the set of images as a slide show too). In VTK versions 5.0 and above a capability has been added to create MPEG and AVI (on some platforms) movies – the choice is controlled by the **ViewModule:AnimationOutput** property.

Animation mode is only available if the data file name has a specific form. Even if you are going to use just one file for your animation (for example, for a fly–by through a fixed scene), you still need to name your data file in a standard way so that IFrIT can understand it as belonging to the animation series – i.e. as an animatable file.

1.5.2 Animatable Files

Animations are normally made from a series of files. IFrIT will automatically read new files in order to create an animation. The file names **must** be in the following format for IFrIT to find them:

```
[any_string_as_prefix]NNNN.suf
```

where `suf` is `txt` for plain text files or `bin` for binary files (see Supported Files Formats), NNNN is a 4–digit number (from 0000 to 9999) called **record number**, and the file name can contain any string before that as a prefix. For example, the following file names will be recognized by IFrIT as series:

```
myfile0345.bin
var_9988.bin
0564.txt
```

whereas the following file names are valid names for a single data file, but cannot be used in making an animation:

```
myfile345.bin
var_9988a.bin
0564var.txt.
```

Files that belong to a series are called **animatable**.

If IFrIT recognizes the data file as a member of a series, it will first complete all the operations requested for the current file, and then will automatically load the next file in the series. The files in the series do not have to be numbered sequentially. For example, if the series contains only two files, `myfile0345.bin` and `myfile0817.bin`, IFrIT will load `myfile0817.bin` right after `myfile0345.bin`, even if many numbers in between are missing. IFrIT will not leave the Animation mode until all the files in the series are visualized.

1.6 Animation and Control Scripts

1.6.1 Overview

IFrIT can understand two specialized script languages: an **Animator Script** and a **Control Script**. These two scripts are similar, but not identical, since they are used for different goals. The **Animator Script** enhances the capabilities of the **Animator** object, while the **Control Script** is used to operate IFrIT in the **Command–line Shell**. Two scripts, however, "know" about each other: a piece of the **Control Script** can be embedded into the **Animator Script**, and vice versa. The two scripts do share a set of common statements.

1.6.2 Expressions

Both IFrIT scripts understand variables of four types: boolean (logical), integer, float, and strings (which must be enclosed in double quotes). Both scripts accept standard C–type expressions on the right hand side of an assignment operator, or as loop count or **`if`** statement clause. In addition, `x^y` means "`x` in power `y`". Square brackets can be used in a usual way to select one component of an array (`a[1]` means the first component of an array `a`). The only notable exception is that components of script arrays start with 1, not with 0, as in usual C–style arrays. The following functions can be used in the expressions:

> **`abs`**, **`exp`**, **`ceil`**, **`floor`**, **`log`**, **`sqrt`**, **`sin`**, **`cos`**, **`tan`**, **`asin`**, **`acos`**, **`atan`**, **`sinh`**, **`cosh`**, **`tanh`**

In addition to standard scalar values, vectors (3–component float arrays) can also be used in the expressions. A dot symbol "." is used to denote the scalar product of two vectors, and two special functions **`mag`** and **`norm`** can be used to get a magnitude (a scalar) of a vector and a normalized vector. A vector can be assigned a vector–valued construct in the form `(c1,c2,c3)` (no spaces), where `c1`, `c1`, `c3` are either numbers of variable names – no expressions can be used in the vector–valued construct, i.e. the form `(1,2,a)` (`a` is a variable declared earlier) is valid, but the form `(1+1,2,a*2)` is not.

It is **important** to remember that *only 3–component arrays* can be used in expressions, but arrays with a different number of components can still be assigned a single array–valued construct. For example, the two expressions in the following fragment are both valid (since both `v1` and `v2` are declared as 3–component arrays):

```
var float[3] v1, v2
set v1 = (1,2,3)
set v2 = 3*(1,2,3)
```

but in the next fragment the last statement is invalid, because arrays with other than 3 components cannot be used in expressions:

```
var float[4] v1, v2
set v1 = (1,2,3,4)
set v2 = 3*(1,2,3,4)
```

It is illegal to add boolean and either integer or float values, but integers and float numbers can be added together.

1.6.3 Common Statements for Animator and Control Scripts

var *type*[*dimension*] *variable-name* [, *variable-name* ...]

This statement declares new variables with names given by one or more alpha–numeric identifiers *variable-name* (which must be unique throughout the script). The type of the new variables is determined by the keyword *type*, which must be one of the following 3 reserved words: **int**, **float**, or **bool**. If the optional *dimension* in the form [*expression*] is present (where *expression* is an algebraic expression), the variables become 1–dimensional arrays (vectors) of respective size, which must be positive. For example, the following statement declares an two integer arrays p and q with 5 elements each:

```
var int[5] p, q
```

The new variables are initialized to zero. Declared variables can be assigned values and can be used in expressions.

[**set**] *variable-name*[*dimension*] **[+–*]=** *expression*

This statement assigns (or increments in case of +=, –=, or *=) a value given by an algebraic expression *expression* to the variable identified by *variable-name*. The variable must be either a variable declared earlier in the script with the **var** command, or, in the case of the **Animator Script**, one of the pre–defined script variables. If the optional *dimension* in a form [*expression*] is present, and the variable is an array, then the value is assigned to the appropriate component of the array, which must have a value between 1 and the size of the array. The reserved word **set** can, in fact, be omitted. Here are some examples of assignment statements:

```
q[2] = 3
q += (1,2,3,4,5)
q[2] *= q[1]*sin(q[3]^2)
```

loop *expression*		**for** *variable* **to** *expression*
...	or	...
end		**end**

These two pairs of statements form two kinds of simple loops: they repeats the statements between **loop** or **for** and their respective **end** a number of times that is specified by the value of *expression*. In the **for** form, the variable with name *variable* (which must be declared earlier with the **var** command) serves as an index of the loop, that takes values from 1 to the number of iterations. If the number of iterations is less than 1, then the body of the loop is not executed.

if *boolean-expression* **then**
...
else
...
endif

These statements form a normal conditional execution branch, and **`else`** branch can, of course, be omitted. You should use usual C–style comparison operators to compare numbers: <, >, >=, <=, &&, ||, !=.

`embed animator-script` (to be used in the Control Script)
`embed control-script` (to be used in the Animator Script)

The first form of this statement includes a piece of the **Control Script** inside the **Animator Script**, the second form is for the inverse insertion. The piece of the script inserted should begin on the same or the next line, and every line after **`embed`** should start with the symbol greater (>). The embedded piece is immediately executed. For example, here is the piece of the **Animator Script** (blue) embedded into the **Control Script** (black), into which is embedded another piece of the **Control Script** (green):

```
exec DataReader:LoadData/ScalarField/+z1000.txt
embed animator-script
> set frames-per-file = 10
> set style = tumble
> render 2
> embed control-script
> > show Surface
> > animate           # This command is silently ignored
> render 2
hide Surface
```

To avoid infinite recursion, only two levels of embedding are allowed, and only one of the embedded scripts can be the **Animator Script**, i.e. the example above shows the only allowed case of double embedding. Any attempt of further embedding, or calling **`animate`** from the inner **Control Script** will be silently ignored (as commented in the example above).

A script embedded into another has read–only access to all variables and parameters defined in the parent script, but cannot access grandparent's variables. For example, in the following fragment:

```
var int granny
embed animator-script
> var int daddy
> embed control-script
> > var int child
> > kid = daddy + granny   # error: granny is not defined.
> > daddy = kid^2          # error: daddy is read-only.
```

the variable `daddy` is defined in the innermost script, but the variable `granny` will be reported as undefined. Variables from the parent script are defined as read–only, i.e. they can appear on the right hand side of the assignment operator, but cannot be assigned values to.

Note that whitespaces after the command words are important. For example, the following statements are all syntax errors:

```
loop(1+2)
if(i>1)then
```

```
n+=1
```

The correct forms of these statements are:

```
loop (1+2)
if (i>1) then
n +=1
```

1.6.4 Specific Control Script Statements

The following statements are only valid in the control script:

exec *request* [*request* ...] (short form **.e**)
exec-all-objects *request* [*request* ...] (short form **.eo**)
exec-all-windows *request* [*request* ...] (short form **.ew**)

The first form of this command executes one or more control module requests for the current object. A request has the following form:

```
object:property[index]/value[/value...]
```

where *object* is the name of the object to which this request is addressed, *property* is the property (i.e. parameter) that serves the request, and *value* is the value to be assigned to the object parameter which is controlled by this property (can be an array). The optional *index* specified in the square brackets can be used to set a single component of an array. For example, the following command:

```
exec Camera:ParallelProjection/0
```

will set the projection in the visualization window to perspective. The same command can be types in a short form as:

```
.e c:pp/0
```

where **.e** is the short form for **exec**. The first of the following two requests sets the position of the current marker to `(1.0,2.0,3.0)`, while the second request only changes marker's Y coordinate (array indicies start with 1):

```
exec Marker:Position/1.0/2.0/3.0
exec Marker:Position[2]/4.0
```

Two other forms of the **exec** command will execute the request for all instances of the object in the current window and for all instances in all windows respectively. The *value* in the request may be a script expression, but in that case it must be enclosed into braces ({}) even if it is a single variable. For example the following statements are the perfectly valid way to issue a request:

```
var int mode, a, b
mode = 2
a = 2
```

```
b = 12
exec View Module:ParallelProjection/{1-mode/2}
exec Marker:Position[{a}]/{sin(b)}
```

show *object* (short form **.s**)
hide *object* (short form **.h**)

These two commands show/hide a visualization object specified by *object*.

create *object* [**/***type*] (short form **.c**)
delete *object* (short form **.d**)

These two commands create/delete a new instance of a visualization object. The valid values for *object* are `Surface`, `CrossSection`, `ParticleGroup`, and `View Module` (other objects cannot have multiple instances). In the latter case an optional *type* can be specified, which takes two possible values: **copy** (creates a new View Module and copies internal settings from the old one to the new one) or **clone** (a clone shares the data with the parent View Module).

[**set**] **current** *object=id* (short form **.u**)

This command sets the *id* instance of an object *object* as current. Here *id* is an integer number from 1 to the number of instances available. For example, the following 2 commands create a new visualization window (module) and make the second window current:

```
create View Module
current View Module=2
```

Note, that no spaces should be present between *object*, the equality sign, and *id*.

animate [*filename*] (short form **.a**)

This statement starts an animation. If the optional name of the file (which should not be enclosed in quotes) with the **Animation Script** is specified, the script is used for the animation, otherwise the **Animator** objects settings are used for producing the animation. The form with the file name is analogous to the **embed** command, with the only difference that the **embed** command places the text of the **Animator Script** to be embedded directly into the body of the **Control Script**, while the **animate** command reads the **Animator Script** from a file.

render *state* (short form **.r**)

This command toggles whether the scene is rendered after each command; *state* has only two values: **on** or **off**.

The following commands work only for the top–level interactive **Constrol Script**. For an embedded **Control Script** these commands are silently ignored, without generating any error, so that the same script can be used as both the top–level script and as an embedded one.

print *query* (short form **.p**)

This command prints the value of *query*, which can be any script expression, including an object property in the form *object*:*property*. If this script is embedded, this command will produce nothing.

list objects (short form **.lo**)
list properties *object* (short form **.lk**)

The first of these two commands lists all available objects, the second lists all properties for an object specified by *object*

help [*command*] (short form **.q**)
help object *object* (short form **.qo**)
help property *object*:*property* (short form **.qk**)

The first form of this command prints a short help for each of the **Control Script** commands. A call to **help** without an argument lists all script commands with short annotations. The second and third forms produce help information about a specific object or a specific property.

An object property in the form *object*:*property* is considered a script expression, and it can be assigned to variables, but it **cannot** be used in expressions. For example, in the following example, the assignment to `x` is valid, while the assignment to `y` (shown in red) is not, because it is an expression rather than a direct assignment:

```
var float[3] x, y
x = Marker:Position
y = 2*Marker:Position
```

Needless to say, that types and dimensions of the script variable and the object property should match, with two exceptions: a double–valued property can be assigned to a float–valued variable, and a color–valued property is assigned to an integer so that a `(r,g,b)` color is assigned as `r+256*b+65536*b`.

1.6.5 Specific Animator Script Statements

The following statements are only valid in the animator script:

render *expression*
render-all *expression*

This statement evaluates the algebraic expression *expression* and performs that number of consequent renderings of the scene in the current window (the first form) or in all visualization windows (the latter form).

render-set *array*

This statement performs one rendering of the scene in the subset of all visualization windows whose numbers are contained in *array*.

reset

This statement resets all the values of the pre–defined script variables to their default values. This command is useful for making sure that script always executes the same way independently of the internal settings of the **Animator** object.

load *expression*
load scalar-field-file *string*
load particle-set-file *string*
load vector-field-file *string*
load tensor-field-file *string*

The first form of this statement evaluates the algebraic expression *expression* and loads the data set with that record number. As a special case, a *parameter* **next** is allowed instead of the *expression*, in which case the next record is loaded. The last four forms can be used to load one of the four supported file formats which name is specified by *string*. If the *string* starts with the plus sign, a standard directory name will be prepended to *string*.

execure-control-script *filename*

This statement executed the **Control Script** from the file with name *filename* (which must exist and must contain the **Control Script**). This command is analogous to the **embed** command, with the only difference that the **embed** command places the text of the **Control Script** to be embedded directly into the body of the **Animator Script**, while the **execute-control-script** command reads the **Control Script** from a file.

1.6.6 Pre–defined Animator Script Variables

In addition to local variables declared with the **var** command, the **Animator Script** understands several pre–defined (global) variables. These variables can be assigned values to, but cannot be used on the right hand side of an assignment operator, i.e. they don't have any values (in fact, they simply pass the values that are "assigned" to them to other components of IFrIT). In this respect, they are the opposite to *parameters*, which have values but cannot be assigned to.

Pre–defined variables:

- **style** sets the style of the animation to one of the five *parameters*: **static**, **rotate-scale**, **tumble**, **fly-by**, and **camera-path**.
- **frames-per-file** specifies the number of frames to animate for each data file.
- **flyby-speed** sets the speed of the camera in the fly–by mode.
- **rotation-phi** sets the rotation (in degrees) around the vertical axis between the two consequent frames (camera's azimuth).
- **rotation-theta** sets the rotation (in degrees) around the horizontal axis between the two consequent frames (camera's elevation).
- **rotation-roll** sets the rotation (in degrees) around the camera axis (the axis perpendicular to the screen) between the two consequent frames.
- **zoom** sets zooming of the scene between the two consequent frames.
- **cross-section-speed** sets the speed of the cross–section (if it is set to move through the visualization scene) relative to the speed of animation.
- **blended-frames** specifies the number of frames for the gradual blending of subsequent frames.
- **transition-frames** specifies the number of frames for the gradual transition between the two consequent data files (the old scene slowly dissolves while the new scene slowly forms).

- **camera-focal-point**
  camera-view-up
  camera-position are 3D spatial vectors. The value that can be assigned to them should be a 3–component array. Setting these variables moves the focal point, the "view up" vector (the vertical direction from the camera's point of view), and the position of the camera respectively to the specified point.
- **camera-scale** changes the camera parallel scale, i.e. the height of the viewport in world–coordinate distances. Note that the "scale" variable works as an "inverse scale" – larger numbers produce smaller images. This variable has no effect in perspective projection mode.
- **projection** sets the projection of the scene to either **parallel** or **perspective**.
- **record-label**
  color-bars
  bounding-box switch on and off these scene features. They can only be assigned one of two *parameters*: **hidden** or **visible**.
- **record-label-value** allows to specify the value displayed as the record label from the script.
- **surface**
  cross-section
  volume
  particles
  vector-field
  tensor-field can only be assigned one of two *parameters*: **hidden** or **visible**. They switch rendering of respective visualization objects on and off.
- **title-page-file** specifies the name (a double–quoted string) for the image file to use as a title page for the animation.
- **title-page-number-of-frames** specifies the number of frames to display the title page before the start of the animation.
- **title-page-number-of-blended-frames** specifies the number of frames during which the title page gradually dissolves, revealing the visualized scene.
- **logo-file** specifies the name (a double–quoted string) for the image file to use as a logo image in the corner of the visualization scene.
- **logo-opacity** specifies the opacity of the logo as a floating point number between 0 and 1.
- **logo-position** specifies the position of the logo as one of the four possible *parameters*: **upper-left-corner**, **upper-right-corner**, **lower-left-corner**, and **lower-right-corner**.

The following variables are depreciated. They are still supported for backward compatibility, but **embed control-script** command should be used in all new scripts to directly embed a portion of the **Control Script** into the **Animator Script**.

- **scalar-field-lower-limit[N]**
  scalar-field-upper-limit[N] are two arrays. The positive number `N` in square brackets corresponds to the Scalar Field variable number. Setting these variables sets the lower/upper limit for the Scalar Field variable `N` when it is used to paint actors with the color map.
- **particle-set-lower-limit[N]**
  particle-set-upper-limit[N] are two arraya. The positive number `N` in square brackets corresponds to the Particle Set attribute number. Setting this variable sets the upper limit for the Particle Set attribute `N` when it is used to color particles with the color map or to size them with the sizing function.
- **vector-field-upper-limit** sets the maximum value for the Vector Field magnitude when it is used to paint streamlines with a color map.
- **tensor-field-upper-limit** sets the maximum value for the tensor norm.

- **surface-var[N]**
  surface-level[N]
  surface-opacity[N] are arrays. The positive number `N` in square brackets corresponds to the surface instance. A special *parameter* **current** means the current instance. The variables set isosurface variable, level, and opacity for the instance N. If such instance does not exist, this command does nothing (no error is reported).
- **cross-section-var[N]**
  cross-section-dir[N]
  cross-section-position[N] are arrays. The positive number `N` in square brackets corresponds to the cross section instance. A special *parameter* **current** means the current instance. The variables set cross section variable, direction, and position for the instance `N`. If such instance does not exist, this command does nothing (no error is reported).
- **volume-var[N]** is an array. The positive number `N` in square brackets corresponds to the volume instance, but only `N=1` value is supported presently. A special *parameter* **current** means the current instance (it is equivalent to `N=1`). The variable sets the volume variable for the instance `N`. If such instance does not exist, this command does nothing (no error is reported).
- **particles-opacity[N]** is the array. The positive number `N` in square brackets corresponds to the particles group number. A special *parameter* **current** means the current group. This variable sets the particle opacity for the group `N`. If such group does not exist, this command does nothing (no error is reported).
- **particles-downsample-factor** sets the downsample factor for the Basic Particles data.
- **marker-position[N]**
  marker-size[N]
  marker-opacity[N] are arrays. The positive number `N` in square brackets corresponds to the marker instance. A special *parameter* **current** means the current marker. These variables set the position, size, and opacity the marker `N`. If such marker does not exist, this command does nothing (no error is reported).

In addition, a special *parameter* value may be used in expressions:

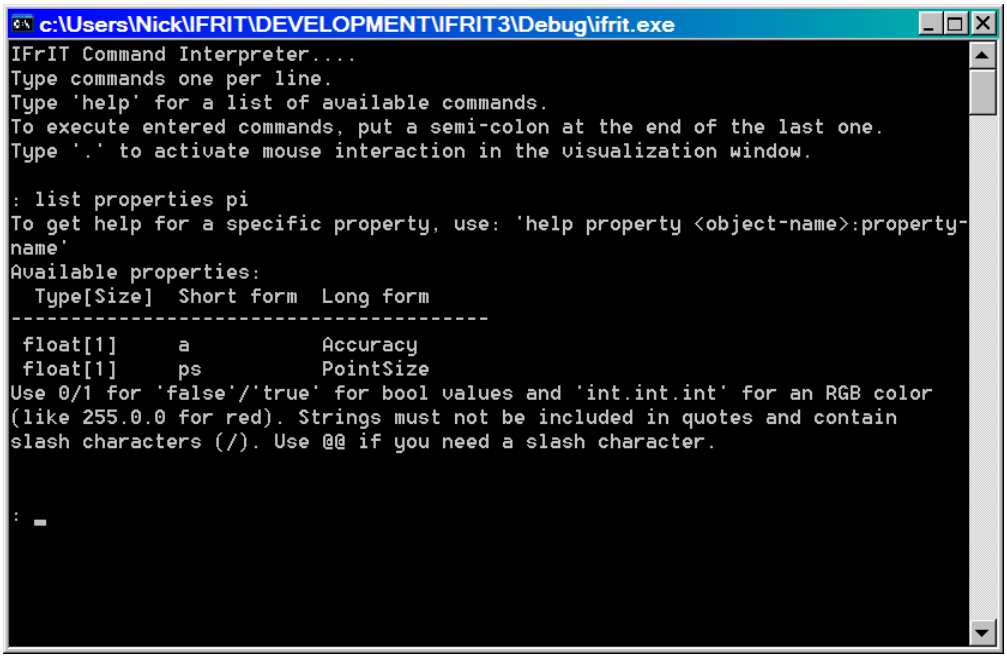
> **marker**`[N]` returns a vector (3–component array) that gives the location of a marker with number `N`. If `N` is less than 1 or greater than the total number of markers in the scene, an error occurs. This parameter is useful for specifying camera's position or focal point at the marker's location.

Some of these variables had slightly different names under IFrIT 2. These names are also understood, but are not listed here as they are depreciated even more.

2 Shell Reference

2.1 Command–line Shell Reference

2.1.1 Command–line Shell



The command–line shell is a little more than an interface to the **Control Script**. Script commands are typed one per line. In order to allow for multi–line commands (like `loop`...`end`), typed commands are not executed immediately, but saved in a buffer. The full buffer gets executed when the typed command is appended with the semi–colon (;). Script commands that produce help information (`help`, `list`, and `print`) are executed immediately even without a semi–colon. Command–line shell also adds the following three commands to the **Control Script**:

`window` (use a single dot **`.`** as a short form)

> Because the command–line shell does not implement an event loop, like in a GUI shell, a keyboard focus has to be switched explicitly between the command–line prompt and the mouse interaction in visualization windows. This command switches the focus from the command line to the visualization windows. Press 'q' in one of the visualization windows to switch the focus back to the command line.

`<` *`filename`*

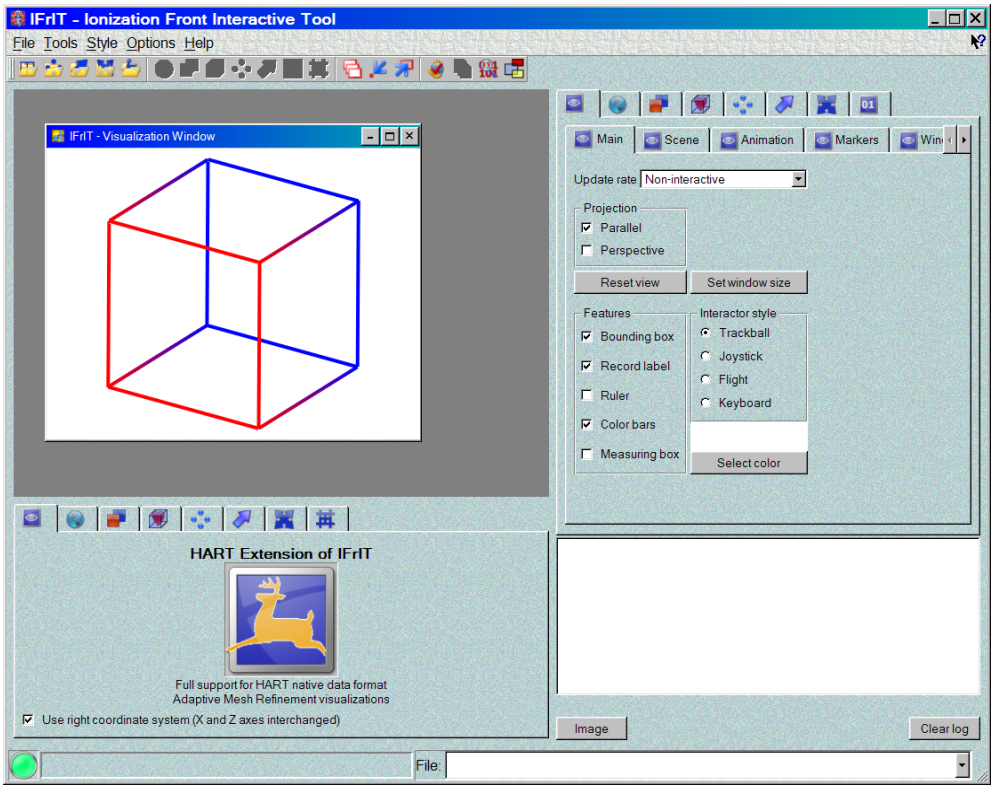
This command includes the text from a file with *filename* into the body of the script (as if it was typed from the keyboard). If the file name starts with the plus sign (+), the name of the IFrIT main directory will be prepended to the file name, replacing the plus sign.

exit
quit

These two commands exit the command–line shell and terminate IFrIT.

2.2 GUI Shell Reference

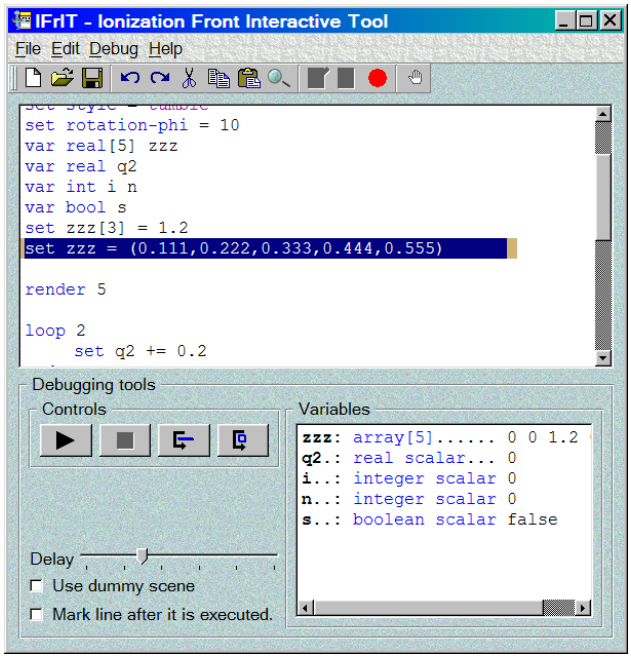
2.2.1 Graphical User Interface (GUI) Shell



The GUI shell provides a fast access to all object properties through a set of GUI **widgets** (on–screen control elements). All widgets have a dynamic help feature: pressing [Cntr] and F1 keys together while a mouse cursor points on a particular GUI widget will pop–up a small help window for that widget.

In addition, the GUI shell includes several dialogs that provide additional functionality, as described below.

2.2.2 Animation Script Debugger



Animation Script Debugger is a tool designed to ease–up debugging of animation scripts. It includes a standard editor with usual **New**, **Open**, **Save**, etc file manipulation functions and **Undo**, **Cut**, **Paste** etc text editing functions for editing the script. The editor supports script syntax highlighting. Script debugging is controlled by menu entries (and tool buttons) for compiling the script, for setting or removing a breakpoint (marked by a different background color of a script statement), and for starting debugging.In the debugging mode a control panel appears at the bottom. The panel includes 4 buttons for running and stopping the script and for stepping the script one line or one frame at a time. The latter two buttons have different behaviour only with statements **render**, **render–all**, and **render–set**. For example, the following statement:

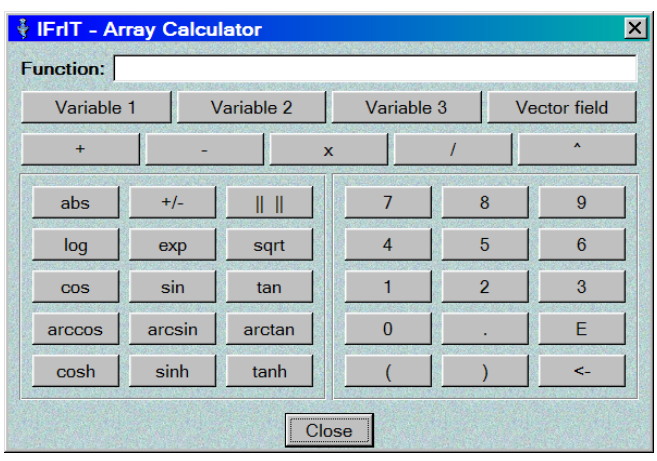
```
render 10
```

will be executed in one step with the step–one–line–at–a–time button, but will require 10 clicks with the step–one–frame–at–a–time button. Breakpoints can be set in the script and will behave as expected.

During the execution, the current script line is highlighted as if it was selected in the editor mode, and selection follows the script as it is being executed. A **Variables** window will display all currently defined script variables and their respective values. In addition, any internal loop counter (like inside **render** statetements or a silent counter of the **loop** statement) will be displayed when it value changes.

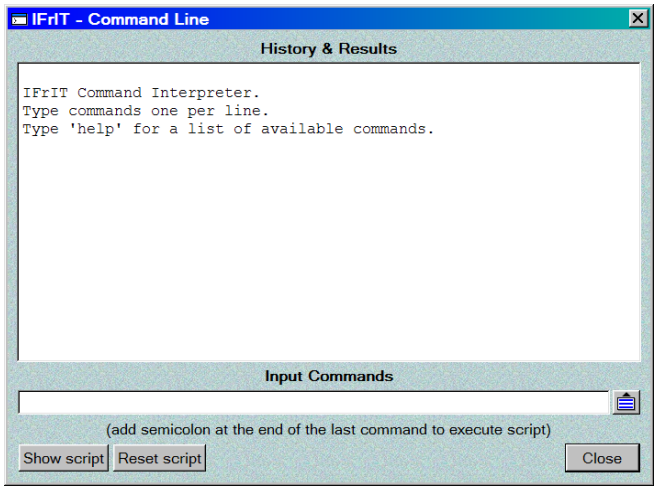
A **delay slider** can be used to adjust a time delay between execution of two subsequent lines – without a delay, the script will be executed as fast as possible, but it might be difficult to follow with the eye what statements are being executed at every moment. The **use dummy scene** box, if checked, will replace the real scene with a simple dummy object, to speed up rendering. In addition, subsequent data files in the dummy mode are not actually loaded from the disk, but only checked to exist.

2.2.3 Array Calculator



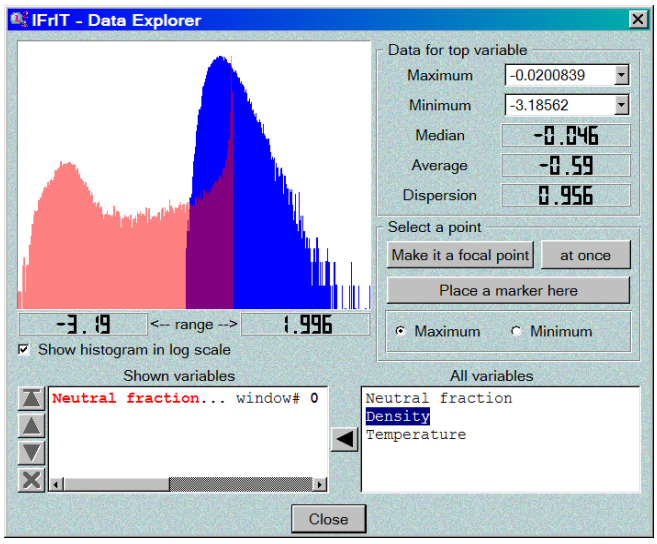
Array Calculator allows a user to perform mathematical manipulations with scalar field variables. Up to first three variables can be used in the expression, and, in addition, a magnitude of the vector field be used as well, if the dimensions of the vector field data coincide with the dimensions of the scalar data. The array calculator window has a calculator–like interface that is straightforward to use.

2.2.4 Command Line



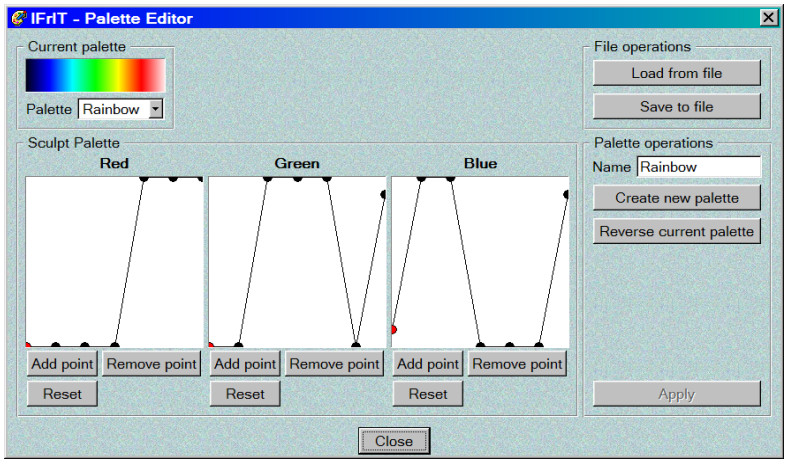
The **Command Line** dialog provides access to **Control Module Requests**, just like in the command–line shell.

2.2.5 Data Explorer



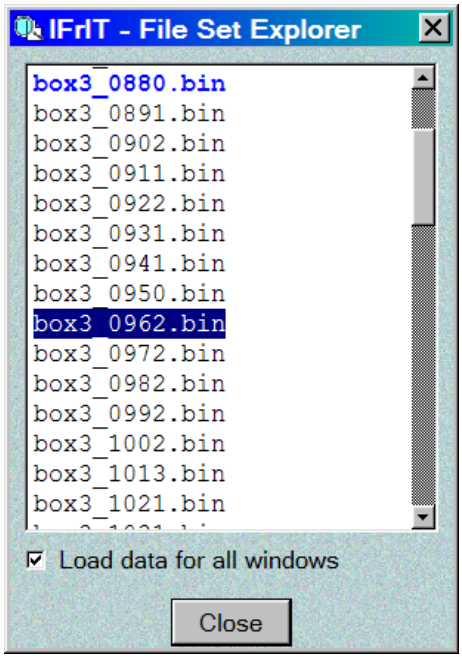
The **Data Explorer** dialog can be used to obtain information about the currently loaded data. The available variables from the "All variables" window can be moved to the "Shown variables" window with a click of a button. The histograms of all shown variables are displayed in the main window, and the information about the top shown variables is listed on the right.

2.2.6 Palette Editor



Palette Editor allows to modify existing palettes, create new ones, save or load custom palettes to/from a file, and remove palettes from the list. Editing of a palette is performed using three windows for sculpting a color component for all three colors. Other functions can be invoked by pressing the respective buttons. **Palette Editor** saves palettes into files with extensions ".ipf" (IFrIT Palette File).

2.2.7 File Set Explorer



File Set Explorer can be used to load an arbitrary member of a file set. The main part of the window is a list of members in the current file set. Clicking with the left mouse button on any member of the set will highlight it. Using the mouse with the left button pressed, several consequitive members can be highlighted at once. Pressing the **`Enter`** key will load selected members of the set one after another. Double clicking on a given member will load this one member too. In addition to using the mouse, a user can also use keyboard navigation (**`up`** and **`down`** keys, **`Page Up`** and **`Page Down`**, and **`Home`** and **`End`** keys) to move along the displayed list. If the **`Control`** key is pressed together with other navigation keys, the selection will change instead.
If the **`Load data for all visualization windows`** box is checked, IFrIT will try to load corresponding members (with the same record number) for all sets displayed in various visualization windows. For example, if window #1 shows the data from a file `aaa_0010.bin`, and window #2 shows the data from a file `bbb_0015.bin`, then loading the file `aaa_0020.bin` in window #1 will also cause loading of the file `bbb_0020.bin` in window #2 (if such a file exists).

2.2.8 Image Composer

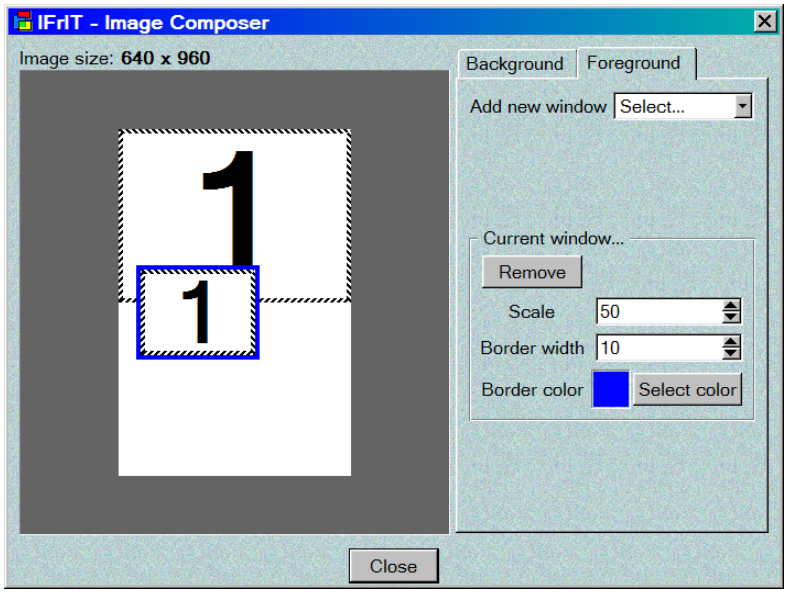
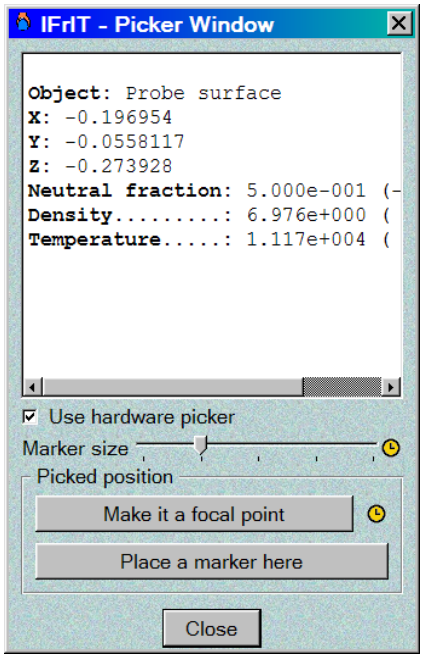


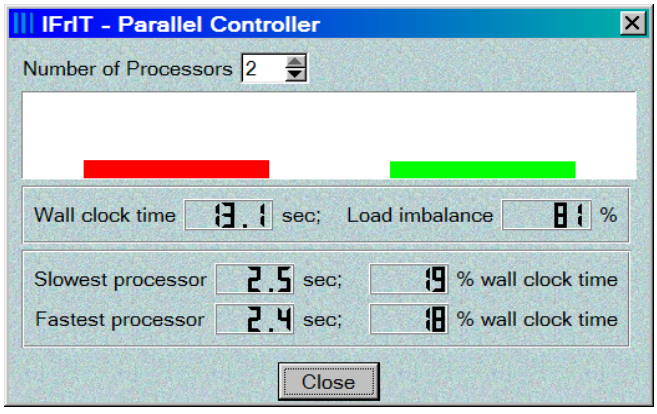
Image Composer dialog is a tool that allows you to organize several visualization windows into one image (both as a snapshot image and as a part of an animation). **Image Composer** has two layers: a background and a foreground. A small book (tab) widget to the right of the drawing area gives you controls for these two layers. Background layer consists of regularly tiled images from one or several visualization windows. You van change the number of tiles in the horizontal and vertical directions, assign a given visualization window to a given tile (or not assign any window at all), add a border to the full image with a selectable color, and specify whether the border only surrounds the whole image or each individual tile as well. Tiles that have no visualization window attached to them can be filled with a background image that should be read from an external file. Usual IFrIT image formats are understood (PNG, JPEG, PNM, BMP, and TIFF). Foreground layer consists of freely floating "inserts" that must be associated with a visualization window. Inserts can be moved around by clicking on them with the left mouse button and dragging them around. They can be scaled to a fraction of their original size and added a border of a predefined color. Check the Reference Guide for the **Image Composer** object for more information.

2.2.9 Picker Window



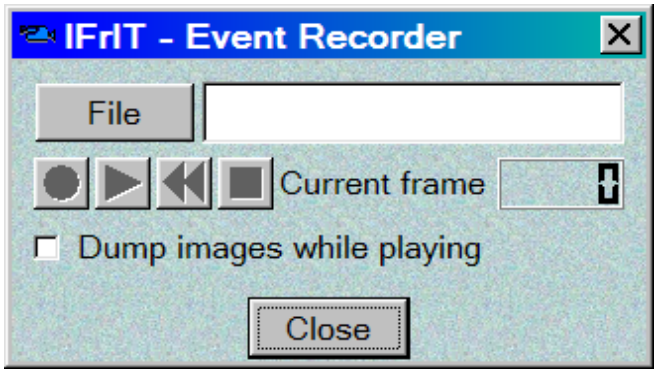
Picker Window is brought up during the pick operation – when you press the "P" key in the visualization window. If anything is picked in your scene, the **Picker Window** will appear, and a small marker sphere will be placed at the picked position. The **Picker Window** has a slider to control the size of the matker and a check box to select hardware vs software picking. The hardware picking is much faster, but may not work properly when translucent objects are present in the scene.

2.2.10 Parallel Controller



Parallel Controller can be used to monitor the parallel performance. Most of IFrIT operations can be done in parallel if more than one processor is available. The maximum possible number of processors that IFrIT will use is set by **`-np`** command–line option.

2.2.11 Event Recorder



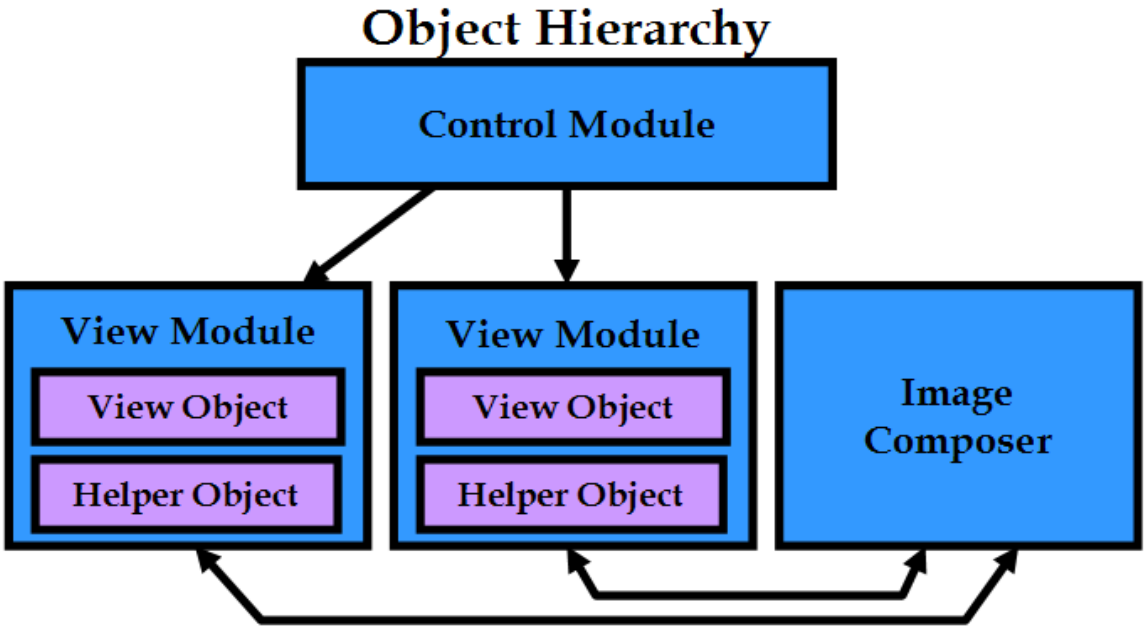
Event Recorder is used to record and play back mouse and keyboard events registered in the visualization windows. Its basic functionality is very similar to a usual tape recorder, except the tape is replaced by a file. As an option, an image of the scene can be dumped after each event is played back – this can serve as a crude way of making animations, although IFrIT supports much better ways of making very complex .

3 Object Reference

3.1 Overview

3.1.1 Components of the IFrIT core

The core of IFrIT consists of objects. Depending on their use, objects are divided into two categories: **modules** and **View Objects**. A hierarchy of IFrIT objects is shown in the figure below.



IFrIT object hierarchy

Modules form the highest level of object hierarchy. A **Control Module** object plays a role of the root of the hierarchy, its main function is to provide coordination between various **View Module**s. A **View Module** appears as an IFrIT visualization window on the screen. All **View Objects** belong to one of the **View Module**s, with the single exception of **ImageComposer** module, which is responsible for composing a snapshot or an animation image from several **View Module**s.

Each **View Module** is an independent part of IFrIT that has its own visualization window on the computer screen, its own set of other objects and, often, its own data. Some of **View Module**s may share the data with another **View Module** – in that case the owner of the data is called a "parent" **View Module**, and sharing **View Module**s are called "clones". Irrespectively of whether a given **View Module** is a clone or a parent, it has its own set of **View Objects**. **View Module**s are directly responsible for a general setup of the visualization scene, such as lighting, camera properties, bounding box, clipping plane, and various accessories (such as rulers, labels, measuring boxes, etc). They "outsource" operations on various components of the visualization scene and manipulations on the data to their own View Objects.

Some **View Objects** are responsible for maintaining various parts of the visualization scene. For example the **Surface** object represents a two−dimensional surface within the visualization scene that samples the three−dimensional scalar data (either as an isosurface of a particular scalar variable, or a specified geometric shape like a sphere or a plane), while the **Particles** object represents a set of particles (points).

Other **View Objects** (also called **Helper Objects**) perform various functions that are not directly represented in the visualization scene. For example, the **Data Reader** object is responsible for loading data files into IFrIT, while the **Animator** object creates animations of your visualization scene in a diverse variety of ways.

Object are manipulated by special requests, which invoke specialized parts of each object called **properties**. You can think of a property as of a special variable: as soon as you assign a new value to the variable, something happens. For example, if you assign a value 1 (which is equivalent to `true`) to the property **ViewModule:BoundingBox**, the bounding box will appear in the visualization scene. Assign 0 (`false`) to the same property, and the bounding box disappears. In the **Command−line Shell** these requests are typed explicitly in the command prompt, but in GUI shells commands are issued by GUI widgets, and a user does not need to know the specific form of every command.

3.1.2 ControlModule Requests

Objects in IFrIT are manipulated by means of requests to the **Control Module** object. Each request is a string in the following format:

```
object:property[index]/value[/value...]
```

where *`object`* is the name of the object to which this request is addressed, *`property`* is the name of the object property that serves the request, and *`value`* is the value to be assigned to the object parameter which is controlled by this property (can be an array). The optional *`index`* specified in the square brackets can be used to set a single component of an array.

Most of object properties correspond to internal settings of an object, i.e. numerical values can both be assigned to them and read from them. Some properties, however, are "read−only", i.e. they give access to a value, but do not allow to change it. A small subset of properties are so−called "action" properties, i.e. they are write−only properties: assigning a value to them causes some action to be performed, without necessarily changing the internal state of an object. One example of action properties is the LoadData property of the **DataReader** object; assigning this property a two−component string array loads a data file from the disk.

3.2 Available objects

Modules:

- **ControlModule**
- **ImageComposer**
- **ViewModule**

View objects:

- **CrossSection**
- **Marker**
- **ParticleGroup**

- **Particles**
- **Surface**
- **Tensor**
- **Vector**
- **Volume**

Helper objects:

- **Animator**
- **Camera**
- **ColorBars**
- **Data–BasicParticles**
- **Data–UniformScalars**
- **Data–UniformTensors**
- **Data–UniformVectors**
- **DataReader**
- **Picker**

3.3 Animator object

Animator creates a series of snapshots ("animation"). It can modify the scene between the subsequent snapshots in a variery of ways, from plain rotation to following a camera path, and load subsequent data files at specified intervals to follow the evolution of the scene, as explained in Animation Support. The animation can start with a title page, and a small logo image can be displayed in every animation frame. **Animator** can work either by using its internal settings, or by following the **Animator Script**.

Short form: **a**

Available properties:

- **CameraPathClosed** (short form: **cpc**; type: **`bool`**; # of arguments: **`1`**)
  A switch that specifies whether the camera path is a closed loop (**PathLoop=1**) or an open curve (**PathLoop=0**).
- **CameraPathDemo** (short form: **cpd**; type: **`bool`**; # of arguments: **`1`**)
  This is an action property, assigning 1 to it causes the camera to follow the camera path without actually loading any new data files or producing animation snapshot images. After the demonstration of the camera path, the camera view is restored to its original state.
- **CameraPathX** (short form: **cpx**; type: **`float`**; # of arguments: **`any`**)
  Properties **CameraPathX**, **CameraPathY**, and **CameraPathZ** set X, Y, and Z coordinates of the points on the camera path. These three properties are arrays, and each of them must have the number of components specified by the NumberOfCameraPathSteps property. Extra components of these arrays will be ignored, and missing components will be undefined. These properties are mostly used for saving and restoring the internal state, since the camera path is most conveniently modified interactively with the mouse using the camera path handles.
- **CameraPathY** (short form: **cpy**; type: **`float`**; # of arguments: **`any`**)
  See CameraPathX.

- **CameraPathZ** (short form: **cpz**; type: **`float`**; # of arguments: **`any`**)
  See CameraPathX.
- **FlybySpeed** (short form: **fs**; type: **`float`**; # of arguments: **`1`**)
  A camera speed in the fly–by mode, as measured by a distance a camera travels between two consequitive frames. The distance is measured in OpenGL units, with the bounding box size being 2.
- **FocalPathEnabled** (short form: **fpe**; type: **`bool`**; # of arguments: **`1`**)
  A switch that toggles the focal point path. The focal point path is a path along which the focal point of the camera moves when the camera is marching along the camera path. Using both the camera path and the focal point path allows to specify an arbitrary transformation of the camera during the animation.
- **FocalPathToPoint** (short form: **fpp**; type: **`bool`**; # of arguments: **`1`**)
  A switch that specifies whether the focal point path is compressed ito a single point (**FocalPointPathToPoint=1**). In that case the camera is pointing towards a single point during the whole animation.
- **FocalPathX** (short form: **fpx**; type: **`float`**; # of arguments: **`any`**)
  Properties **FocalPathX**, **FocalPathY**, and **FocalPathZ** set X, Y, and Z coordinates of the points on the path the camera focal point follows in the camera path mode. These three properties are arrays, and each of them must have the number of components specified by the NumberOfCameraPathSteps property. The focal path may be disabled using the **FocalPathEnabled** property, or compressed into a single point using the **FocalPathToPoint** property.
- **FocalPathY** (short form: **fpy**; type: **`float`**; # of arguments: **`any`**)
  See FocalPathX.
- **FocalPathZ** (short form: **fpz**; type: **`float`**; # of arguments: **`any`**)
  See FocalPathX.
- **InheritSettings** (short form: **is**; type: **`bool`**; # of arguments: **`1`**)
  A switch that specified whether the internal settings of the **Animator** object are retained when the **Animator Script** is used. If **InheritSettings=0**, the **Animator Script** starts with all settings initialized to zero. This is equivalent to calling **`reset`** command as the first command of the script.
- **LogoFile** (short form: **lf**; type: **`string`**; # of arguments: **`1`**)
  The name of the file with the image to be used as a logo shown on every animation snapshot image. The image must be in one of the formats understood by VTK (JPEG, PNM, BMP, TIFF, and PNG). Logo images are usually small, if the image in the file exceeds 20% of the animation snapshot image, it will be scaled down. The logo image can be placed in one of the four corners of the animation image with the LogoPosition property, and its opacity (transparency) can be controlled with the LogoOpacity property.
- **LogoOpacity** (short form: **lo**; type: **`float`**; # of arguments: **`1`**)
  The opacity of the logo image, given as a floating point number from 0 to 1. Zero opacity makes the logo transparent (invisible), while opacity of 1 makes it solid (non–transparent).
- **LogoPosition** (short form: **lp**; type: **`int`**; # of arguments: **`1`**)
  The position of the logo image on the screen. Integer numbers from 0 to 3 are accepted, as follows:
  - upper–left corner: LogoPosition=0
  - upper–right corner: LogoPosition=1
  - lower–left corner: LogoPosition=2
  - lower–right corner: LogoPosition=3
- **Mode** (short form: **m**; type: **`int`**; # of arguments: **`1`**)
  A method of transformation of a scene between the two subsequent frames. The first frame of the animation is produced from the current scene. After the first frame, IFrIT can transform the scene for the next frame in one of the five ways:
  - **static (Mode=0)**: the scene does not change from a frame to frame. The scene will change only if new data files are loaded.

- **rotate/scale (Mode=1)**: the scene rotates (and optionally scales) during the animation by a predefined amount (set with **Phi**, **Theta**, **Roll**, and **DZoom** properties).
- **tumble (Mode=2)**: the scene tumbles in a random fashion during the animation, i.e. it rotates (and optionally scales) by a predefined amount in the direction that slowly changes by a random amount. This is a nice feature for making animations that show the whole scene all the time.
- **fly−by (Mode=3)**: the camera flies through the visualization scene along a randomly precessing trajectory. This feature allows to focus on the detail, but the whole scene may not be visible for most of the time. IFrIT however takes care to insure that the camera always points toward the central part of the scene. The starting position of the camera depends on the projection mode: in the perspective projection mode the camera starts from the current position, in the parallel projection mode the camera moves closer to the center of the scene before starting a fly−by.
- **camera path (Mode=4)**: an interactively adjustable curve is placed in the scene which serves as a path along which the camera moves during the animation. Optionally, another curve can be placed into the scene along which the focal point of the camera moves. These curves will not show in the images produced during the animation.

- **NumberOfBlendedFrames** (short form: **nb**; type: **`int`**; # of arguments: **`1`**)
  The number of consequitive frames that are blended together in each animation snapshot. Blending frames creates an impression of gradual transition from one frame to another.
- **NumberOfCameraPathHandles** (short form: **nph**; type: **`int`**; # of arguments: **`1`**)
  The number of "handles" on the camera path. Handles are used to interactively morph the camera path by dragging them around with the mouse. More handles will make the camera path more flexible, but the scene more crowded.
- **NumberOfCameraPathSteps** (short form: **nps**; type: **`int`**; # of arguments: **`1`**)
  The number of steps along the camera path. In the camera path mode, the camera moves along the path one step per frame. If the path is too short, the animation will stop prematurely. Selecting the correct number of steps along the camera path for complex animations may be a non−trivial task. For examaple, if you are animating a sequence of files with, say, 100 files, and specified 10 frames per file, then the number of steps along the camera path should be 1000 – then the camera reaches the end of the path by the time the last file is used up.
- **NumberOfFrames** (short form: **nf**; type: **`int`**; # of arguments: **`1`**)
  The number of frames for each input data file. This option can be used in several ways. For example, if you have only one file in a series, using this option you can create an animation of rotation or fly−by through your data series. If you have a whole series of data files, by setting the number of frames per file you can control the speed with which the animation is played. For example, if you have just 30 data files, they will be played in 1 second with a regular MPEG stream. By setting the number of frames per file to 10, you extend your animation for 10 seconds (with 3 frames per second it will still look ok if the differences between subsequent data files are not great).
- **NumberOfTitlePageBlendedFrames** (short form: **tbf**; type: **`int`**; # of arguments: **`1`**)
  The number of frames during which the title page is blended with the first image of the visualization scene. Blending creates an effect of a title page slowly dissolving to reveal the visualization scene.
- **NumberOfTitlePageFrames** (short form: **tnf**; type: **`int`**; # of arguments: **`1`**)
  The number of frames during which the title page is displayed at the beginning of the animation.
- **NumberOfTransitionFrames** (short form: **nt**; type: **`int`**; # of arguments: **`1`**)
  The number of consequitive frames that are blended together when a new data file is read. This property is similar to **NumBlendedFrames**, but only blends frame after a file read, other frames are saved into the image files as they are.
- **Phi** (short form: **dp**; type: **`float`**; # of arguments: **`1`**)
  Properties **Phi**, **Theta**, and **Roll** control the angles (in degrees) by which the scene is rotated between the two consequitive frames. **Phi** and **Theta** are usual spherical angles, i.e. phi direction is horizontal

and theta direction is vertical. **Roll** angle is the angle around the axis parallel to the axis of the camera (an axis perpendicular to the plane of the screen). In the **tumble** mode the direction of rotation changes, so only the total angle of rotation matters.

- **PositionOnPath** (short form: **pop**; type: `int`; # of arguments: `1`)
  An integer property that specifies the camera position (step number) on the path.
- **RestoreCamera** (short form: **rc**; type: `bool`; # of arguments: `1`)
  A switch that specifies whether the came view should be restored after the animation.
- **Roll** (short form: **dr**; type: `float`; # of arguments: `1`)
  See Phi.
- **ScriptFileName** (short form: **sfn**; type: `string`; # of arguments: `1`)
  The name of the file with the text of the **Animator Script** to be used if the animation is to run with the script.
- **SlideSpeed** (short form: **ss**; type: `float`; # of arguments: `1`)
  The speed of the **CrossSection** object (measured as a distance by which cross section moves between the two consequitive frames) during the animation. The current cross section will move through the bounding box with this speed. If multiple instances of the **CrossSection** object are shown, only the current instance will move during the animation. The cross section will bounce off the bounding box edge, if it reaches it.
- **StopOnPath** (short form: **sop**; type: `bool`; # of arguments: `1`)
  A boolen property that instructs the **Animator** to stop on the camera path and continue the animation, keeping the camera fixed in space.
- **Theta** (short form: **dt**; type: `float`; # of arguments: `1`)
  See Phi.
- **TitlePageFile** (short form: **tf**; type: `string`; # of arguments: `1`)
  The name of the file with the image that should be used as title page for the animation. The image must be in one of the formats understood by VTK (JPEG, PNM, BMP, TIFF, and PNG). If the size of the title page image is different than the size of the animation snapshot image, the title page image will be smoothly stretched to match the animation image. The title page will be displayed for a number of frames set by the NumTitlePageFrames property. After that, a number of frames specified by the NumTitlePageBlendedFrames property will be blended with the first image of the visualization scene, producing an effect of a gradually dissolving title page.
- **UseScript** (short form: **us**; type: `bool`; # of arguments: `1`)
  A switch that specifies whether the animation must be made using the **Animator Script** (if **UseScript**=1) or the current setting of the **Animator** object (if **UseScript**=0).
- **Zoom** (short form: **dz**; type: `float`; # of arguments: `1`)
  A zooming factor by which the scene is scaled between the two consequitive frames.

3.4 Camera object

Camera object represents the current camera that observes the scene. The camera supports two projection modes: parallel (orthogobal) and perspective. The orientation of the camera are set by its **Position**, a **FocalPoint** (the point in space the camera is directly looking at), and direction in space that the camera considers to be "up" (**ViewUp** vector).

Short form: **c**

Available properties:

- **Azimuth** (short form: **a**; type: **`float`**; # of arguments: **`1`**)
  Rotate the camera horizontally (about the view up vector) centered at the focal point. This is the action property that specifies the angle of rotation in degrees.
- **ClippingRange** (short form: **cr**; type: **`double`**; # of arguments: **`2`**)
  This property sets the location of the near and far clipping planes along the direction of projection (in OpenGL coordinates). Both of these values must be positive. How the clipping planes are set can have a large impact on how well z–buffering works. In particular the front clipping plane can make a very big difference. Setting it to 0.01 when it really could be 1.0 can have a big impact on your z–buffer resolution farther away.
- **ClippingRangeAuto** (short form: **cra**; type: **`bool`**; # of arguments: **`1`**)
  Switch the automatic adjustment of the **Clipping Range** on and off. In almost all cases this property should be set.
- **Elevation** (short form: **e**; type: **`float`**; # of arguments: **`1`**)
  Rotate the camera vertically (about the cross product of the direction of projection and the view up vector) centered on the focal point. This is the action property that specifies the angle of rotation in degrees.
- **EyeAngle** (short form: **ea**; type: **`float`**; # of arguments: **`1`**)
  This property specifies the separation between eyes (in degrees) in the stereo mode.
- **FocalPoint** (short form: **f**; type: **`double`**; # of arguments: **`3`**)
  The property that specifies the focal point of the camera.
- **ParallelProjection** (short form: **pp**; type: **`bool`**; # of arguments: **`1`**)
  Sets the camera projection mode to parallel (if `true`), or perspective (if `false`).
- **ParallelScale** (short form: **ps**; type: **`float`**; # of arguments: **`1`**)
  This property sets the scaling used for a parallel projection, i.e. the height of the viewport in OpenGL distances. Note that the **ParallelScale** parameter works as an "inverse scale" – larger numbers produce smaller images. This method has no effect in perspective projection mode.
- **Pitch** (short form: **p**; type: **`float`**; # of arguments: **`1`**)
  Rotate the focal point vertically (about the cross product of the view up vector and the direction of projection, centered at the camera's position. This is the action property that specifies the angle of rotation in degrees.
- **Position** (short form: **x**; type: **`double`**; # of arguments: **`3`**)
  The position of the camera.
- **Reset** (short form: **rs**; type: **`bool`**; # of arguments: **`1`**)
  This is an action property that resets the camera so that the nearest side of the bounding box faces the screen.
- **Roll** (short form: **r**; type: **`float`**; # of arguments: **`1`**)
  Rotate the camera about the direction of projection. This is the action property that specifies the angle of rotation in degrees
- **ViewAngle** (short form: **va**; type: **`float`**; # of arguments: **`1`**)
  This property specifies the width (in degrees) of the field of view.
- **ViewAngleVertical** (short form: **vav**; type: **`bool`**; # of arguments: **`1`**)
  A boolen property that specifies whether the **ViewAngle** is measured vertically or horizontally.
- **ViewUp** (short form: **u**; type: **`double`**; # of arguments: **`3`**)
  The property that specifies the unit direction that camera considers to be "up".
- **Yaw** (short form: **y**; type: **`float`**; # of arguments: **`1`**)
  Rotate the focal point horizontally (about the view up vector centered at the camera's position). This is the action property that specifies the angle of rotation in degrees.
- **Zoom** (short form: **z**; type: **`float`**; # of arguments: **`1`**)
  In perspective mode, decrease the view angle by the specified factor. In parallel mode, decrease the parallel scale by the specified factor. A value greater than 1 is a zoom–in, a value less than 1 is a zoom–out.

3.5 ColorBars object

ColorBars show a correspondence between the color of a point in space and the value of scalar variable which this color represents. They appear as two bars with text captions on the left and right side of the visualization window. Normally, IFrIT will display color bars automatically, but they can also be manipulated manually.

Short form: **cb**

Available properties:

- **Automatic** (short form: **a**; type: **`bool`**; # of arguments: **`1`**)
  A switch specifying whether the **ColorBars** are in the automatic mode or not. In the automatic mode, IFrIT determines what variables to display at what color bar. In the manual mode the contents of each color bar is determined by the user.
- **BarLeft** (short form: **bl**; type: **`int`**; # of arguments: **`3`**)
  See BarRight.
- **BarRight** (short form: **br**; type: **`int`**; # of arguments: **`3`**)
  BarLeft and **BarRight** properties set the properties of each of the two color bars. The argument to this property is a 3–component integer array, with three components being the Scalar Field variable, the active data type, and the palette id displayed on the color bar. The palette id is the same quantity used to specify a palette for **View Objects** (palette index if >0 and –palette index if
- **Color** (short form: **c**; type: **`color`**; # of arguments: **`1`**)
  A color of the text legend. In properties colors are specified as 3–component RGB values `int.int.int` (like 255.0.0 for red).
- **SideOffset** (short form: **so**; type: **`float`**; # of arguments: **`1`**)
  A distance by which the color bars are offset from the side of the visualization window, in the fraction of the window width.

3.6 ControlModule object

ControlModule establishes interactions between various visualization windows. For example, it can synchronize all windows, so that mouse interaction in one window affects all other windows in the same way. It is also responsible for channeling requests to other objects.

Short form: **cm**

Available properties:

- **AutoRender** (short form: **ar**; type: **`bool`**; # of arguments: **`1`**)
  A switch controlling whether the visualization windows are rendered automatically after every request, or a render command needs to be issued explicitly. Normally, you do not need to use this

switch.

- **OptimizationMode** (short form: **om**; type: **`int`**; # of arguments: **`1`**)
  This integer property specifies the way IFrIT tries to optimize its performance. If it is set to 0, IFrIT will try to optimize for speed, using extra memory whenever it can lead to faster performance; the value 1 will set optimization for memory, and IFrIT will try to minimize its memory use at the expense of doing extra calculations. With the value 2, IFrIT will optimize for quality, i.e. it will try to make the highest quality rendering of the scene even if it takes more memory and more computing time.
- **SynchronizeCameras** (short form: **sc**; type: **`bool`**; # of arguments: **`1`**)
  A boolean "action" (write–only) property synchronizing (i.e. orienting them in the same way) cameras in different visualization windows.
- **SynchronizeInteractors** (short form: **si**; type: **`bool`**; # of arguments: **`1`**)
  A boolean property switching the interactor synchronization between different visualization windows on and off. If interactors in different visualization windows are synchronized, mouse interaction in one window will affect all other visualization windows. This can be useful for, for example, comparing two different data files, or for looking at two sides of the same scene. If the interactors are synchronized, the views in different windows do not have to be same – different cameras may be oriented differently, but they will all move in unison.

3.7 CrossSection object

A **Cross Section** object shows a section (a slice) of the visualization scene. The slice is orthogonal, i.e. it is always perpendicular to either X, or Y, or Z axis. Use a **Surface** object to show an arbitrary oriented slice. Because of the restricted orientation, the **Cross Section** object is significantly faster to visualize than a **Surface** object. Multiple slices can exists simulatenously.

Short form: **x**

Available properties:

- **Ambient** (short form: **ma**; type: **`float`**; # of arguments: **`1`**)
  Properties **Ambient**, **Diffuse**, **Specular**, and **SpecularPower** control material properties of the object, as specified in OpenGL. The first three take a floating–point value between 0 and 1, while the **SpecularPower** property, which controls the OpenGL *shininess* property (it is called *specular power* in VTK), takes a floating–point value between 0 and 128. You need to know how OpenGL handles material properties to understand what these values mean.
- **AttachToMarker** (short form: **am**; type: **`int`**; # of arguments: **`1`**)
  If this property is set to the value of the index of an existing **Marker**, then the location of the cross section will be "attached" to a newly created **Marker**. If the **Marker** is moved, the cross section will move with it too.
- **ClipPlane** (short form: **cp**; type: **`bool`**; # of arguments: **`1`**)
  A switch activating or deactivating the clipping plane. Takes a boolean (0/1) value.
- **DataPresent** (short form: **dp**; type: **`bool`**; # of arguments: **`1`**)
  A read–only boolean property returning 1 if the object has the data and 0 otherwise.
- **DataType** (short form: **dt**; type: **`int`**; # of arguments: **`1`**)
  This property specifies the type of data a given object uses. It has no effect for the basic edition of IFrIT, but extensions to IFrIT use multiple data types and this property is used to select among

different types of data.

- **Diffuse** (short form: **md**; type: **`float`**; # of arguments: **`1`**)
  See Ambient.
- **Dir** (short form: **d**; type: **`int`**; # of arguments: **`1`**)
  Orientation of the **Cross Section** object. Takes one of the values 0, 1, or 2 (the slice is perpendicular to the X, Y, or Z axis respectively).
- **InterpolateData** (short form: **id**; type: **`bool`**; # of arguments: **`1`**)
  The boolean property that specifies whether the data are linearly interpolated on the cross section.
- **Location** (short form: **x**; type: **`double`**; # of arguments: **`1`**)
  The current location of the **Cross Section** along its direction (see also the **Position** property).
- **Method** (short form: **m**; type: **`int`**; # of arguments: **`1`**)
  Sets one of the available methods to render a cross section: **polygons** (0) or **texture** (1). The textute method is usually faster and gives the best quality, but it may not be always available.
- **OverTheEdgeFlag** (short form: **oe**; type: **`bool`**; # of arguments: **`1`**)
  A boolean read–only property that returns `true` if the **Cross Section** reached the edge of the box. It is probably only useful for internal purposes.
- **Palette** (short form: **p**; type: **`int`**; # of arguments: **`1`**)
  See Color.
- **Position** (short form: **x3**; type: **`double`**; # of arguments: **`3`**)
  A 3D vector that specifies the current location of the **Cross Section** along all 3 directions. This an alternative way of setting the **Cross Section** location, similar to the **Location** property.
- **PositionToPickedPoint** (short form: **p2p**; type: **`bool`**; # of arguments: **`1`**)
  This is an "action" (write–only) property that moves the location of the cross section to the currently picked position. If the cross section is attached to a marker, the attachment is broken.
- **Ready** (short form: **r**; type: **`bool`**; # of arguments: **`1`**)
  A read–only boolean value for checking whether the object is ready to be shown (i.e. has the data and all the correct settigs). An attept to show an object that is not ready will not succeed.
- **Shading** (short form: **mw**; type: **`bool`**; # of arguments: **`1`**)
  A boolean switch that can be used to switch off shading of light on the material (so that a surface looks the same no matter what its orientation relative to lights and a camera is).
- **SpecialLocation** (short form: **loc**; type: **`int`**; # of arguments: **`1`**)
  This integer "action" (write–only) property moves the cross–section to a special location in the current data, depending on the property value. The valid values are
  - **0**: place at the location of the minimum of current variable;
  - **1**: place at the location of the maximum of current variable;
  - **2**: place at 1/4 of the box size along its direction;
  - **3**: place at 1/2 of the box size along its direction;
  - **4**: place at 3/4 of the box size along its direction;
- **Specular** (short form: **ms**; type: **`float`**; # of arguments: **`1`**)
  See Ambient.
- **SpecularPower** (short form: **mp**; type: **`float`**; # of arguments: **`1`**)
  See Ambient.
- **Var** (short form: **v**; type: **`int`**; # of arguments: **`1`**)
  The scale variable to show on the **Cross Section**.
- **Visible** (short form: **vis**; type: **`bool`**; # of arguments: **`1`**)
  A read–only boolean value for checking whether the object is visible. It has a value of **true** (1) is the **View Object** is visible, and **false** otherwise. This property cannot be used to show/hide the object, only to test the visibility state.

3.8 Data–BasicParticles object

This object represents the BasicParticles data type.

Short form: **d–bp**

Available properties:

- **DataPresent** (short form: **dp**; type: **`bool`**; # of arguments: **`1`**)
  A read–only boolean property checking whether the data has been loaded.
- **FileName** (short form: **fn**; type: **`string`**; # of arguments: **`1`**)
  This property returns the name of last data file read. The property is read–only – assigning it a name of a file will not cause this file to be read. Use **DataReader:LoadData** property for loading data.
- **FixedLimits** (short form: **fl**; type: **`bool`**; # of arguments: **`1`**)
  A boolean switch toggling whether the range for available variables (properties **Max and Min**) is taken as the range of the data in the current file (if **FixedLimits**=`false`), or as a fixed range unchanged after loading of a data file.
- **FixedStretch** (short form: **fs**; type: **`bool`**; # of arguments: **`any`**)
  This boolean property specifies whether the stretch for variables can be changed. If it is set to **true** (1), the stretch is fixed once and for all. This property is read–only and cannot be changed. All basic IFrIT data types have adjustable stretches, but this property may be set to **true** in an extension, preventing a user from changing a stretch of a variable that – according to the judgement of the extension author – should be used only with a specific stretch.
- **LowerLimit** (short form: **lo**; type: **`float`**; # of arguments: **`any`**)
  Array properties **LowerLimit** and **UpperLimit** have dimensions given by the **NumVars** property and they take values in this range specified by **Max and Min** properties. These two properties are used by **View Objects** to create a color representation of a scalar value with a **Palette**.
- **Max** (short form: **max**; type: **`float`**; # of arguments: **`any`**)
  Array properties **Min** and **Max** control the range for each available variable.
- **Min** (short form: **min**; type: **`float`**; # of arguments: **`any`**)
  See Max.
- **Name** (short form: **na**; type: **`string`**; # of arguments: **`any`**)
  A string–valued **Name** array sets the names of all available variables. Names are displayed on the scene as **Color Bars** captions.
- **NumVars** (short form: **n**; type: **`int`**; # of arguments: **`1`**)
  The read–only property **NumVars** reports the number of available variables in the current data.
- **Stretch** (short form: **s**; type: **`int`**; # of arguments: **`any`**)
  A **Stretch** property sets the stretch (value 0: linear, value 1: logarithmic) for each available variable.
- **UpperLimit** (short form: **up**; type: **`float`**; # of arguments: **`any`**)
  See LowerLimit.

3.9 Data–UniformScalars object

This object represents the UniformScalars data type.

Short form: **d–us**

Available properties:

- **DataPresent** (short form: **dp**; type: **`bool`**; # of arguments: **`1`**)
  A read–only boolean property checking whether the data has been loaded.
- **FileName** (short form: **fn**; type: **`string`**; # of arguments: **`1`**)
  This property returns the name of last data file read. The property is read–only – assigning it a name of a file will not cause this file to be read. Use **DataReader:LoadData** property for loading data.
- **FixedLimits** (short form: **fl**; type: **`bool`**; # of arguments: **`1`**)
  A boolean switch toggling whether the range for available variables (properties **Max** and **Min**) is taken as the range of the data in the current file (if **FixedLimits**=`false`), or as a fixed range unchanged after loading of a data file.
- **FixedStretch** (short form: **fs**; type: **`bool`**; # of arguments: **`any`**)
  This boolean property specifies whether the stretch for variables can be changed. If it is set to **true** (1), the stretch is fixed once and for all. This property is read–only and cannot be changed. All basic IFrIT data types have adjustable stretches, but this property may be set to **true** in an extension, preventing a user from changing a stretch of a variable that – according to the judgement of the extension author – should be used only with a specific stretch.
- **LowerLimit** (short form: **lo**; type: **`float`**; # of arguments: **`any`**)
  Array properties **LowerLimit** and **UpperLimit** have dimensions given by the **NumVars** property and they take values in this range specified by **Max** and **Min** properties. These two properties are used by **View Objects** to create a color representation of a scalar value with a **Palette**.
- **Max** (short form: **max**; type: **`float`**; # of arguments: **`any`**)
  Array properties **Min** and **Max** control the range for each available variable.
- **Min** (short form: **min**; type: **`float`**; # of arguments: **`any`**)
  See Max.
- **Name** (short form: **na**; type: **`string`**; # of arguments: **`any`**)
  A string–valued **Name** array sets the names of all available variables. Names are displayed on the scene as **Color Bars** captions.
- **NumVars** (short form: **n**; type: **`int`**; # of arguments: **`1`**)
  The read–only property **NumVars** reports the number of available variables in the current data.
- **Stretch** (short form: **s**; type: **`int`**; # of arguments: **`any`**)
  A **Stretch** property sets the stretch (value 0: linear, value 1: logarithmic) for each available variable.
- **UpperLimit** (short form: **up**; type: **`float`**; # of arguments: **`any`**)
  See LowerLimit.

3.10 Data–UniformTensors object

This object represents the UniformTensors data type.

Short form: **d–ut**

Available properties:

- **DataPresent** (short form: **dp**; type: **`bool`**; # of arguments: **`1`**)
  A read–only boolean property checking whether the data has been loaded.

- **FileName** (short form: **fn**; type: **`string`**; # of arguments: **`1`**)
  This property returns the name of last data file read. The property is read–only – assigning it a name of a file will not cause this file to be read. Use **DataReader:LoadData** property for loading data.
- **FixedLimits** (short form: **fl**; type: **`bool`**; # of arguments: **`1`**)
  A boolean switch toggling whether the range for available variables (properties **Max and Min**) is taken as the range of the data in the current file (if **FixedLimits**=`false`), or as a fixed range unchanged after loading of a data file.
- **FixedStretch** (short form: **fs**; type: **`bool`**; # of arguments: **`any`**)
  This boolean property specifies whether the stretch for variables can be changed. If it is set to **true** (1), the stretch is fixed once and for all. This property is read–only and cannot be changed. All basic IFrIT data types have adjustable stretches, but this property may be set to **true** in an extension, preventing a user from changing a stretch of a variable that – according to the judgement of the extension author – should be used only with a specific stretch.
- **LowerLimit** (short form: **lo**; type: **`float`**; # of arguments: **`any`**)
  Array properties **LowerLimit** and **UpperLimit** have dimensions given by the **NumVars** property and they take values in this range specified by **Max and Min** properties. These two properties are used by **View Objects** to create a color representation of a scalar value with a **Palette**.
- **Max** (short form: **max**; type: **`float`**; # of arguments: **`any`**)
  Array properties **Min** and **Max** control the range for each available variable.
- **Min** (short form: **min**; type: **`float`**; # of arguments: **`any`**)
  See Max.
- **Name** (short form: **na**; type: **`string`**; # of arguments: **`any`**)
  A string–valued **Name** array sets the names of all available variables. Names are displayed on the scene as **Color Bars** captions.
- **NumVars** (short form: **n**; type: **`int`**; # of arguments: **`1`**)
  The read–only property **NumVars** reports the number of available variables in the current data.
- **Stretch** (short form: **s**; type: **`int`**; # of arguments: **`any`**)
  A **Stretch** property sets the stretch (value 0: linear, value 1: logarithmic) for each available variable.
- **UpperLimit** (short form: **up**; type: **`float`**; # of arguments: **`any`**)
  See LowerLimit.

3.11 Data–UniformVectors object

This object represents the UniformVectors data type.

Short form: **d–uv**

Available properties:

- **DataPresent** (short form: **dp**; type: **`bool`**; # of arguments: **`1`**)
  A read–only boolean property checking whether the data has been loaded.
- **FileName** (short form: **fn**; type: **`string`**; # of arguments: **`1`**)
  This property returns the name of last data file read. The property is read–only – assigning it a name of a file will not cause this file to be read. Use **DataReader:LoadData** property for loading data.
- **FixedLimits** (short form: **fl**; type: **`bool`**; # of arguments: **`1`**)
  A boolean switch toggling whether the range for available variables (properties **Max and Min**) is taken as the range of the data in the current file (if **FixedLimits**=`false`), or as a fixed range

unchanged after loading of a data file.

- **FixedStretch** (short form: **fs**; type: **`bool`**; # of arguments: **`any`**)
  This boolean property specifies whether the stretch for variables can be changed. If it is set to **true** (1), the stretch is fixed once and for all. This property is read–only and cannot be changed. All basic IFrIT data types have adjustable stretches, but this property may be set to **true** in an extension, preventing a user from changing a stretch of a variable that – according to the judgement of the extension author – should be used only with a specific stretch.
- **LowerLimit** (short form: **lo**; type: **`float`**; # of arguments: **`any`**)
  Array properties **LowerLimit** and **UpperLimit** have dimensions given by the **NumVars** property and they take values in this range specified by **Max and Min** properties. These two properties are used by **View Objects** to create a color representation of a scalar value with a **Palette**.
- **Max** (short form: **max**; type: **`float`**; # of arguments: **`any`**)
  Array properties **Min** and **Max** control the range for each available variable.
- **Min** (short form: **min**; type: **`float`**; # of arguments: **`any`**)
  See Max.
- **Name** (short form: **na**; type: **`string`**; # of arguments: **`any`**)
  A string–valued **Name** array sets the names of all available variables. Names are displayed on the scene as **Color Bars** captions.
- **NumVars** (short form: **n**; type: **`int`**; # of arguments: **`1`**)
  The read–only property **NumVars** reports the number of available variables in the current data.
- **Stretch** (short form: **s**; type: **`int`**; # of arguments: **`any`**)
  A **Stretch** property sets the stretch (value 0: linear, value 1: logarithmic) for each available variable.
- **UpperLimit** (short form: **up**; type: **`float`**; # of arguments: **`any`**)
  See LowerLimit.

3.12 DataReader object

DataReader is responsible for loading data files of all types from the disk. It can also perform calculations on Uniform Scalars data.

Short form: **dr**

Available properties:

- **BoundaryConditions** (short form: **bc**; type: **`int`**; # of arguments: **`1`**)
  This integer property specifies the tpe of boundary conditions:
  - **Wall** boundary conditions (**BoundaryConditions**=0) imply that there is nothing outside the bounding box;
  - **Periodic** boundary conditions (**BoundaryConditions**=1) imply that opposite sides of the bounding box are identical, i.e. the value of `x=-0.1` is identical to `x=1.9` (the size of the bounding box is 2 in OpenGL units).

  Additional options for boundary conditions can be added in the future.
- **CellToPointMode** (short form: **cp**; type: **`int`**; # of arguments: **`1`**)
  A method for transforming the cell data to the point data (see Cell and Point Data). A value at a vertex of a uniform mesh can be obtained by taking 8 cell neighbors and computing an average (**CellToPointMode**=1), a median (**CellToPointMode**=2), a minimum (**CellToPointMode**=3), or a maximum value (**CellToPointMode**=4). All these methods cause the loss of information and

effectively smooth the input data. Alternatively, IFrIT can project the cell value to a point (**CellToPointMode**=5). In that case the data become displaced by half of the cell size. This is acceptable if you work with Scalar Field data only, or if you have Scalar, Vector and Tensor Field data of the same dimensions. However, if you use non−commensurate mesh sizes for Scalar, Vector and Tensor Field data, or if you use particles, then various data sets will be displaced relative to each other − which can bias the relative relation of different objects. The most accurate method is called the "Fourier shift" (**CellToPointMode**=0). It uses Fast Fourier Transform to project the cell data onto points. This method is lossless, and preserves location of the data, but it can be very slow for large mesh dimensions which are not powers of two. It also uses additional storage.

- **EraseData** (short form: **e**; type: **`int`**; # of arguments: **`1`**)
  An "action" (write−only) property that erases some of the loaded data from memory. It accepts one integer argument that specifies which data are to be erased:
  - 0 is for Uniform Scalars data.
  - 1 is for Uniform Vectors data.
  - 2 is for Uniform Tensors data.
  - 3 is for Basic Particles data.

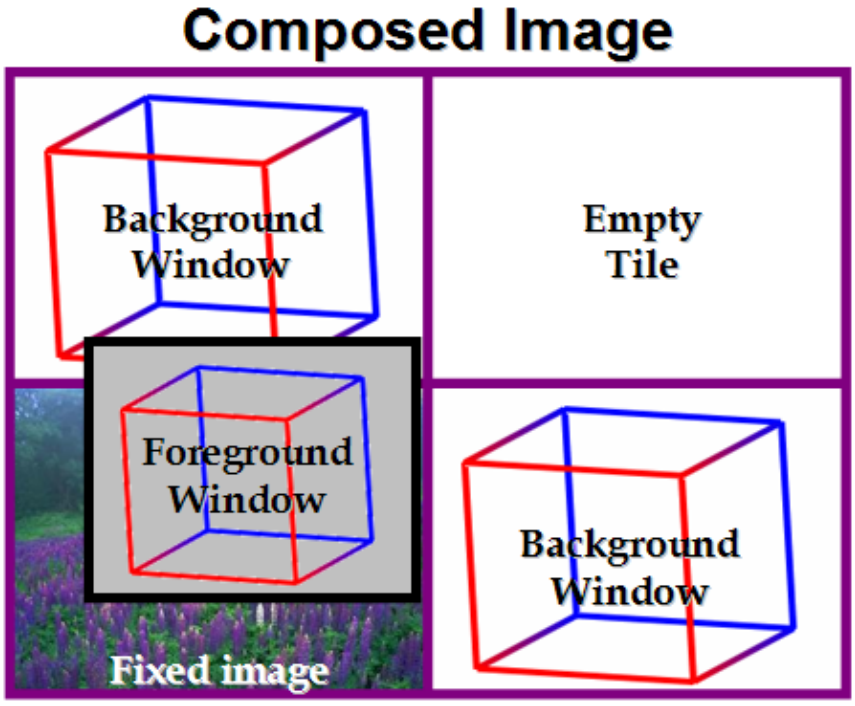
  For example, assigning a value of 2 to the argument of this property will erase Tensor Field data.
- **ErrorMessage** (short form: **em**; type: **`string`**; # of arguments: **`1`**)
  A string−values read−only property that returns the text of the error message if the error occured, or an empty string if the file was loaded successfully.
- **IsSet** (short form: **is**; type: **`bool`**; # of arguments: **`1`**)
  A boolean read−only property that returns 1 if the current data files form a set, and 0 if not.
- **LoadData** (short form: **l**; type: **`string`**; # of arguments: **`2`**)
  An "action" (write−only) property that loads a data file. The first argument of this property is one of the following strings:
  - **UniformScalars** (short form **us**) to load a Uniform Scalars file.
  - **UniformVectors** (short form **uv**) to load a Uniform Vectors file.
  - **UniformTensors** (short form **ut**) to load a Uniform Tensors file.
  - **BasicParticles** (short form **bp**) to load a Basic Particles file.

  The second argument is the name of the file. If it starts with the plus sign, the plus sign will be replaced with the name of the default data directory for this type of data file.
- **ParticlesDownsampleFactor** (short form: **pdf**; type: **`int`**; # of arguments: **`1`**)
  See ParticlesDownsampleMode.
- **ParticlesDownsampleMode** (short form: **pdm**; type: **`int`**; # of arguments: **`1`**)
  ParticleSetDownsample... properties specify which subset of all particles is to be loaded from a Particle Set file. The **ParticlesDownsampleFactor** is the inverse of the fraction of all particles which must be loaded (i.e. **ParticlesDownsampleFactor**=10 will load every tenth particle), while the **ParticlesDownsampleMode** property specifies the way the particles must be subsampled, and takes three values from 0 to 2:
  - **ParticlesDownsampleMode**=0 assumes that particle coordinates can be considered as a Vector Field on a uniform cubic mesh; then every **ParticlesDownsampleFactor**$^{1/3}$ particle in each dimension will be loaded.
  - **ParticlesDownsampleMode**=1 selects particles to load at random.
  - **ParticlesDownsampleMode**=2 selects the particles from the data file in order, until a needed number of particles is loaded. I.e., if the total number of particles in the file is `n`, then `n/`**`ParticlesDownsampleFactor`** first particles will be loaded.
- **ParticlesOrderInFileIsAttribute** (short form: **poa**; type: **`bool`**; # of arguments: **`1`**)
  A boolean switch that specifies whether the order of particles in the data file should be stored as an additional particle attribute. This can be useful for plotting, for example, trajectories, when individual particles in the data file are actually locations of the same object at different times.

- **ScalarFieldCalculatorFunction** (short form: **scf**; type: **`string`**; # of arguments: **`1`**)
  See ScalarFieldCalculatorOutputVariable.
- **ScalarFieldCalculatorOutputVariable** (short form: **scv**; type: **`int`**; # of arguments: **`1`**)
  ScalarFieldCalculator... properties control operations on Scalar Field variables. One of the Scalar Field variables (specified by the **ScalarFieldCalculatorOutputVariable** property) can be replaced by an arbitrary mathematical expression which may include all of the Scalar Field variables from the data file, and a Vector Field as well (if it loaded). The **ScalarFieldCalculatorFunction** property is a string that encapsulated the mathematical expression. It can use any of the mathematical functions understood by both IFrIT scripts. Scalar Field variables in the expression should be named "Var1", "Var2", etc, and the Vector Field should be named "Vector". For example, the string *`Var1+Var2*mag(Vector)`* is the expression that will multiply the magnitude of the Vector Field by the value of the second Scalar Field variable, and will add to the product the valur of the first Scalar Field variable. This calculation will be done **in every point on the mesh** (if the Vector Field is used in the expression, it must have the same dimensions as the Scalar Field).
- **ScaledDimension** (short form: **sd**; type: **`int`**; # of arguments: **`1`**)
  An integer value from −2 to 2, specifying which dimension (X, Y, or Z) of the uniform mesh data (a scalar, vector, or Tensor Field data) to fit into the bounding box. A value of −2 means the largest dimension, a valur of −1 means the shortest dimension, and a value between 0 and 2 means an X to Z dimension respectively. If the uniform mesh data is an exact cube, this property has no effect.
- **ShiftData** (short form: **s**; type: **`double`**; # of arguments: **`3`**)
  A three−dimensional floating point array that specifies the amounts (in units of the bounding box size = 2) by which the data should be shifted in each of 3 dimensions relative to their location in the data files. This is especially useful for periodic boundary conditions: it allows to bring to the center of the view a feature which, otherwise, may be cut into several peacies by the bounding box edges.
- **VoxelLocation** (short form: **vl**; type: **`int`**; # of arguments: **`1`**)
  A property specifying the location of the data on the uniform mesh. Value 0 means that the data are point data, i.e. they are located at the vertices of the mesh; value 1 implies cell data, which are located at the centers of mesh cells (see Cell and Point Data).

3.13 ImageComposer object

ImageComposer can combine several independent visualization windows into one output image. For example, it can tile several windows together in a regular mesh, or place a reduced representation of one visualization window as a insert inside another visualization window.



An example of an image composed from 3 different visualization windows

Images from different visualization windows are split into *background* and *foreground* images. The background images are tiles on a regular grid. Some of grid tiles may remain empty, or a fixed image from an external file can placed into an empty tile. The foreground images can be scaled down and placed anywhere in fron of the background grid of tiles. Both, background and foreground images may have borders.

If no background or foreground visualization windows are specified, then the **Image Composer** is considered to be inactive (or bypassed): it will simply create an image of whatever is shown in the current visualization windows.

Short form: **ic**

Available properties:

- **BackgroundWindowViewModule** (short form: **bgv**; type: **`int`**; # of arguments: **`any`**)
  An array with number of components equal to the number of background tiles. Each component of the array gives an id of the visualization window whose image goes into that tile, or –1, if that tile is empty.
- **BackgroundWindowViewModule2** (short form: **bgv2**; type: **`int`**; # of arguments: **`any`**)
  See BackgroundWindowViewModule3.
- **BackgroundWindowViewModule3** (short form: **bgv3**; type: **`int`**; # of arguments: **`any`**)
  Properties **BackgroundWindowViewModule2** and **BackgroundWindowViewModule3** specify additional visualization windows for a given tile for psedo–color composing. They are similar to **BackgroundWindowViewModule** property, but if more than one visualization window is specified for a given background tile, images from multiple windows are combined in pseudo–color single image as follows: each image is turned into a grayscale image, and then the grayscale image from the primary window is used to specify the red channel of the combined image, and grayscale images from windows 2 and 3 (if present) are used to provide the green and the blue channels. If no windows 2 and 3 are specified, then normal composing is used – the image from the primary window is displayed in

the respective tile as is.

- **BackgroundWindowWallpaperFile** (short form: **bgw**; type: **`string`**; # of arguments: **`any`**)
  An array with number of components equal to the number of background tiles. Each component of the array gives a name of the image file whose image goes into that tile, or an empty string, if no image should be placed in that tile. If both, the visualization window and the image file are specified for a tile, the visualization window takes precedence.
- **BorderColor** (short form: **bc**; type: **`color`**; # of arguments: **`1`**)
  A color of the border around background images. In properties colors are specified as 3–component RGB values `int.int.int` (like 255.0.0 for red).
- **BorderWidth** (short form: **bw**; type: **`int`**; # of arguments: **`1`**)
  A width (in pixels) of the border around background images.
- **ForegroundWindowBorderColor** (short form: **fgc**; type: **`color`**; # of arguments: **`any`**)
  An array with number of components equal to the number of foreground windows. Each component of the array gives the color of the border (as R.G.B) for the corresponding visualization window.
- **ForegroundWindowBorderWidth** (short form: **fgw**; type: **`int`**; # of arguments: **`any`**)
  An array with number of components equal to the number of foreground windows. Each component of the array gives the width of the border (in pixels) for the corresponding visualization window.
- **ForegroundWindowPositionX** (short form: **fgx**; type: **`int`**; # of arguments: **`any`**)
  See ForegroundWindowPositionY.
- **ForegroundWindowPositionY** (short form: **fgy**; type: **`int`**; # of arguments: **`any`**)
  ForegroundWindowPositionX and **ForegroundWindowPositionY** are two arrays with number of components equal to the number of foreground windows. Each component of the arrays gives the horizontal and vertical location of the lower left corner of the corresponding foreground window in the composed image.
- **ForegroundWindowScale** (short form: **fgs**; type: **`float`**; # of arguments: **`any`**)
  An array with number of components equal to the number of foreground windows. Each component of the array gives a floating point number less or eaual to 1, by which the corresponding visualization window is scaled.
- **ForegroundWindowViewModule** (short form: **fgv**; type: **`int`**; # of arguments: **`any`**)
  An array with number of components equal to the number of foreground windows. Each component of the array gives an id of the visualization window whose image goes into that tile. Each foreground image must have a visualization window associated with it.
- **ForegroundWindowViewModule2** (short form: **fgv2**; type: **`int`**; # of arguments: **`any`**)
  See ForegroundWindowViewModule3.
- **ForegroundWindowViewModule3** (short form: **fgv3**; type: **`int`**; # of arguments: **`any`**)
  Properties **ForegroundWindowViewModule2 and ForegroundWindowViewModule3 are used for psedo–color composing in a foreground window, completely analogous to BackgroundWindowViewModule2/3 properties.**
- **ImageHeight** (short form: **h**; type: **`int`**; # of arguments: **`1`**)
  See ImageWidth.
- **ImageWidth** (short form: **w**; type: **`int`**; # of arguments: **`1`**)
  ImageWidth and **ImageHeight** are read–only properties that give the total width and height of the composed image, including the border. The dimensions of the image are determined by the number of background tiles in both direction, by the size of each tile (which is taken to be the size of the largest visualization window set as a background tile), and the width of the border.
- **InnerBorder** (short form: **ib**; type: **`bool`**; # of arguments: **`1`**)
  A boolean switch specifying whether, if the background images have a border, this border should be present between the inner sides of background tiles, or only on the outside.
- **NumForegroundWindows** (short form: **nfg**; type: **`int`**; # of arguments: **`1`**)
  The number of foreground windows.

- **NumTiles** (short form: **nt**; type: **`int`**; # of arguments: **`2`**)
  The number of background tiles in two dimensions (stored as two components of this array).
- **ScaleBackground** (short form: **sb**; type: **`bool`**; # of arguments: **`1`**)
  A boolean switch specifying whether the background images, if they are smaller than the size of the background tile, should be scaled or padded to the tile size.

3.14 Marker object

Marker is a small resizable object that can be placed at a specified position inside the visualization scene. It can be used simply to mark a particular point, or other visualization objects can be "attached" to a marker. For example, markers can be used as staring points for streamlines of a Vector Field, or as centers of probing surfaces. A text caption – a body of text connected to the object by a line – can be attached to a marker to provide a short explanation. A legend listing all markers and their captions can also be placed at the bottom of the scene. A marker can be moved interactively using the ViewModule:PlaceMarker property. The marker caption can be moved interactively using ViewModule:MoveMarkerCaption property.

Short form: **m**

Available properties:

- **Ambient** (short form: **ma**; type: **`float`**; # of arguments: **`1`**)
  Properties **Ambient**, **Diffuse**, **Specular**, and **SpecularPower** control material properties of the object, as specified in OpenGL. The first three take a floating–point value between 0 and 1, while the **SpecularPower** property, which controls the OpenGL *shininess* property (it is called *specular power* in VTK), takes a floating–point value between 0 and 128. You need to know how OpenGL handles material properties to understand what these values mean.
- **CaptionPosition** (short form: **cx**; type: **`float`**; # of arguments: **`2`**)
  The position of the marker caption in the visualization window in viewport coordinates (in which the visualization window ranges from 0 to 1 in each direction).
- **CaptionText** (short form: **ct**; type: **`string`**; # of arguments: **`1`**)
  The text of the marker caption. The caption will be shown if this string is non–empty.
- **ClipPlane** (short form: **cp**; type: **`bool`**; # of arguments: **`1`**)
  A switch activating or deactivating the clipping plane. Takes a boolean (0/1) value.
- **Color** (short form: **c**; type: **`color`**; # of arguments: **`1`**)
  Properties **Color**, **Opacity**, and **Palette** set these three properties for **View Objects**, which are represented as solid surfaces (OpenGL polygonal mesh) – which are all View Objects except the **Volume** object. These three properties can be either scalar (1–component) or arrays (multi–component), depending on the type of the **View Objects** they belong to (check descriptions for specific objects for more details).The **Color** property takes a 3–component RGB value `int.int.int` (like 255.0.0 for red). The **Opacity** properties takes a floating point number between 0 and 1, and the **Palette** takes the integer value, that is interpreted as follows: if the value is positive, it is taken to be the index of the specific palette; if the value is negative, it is taken as minus the index of the palette, and the palette is reversed. For example, a value of 1 will select a traditional rainbow palette, while a value of –1 will select the rainbow palette but in the reversed order (pink is on the left, dark blue is on the right). Finally, a value of zero will select the special "brightness" palette, with which the object is colored by shades of its current color, with brightness of the shade related to the value of scalar variable used to color the object. The brightness palette cannot be reversed, and **Cross**

Section and **Volume** objects cannot be colored with a brightness palette (they do not have a regular color), so the value 0 is ignored for them.

- **ColorAutomatic** (short form: **ca**; type: **`bool`**; # of arguments: **`1`**)
  This property sets the color of the marker to be "automatic", i.e. to be the inverse of the scene background color. If the scene background color changes, so will the marker color. If this property is set to **false**, the color of the marker is specified by the Color property.
- **DataPresent** (short form: **dp**; type: **`bool`**; # of arguments: **`1`**)
  A read–only boolean property returning 1 if the object has the data and 0 otherwise.
- **DataType** (short form: **dt**; type: **`int`**; # of arguments: **`1`**)
  This property specifies the type of data a given object uses. It has no effect for the basic edition of IFrIT, but extensions to IFrIT use multiple data types and this property is used to select among different types of data.
- **Diffuse** (short form: **md**; type: **`float`**; # of arguments: **`1`**)
  See Ambient.
- **InteractiveMove** (short form: **im**; type: **`bool`**; # of arguments: **`1`**)
  This boolean property toggles whether a marker is moved interactively. During the interactive move, objects attached to a marker are updated immediately, while with the non–interactive move, attached objects are only updated when the move is done. Since updating objects may be computationally expensive, and interactive move may potentially be very slow.
- **Opacity** (short form: **o**; type: **`float`**; # of arguments: **`1`**)
  See Color.
- **Position** (short form: **x**; type: **`double`**; # of arguments: **`3`**)
  The position of the marker in the scene.
- **Ready** (short form: **r**; type: **`bool`**; # of arguments: **`1`**)
  A read–only boolean value for checking whether the object is ready to be shown (i.e. has the data and all the correct settigs). An attept to show an object that is not ready will not succeed.
- **Scaled** (short form: **sc**; type: **`bool`**; # of arguments: **`1`**)
  The boolean property that controls whether the marker is scaled during the scene interaction so that its aparent size remains the same, or behaves as all other objects – unscaled marker will become larger/smaller if the scene is zoomed in/out.
- **Size** (short form: **s**; type: **`double`**; # of arguments: **`1`**)
  The size of the marker.
- **Specular** (short form: **ms**; type: **`float`**; # of arguments: **`1`**)
  See Ambient.
- **SpecularPower** (short form: **mp**; type: **`float`**; # of arguments: **`1`**)
  See Ambient.
- **TextColor** (short form: **tc**; type: **`color`**; # of arguments: **`1`**)
  The color of the marker caption text.
- **Transform** (short form: **tf**; type: **`float`**; # of arguments: **`6`**)
  The first three arguments of this property are angles of rotation of this marker around X, Y, and Z axes (in degrees). The last three arguments are scaling factors of the marker along X, Y, and Z axes respectively. Scaling is done before rotations, so scaling axes are relative to the original marker orientation. Rotations are performed in the order Z, X, and then Y (like Euler angles).
- **Type** (short form: **t**; type: **`int`**; # of arguments: **`1`**)
  The type of the marker. The following values are available:
  - **0**: Point
  - **1**: Sphere
  - **2**: Tetrahydron
  - **3**: Cube
  - **4**: Octahedron
  - **5**: Icosahedron

- **6**: Dodecahedron
- **7**: Cone
- **8**: Cylinder
- **9**: Arrow
- **10**: Cluster (a cloud of points)
- **11**: Galaxy

- **Visible** (short form: **vis**; type: **`bool`**; # of arguments: **`1`**)
  A read–only boolean value for checking whether the object is visible. It has a value of **true** (1) is the **View Object** is visible, and **false** otherwise. This property cannot be used to show/hide the object, only to test the visibility state.

3.15 ParticleGroup object

Particle Group object represents a subset of particles that are visualized together: they share the same representation, coloring and sizing schemes, palette, etc.

Particles in one group can also be connected by lines (to represent trajectories). Two integer properties control how particles are connected. The property **AttributeToConnect** specifies the attribute that is used to connect particles. If this property is zero, no particles are connected. If it is non–zero, particles are connected in order of the increasing values of **AttributeToConnect**. In addition, the property **AttributeToSeparate** can be used to split a single line connecting all particles in the group into several separate lines. If the value of this property is positive, only particles with the same value of **AttributeToSeparate** are connected. For example, if the group contains 4 particles with 2 attributes: (1,1), (4,2), (2,1), and (3,2), then if **AttributeToConnect**=1, the particles are connected by a single line in the following order: 1,3,4,2. If, in addition, **AttributeToSeparate**=2, then two lines will be created, one connecting particles 1 and 3, and the second one connecting particles 4 and 2.

Short form: **pg**

Available properties:

- **Ambient** (short form: **ma**; type: **`float`**; # of arguments: **`1`**)
  Properties **Ambient**, **Diffuse**, **Specular**, and **SpecularPower** control material properties of the object, as specified in OpenGL. The first three take a floating–point value between 0 and 1, while the **SpecularPower** property, which controls the OpenGL *shininess* property (it is called *specular power* in VTK), takes a floating–point value between 0 and 128. You need to know how OpenGL handles material properties to understand what these values mean.
- **AttributeSizeDirect** (short form: **ad**; type: **`bool`**; # of arguments: **`1`**)
  This is a boolean property that specifies whether sizing of particles with the **AttributeToSize** property should be down directly (the value of the attribute = particle size), or via **SizeFunction**.
- **AttributeSizeExtraFactor** (short form: **ae**; type: **`float`**; # of arguments: **`1`**)
  If the particles are szed directly (**AttributeSizeDirect**=`true`), this property specifies an additional factor used to convert the particle attribute to size.
- **AttributeToColor** (short form: **ac**; type: **`int`**; # of arguments: **`1`**)
  Specifies the particle attribute used to color the particles with a palette. A value of zero means no coloring.

- **AttributeToConnect** (short form: **atc**; type: **`int`**; # of arguments: **`1`**)
  Particles in a group can be connected by lines. This property specifies the attribute that determines the order by which particles are connected. If this property is set to zero, no particles are connected.
- **AttributeToSeparate** (short form: **ats**; type: **`int`**; # of arguments: **`1`**)
  If particles are connected with lines, this property specifies the attribute (or none if it is set to 0) that is used to break a single line connecting all particles in a group into separate lines. If this attribute is non−zero, then only particles having the same value of **AttributeToSeparate** attribute are connected.
- **AttributeToSize** (short form: **as**; type: **`int`**; # of arguments: **`1`**)
  Specifies the particle attribute used to size the particle with. Depending on the value of the boolean attribute **AttributeSizeDirect**, particles can be either sized directly (`Size=Attribute*AttributeSizeExtraFactor`), or via a sizing function (`Size=Function(Attribute)`). `Particles can be sized differently only if they are not represented by points. Points cannot be sized differentially in OpenGL.`
- **ClipPlane** (short form: **cp**; type: **`bool`**; # of arguments: **`1`**)
  A switch activating or deactivating the clipping plane. Takes a boolean (0/1) value.
- **Color** (short form: **c**; type: **`color`**; # of arguments: **`1`**)
  Properties **Color**, **Opacity**, and **Palette** set these three properties for **View Objects**, which are represented as solid surfaces (OpenGL polygonal mesh) − which are all View Objects except the **Volume** object. These three properties can be either scalar (1−component) or arrays (multi−component), depending on the type of the **View Objects** they belong to (check descriptions for specific objects for more details).The **Color** property takes a 3−component RGB value `int.int.int` (like 255.0.0 for red). The **Opacity** properties takes a floating point number between 0 and 1, and the **Palette** takes the integer value, that is interpreted as follows: if the value is positive, it is taken to be the index of the specific palette; if the value is negative, it is taken as minus the index of the palette, and the palette is reversed. For example, a value of 1 will select a traditional rainbow palette, while a value of −1 will select the rainbow palette but in the reversed order (pink is on the left, dark blue is on the right). Finally, a value of zero will select the special "brightness" palette, with which the object is colored by shades of its current color, with brightness of the shade related to the value of scalar variable used to color the object. The brightness palette cannot be reversed, and **Cross Section** and **Volume** objects cannot be colored with a brightness palette (they do not have a regular color), so the value 0 is ignored for them.
- **DataPresent** (short form: **dp**; type: **`bool`**; # of arguments: **`1`**)
  A read−only boolean property returning 1 if the object has the data and 0 otherwise.
- **DataType** (short form: **dt**; type: **`int`**; # of arguments: **`1`**)
  This property specifies the type of data a given object uses. It has no effect for the basic edition of IFrIT, but extensions to IFrIT use multiple data types and this property is used to select among different types of data.
- **Diffuse** (short form: **md**; type: **`float`**; # of arguments: **`1`**)
  See Ambient.
- **FixedSize** (short form: **s**; type: **`float`**; # of arguments: **`1`**)
  The basic size of all particles. If particles are sized with a **Size Function**, all sizes are proportional to this value.
- **LineWidth** (short form: **lw**; type: **`int`**; # of arguments: **`1`**)
  The width of the line that connects particles, if they connected with lines
- **LowerLimitToColor** (short form: **ltc**; type: **`float`**; # of arguments: **`1`**)
  See StretchToColor.
- **LowerLimitToSize** (short form: **lts**; type: **`float`**; # of arguments: **`1`**)
  See StretchToSize.
- **NumReplicas** (short form: **nr**; type: **`int`**; # of arguments: **`6`**)
  Some of **View Objects** can be replicated, i.e. their identical replicas placed outside the bounding box.

This 6–dimensional integer property specifies how many replicas should be placed in –X, +X, –Y, +Y, –Z, and +Z directions, respectively. This property only has effect if the data are periodic in at least some of the directions, and if the object can be replicated (not all **View Objects**.

- **Opacity** (short form: **o**; type: **`float`**; # of arguments: **`1`**)
  See Color.
- **Palette** (short form: **p**; type: **`int`**; # of arguments: **`1`**)
  See Color.
- **Ready** (short form: **r**; type: **`bool`**; # of arguments: **`1`**)
  A read–only boolean value for checking whether the object is ready to be shown (i.e. has the data and all the correct settigs). An attept to show an object that is not ready will not succeed.
- **Shading** (short form: **mw**; type: **`bool`**; # of arguments: **`1`**)
  A boolean switch that can be used to switch off shading of light on the material (so that a surface looks the same no matter what its orientation relative to lights and a camera is).
- **SizeFunction** (short form: **sf**; type: any; # of arguments: **`1`**)
  This property sets the sizing function for differentially sizing particles depending on the value of their **AttributeToSize**. The sizing function is piece–wise linear function, and it is specified by a set of **`(X,Y)`** coordinates separated by a semicolon (;). For example, the following **ControlModule** request: **`SizeFunction`**`/0,0;0.5,1.0;1,0/` specifies a sizing function which stars at `(X,Y)=(0,0)`, goes linearly to `(X,Y)=(0.5,1.0)`, and then falls back to `(X,Y)=(1,0)`. The function is always defined for `X` and `Y` changing between 0 and 1, and the value of **AttributeToSize** are scaled to that range. For example, if the values of **AttributeToSize** are from –10 to 30, then the function described before will have a value of 0 for the attribute values of –10 and 30, and will be 1 for the attribute value of `-10+(30+10)/2=10`, so that particles with attribute values of –10 and 30 will be invisible, and particles with the attribute values of 10 will have their size equal to the value of **FixedSize** property, and particles with attribute values in between will be smaller.
- **Specular** (short form: **ms**; type: **`float`**; # of arguments: **`1`**)
  See Ambient.
- **SpecularPower** (short form: **mp**; type: **`float`**; # of arguments: **`1`**)
  See Ambient.
- **StretchToColor** (short form: **stc**; type: **`int`**; # of arguments: **`1`**)
  Properties **StretchToColor**, **LowerLimitToColor**, and **UpperLimitToColor** specify how the particles are colored with a **Palette**. The **StretchToColor** sets the stretch for coloring within the limits specified by **LowerLimitToColor** and **UpperLimitToColor**.
- **StretchToSize** (short form: **sts**; type: **`int`**; # of arguments: **`1`**)
  Properties **StretchToSize**, **LowerLimitToSize**, and **UpperLimitToSize** specify how the particles are sized with a **Size Function**. The **StretchToSize** sets the stretch for sizing within the limits specified by **LowerLimitToSize** and **UpperLimitToSize**. The later two properties are aliases of **Data–BasicParticles** **LowerLimit** and **UpperLimit** properties.
- **Type** (short form: **t**; type: **`int`**; # of arguments: **`1`**)
  This property specifies the representation of individual particles. The values of this property are the same as **Marker:Type**. Points are the fastest to show, but they cannot be sized differentially in OpenGL. If you need to size particles differentially, the fastest type to use is tetrahydron. Spheres are always the slowest type to visualize.
- **UpperLimitToColor** (short form: **utc**; type: **`float`**; # of arguments: **`1`**)
  See StretchToColor.
- **UpperLimitToSize** (short form: **uts**; type: **`float`**; # of arguments: **`1`**)
  See StretchToSize.
- **Visible** (short form: **vis**; type: **`bool`**; # of arguments: **`1`**)
  A read–only boolean value for checking whether the object is visible. It has a value of **true** (1) is the **View Object** is visible, and **false** otherwise. This property cannot be used to show/hide the object, only to test the visibility state.

3.16 Particles object

Particles object represents a collection of particles (points). The set of particles consists of one or more independent subsets called **Particle Group**s. Each **Particle Group** is controlled indepedently, particles within each group can be connected with lines to, for example, represent trajectories. Particles in different **Particle Group**s can have different representation (dots, spheres, etc), color, size, they can be colored or sized according to values of one of the attributes. One of the **Particle Group**s is always considered "current". The properties of a **Particle Group** can be accessed in two ways: either by using **Particle Group** properties for the current group, or using equivalent arrays−valued **Particles** properties with the appropriate arrays index. For example, if the **Particle Group** #2 is current, the **ParticleGroup:Type** property is equivalent to **Particles:Type[2]** property. To avoid referring to the specific **Particle Group** number, two special index values of the **Particles** array−valued property are understood: an empty index (**[]**) or index 0 (**[0]**) are understood as referring to the current **ParticleGroup**. Thus, the last example can also be expressed as **Particles:Type[]** or **Particles:Type[0]**. Of course, array−valued **Particles** properties can also be used to access other, non−current **ParticleGroup**s: **Particles:Type[1]** always refers to the point type of the **ParticleGroup** #1, no matter whether that group is current or not.

Short form: **p**

Available properties:

- **Ambient** (short form: **ma**; type: `float`; # of arguments: `any`)
  Properties **Ambient**, **Diffuse**, **Specular**, and **SpecularPower** control material properties of the object, as specified in OpenGL. The first three take a floating−point value between 0 and 1, while the **SpecularPower** property, which controls the OpenGL *shininess* property (it is called *specular power* in VTK), takes a floating−point value between 0 and 128. You need to know how OpenGL handles material properties to understand what these values mean.
- **AttributeSizeDirect** (short form: **ad**; type: `bool`; # of arguments: `any`)
  An array property that sets **AttributeSizeDirect** property for all **Particle Group**s.
- **AttributeSizeExtraFactor** (short form: **ae**; type: `float`; # of arguments: `any`)
  An array property that sets **AttributeSizeExtraFactor** property for all **Particle Group**s.
- **AttributeToColor** (short form: **ac**; type: `int`; # of arguments: `any`)
  An array property that sets **AttributeToColor** property for all **Particle Group**s.
- **AttributeToConnect** (short form: **atc**; type: `int`; # of arguments: `any`)
  An array property that sets **AttributeToConnect** property for all **Particle Group**s.
- **AttributeToSeparate** (short form: **ats**; type: `int`; # of arguments: `any`)
  An array property that sets **AttributeToSeparate** property for all **Particle Group**s.
- **AttributeToSize** (short form: **as**; type: `int`; # of arguments: `any`)
  An array property that sets **AttributeToSize** property for all **Particle Group**s.
- **AttributeToSplit** (short form: **a**; type: `int`; # of arguments: `1`)
  The particle attribute used in splitting **Particles** into separate **Particle Group**s. All particles within one **Particle Group** have the value of **AttributeToSplit** within the given range. Since ranges of different **Particle Group**s may overlap, a given particle may belong to more than one group.
- **ClipPlane** (short form: **cp**; type: `bool`; # of arguments: `1`)
  A switch activating or deactivating the clipping plane. Takes a boolean (0/1) value.

- **Color** (short form: **c**; type: **`color`**; # of arguments: **`any`**)
  An array property that sets **Color** property for all **Particle Group**s.
- **CurrentGroup** (short form: **cg**; type: **`int`**; # of arguments: **`1`**)
  This property sets the index of the current group. The index takes a value between 1 and the value of **MaxGroup** property.
- **DataPresent** (short form: **dp**; type: **`bool`**; # of arguments: **`1`**)
  A read–only boolean property returning 1 if the object has the data and 0 otherwise.
- **Diffuse** (short form: **md**; type: **`float`**; # of arguments: **`any`**)
  See Ambient.
- **FixedSize** (short form: **s**; type: **`float`**; # of arguments: **`any`**)
  An array property that sets **FixedSize** property for all **Particle Group**s.
- **LineWidth** (short form: **lw**; type: **`int`**; # of arguments: **`any`**)
  An array property that sets **LineWidth** property for all **Particle Group**s.
- **LowerLimitToColor** (short form: **ltc**; type: **`float`**; # of arguments: **`any`**)
  An array property that sets **LowerLimitToColor** property for all **Particle Group**s.
- **LowerLimitToSize** (short form: **lts**; type: **`float`**; # of arguments: **`any`**)
  An array property that sets **LowerLimitToSize** property for all **Particle Group**s.
- **MaxGroup** (short form: **mg**; type: **`int`**; # of arguments: **`1`**)
  The index of the last group (the same as the number of groups available). If the value of this property is changed, additional groups are created and extra groups are deleted automatically.
- **NumReplicas** (short form: **nr**; type: **`int`**; # of arguments: **`6`**)
  Some of **View Objects** can be replicated, i.e. their identical replicas placed outside the bounding box. This 6–dimensional integer property specifies how many replicas should be placed in –X, +X, –Y, +Y, –Z, and +Z directions, respectively. This property only has effect if the data are periodic in at least some of the directions, and if the object can be replicated (not all **View Objects**.
- **Opacity** (short form: **o**; type: **`float`**; # of arguments: **`any`**)
  An array property that sets **Opacity** property for all **Particle Group**s.
- **Palette** (short form: **p**; type: **`int`**; # of arguments: **`any`**)
  An array property that sets **Palette** property for all **Particle Group**s.
- **Ready** (short form: **r**; type: **`bool`**; # of arguments: **`1`**)
  A read–only boolean value for checking whether the object is ready to be shown (i.e. has the data and all the correct settigs). An attept to show an object that is not ready will not succeed.
- **Shading** (short form: **mw**; type: **`bool`**; # of arguments: **`any`**)
  A boolean switch that can be used to switch off shading of light on the material (so that a surface looks the same no matter what its orientation relative to lights and a camera is).
- **SizeFunction** (short form: **sf**; type: any; # of arguments: **`any`**)
  An array property that sets **SizeFunction** property for all **Particle Group**s.
- **Specular** (short form: **ms**; type: **`float`**; # of arguments: **`any`**)
  See Ambient.
- **SpecularPower** (short form: **mp**; type: **`float`**; # of arguments: **`any`**)
  See Ambient.
- **SplitRangeLowerLimit** (short form: **rl**; type: **`float`**; # of arguments: **`any`**)
  Arrays properties **SplitRangeLowerLimit** and **SplitRangeUpperLimit** set the range of the values of **AttributeToSplit** that belong to the particular group. For example, if **AttributeToSplit**=1, setting **SplitRangeLowerLimit[2]=0.5 and SplitRangeUpperLimit[2]=1.5 will lace all particles with the value of attribute #1 between 0.5 and 1.5 into group #2. These two properties can be thought of as boundaries of each Particle Group.**
- **SplitRangeUpperLimit** (short form: **ru**; type: **`float`**; # of arguments: **`any`**)
  See SplitRangeLowerLimit.
- **SplitRangesMax** (short form: **rmax**; type: **`float`**; # of arguments: **`1`**)
  Properties **SplitRangesMin** and **SplitRangesMax** specify the absolute minimum and maximum

between which the boundaries of separate **Particle Group**s are allowed to vary.

- **SplitRangesMin** (short form: **rmin**; type: **`float`**; # of arguments: **`1`**)
  See SplitRangesMax.
- **SplitRangesStretch** (short form: **rs**; type: **`int`**; # of arguments: **`1`**)
  This property specifies the stretch used for scaling attribute ranges. The values of range boundaries are always scaled linearly, and this property has a minor, mostly stylistic effect. For example, when a new range is created, the current group is split into two; with linear scaling the current range is split into two halves; with logarithmic scaling, the splitting value is taken to be the geometric mean of the two limits of the current range.
- **SplitRangesTiled** (short form: **rt**; type: **`bool`**; # of arguments: **`1`**)
  If this boolean property is set, the ranges of separate **Particle Group**s are always adjoint ("tiled"), i.e. **SplitRangeMin**[1] = **SplitRangeLowerLimit**[1], **SplitRangeUpperLimit**[1] = **SplitRangeLowerLimit**[2], **SplitRangeUpperLimit**[2] = **SplitRangeLowerLimit**[3], etc. For example, chaning the value of **SplitRangeUpperLimit**[2] property automatically changes the value of the **SplitRangeLowerLimit**[3] property to the same value. This is useful for making sure that every particle always belongs to exactly one group.
- **StretchToColor** (short form: **stc**; type: **`int`**; # of arguments: **`any`**)
  An array property that sets **StretchToColor** property for all **Particle Group**s.
- **StretchToSize** (short form: **sts**; type: **`int`**; # of arguments: **`any`**)
  An array property that sets **StretchToSize** property for all **Particle Group**s.
- **Type** (short form: **t**; type: **`int`**; # of arguments: **`any`**)
  An array property that sets **Type** property for all **Particle Group**s.
- **UpperLimitToColor** (short form: **utc**; type: **`float`**; # of arguments: **`any`**)
  An array property that sets **UpperLimitToColor** property for all **Particle Group**s.
- **UpperLimitToSize** (short form: **uts**; type: **`float`**; # of arguments: **`any`**)
  An array property that sets **UpperLimitToSize** property for all **Particle Group**s.
- **Visible** (short form: **vis**; type: **`bool`**; # of arguments: **`1`**)
  A read–only boolean value for checking whether the object is visible. It has a value of **true** (1) is the **View Object** is visible, and **false** otherwise. This property cannot be used to show/hide the object, only to test the visibility state.

3.17 Picker object

Picker allows the user to "pick" an object in the visualization scene, and obtain information about the data at the picked location. Picking is activated by pointing the cursor to the desired location and pressing a key `P` on the keyboard. IFrIT then follows an imaginary line drawn through the cursor position, and finds the **View Object** that the line intersects. The information about the location of the intersection and properties of the **View Object** is then reported.

Picker operates in one of three modes:

- **Cell** mode is the slowest, but it is most intuitive. In this mode Picker picks a "cell" that first intersects the line of sight. For an OpenGL polygonal mesh the cell is a polygon of the mesh, for volume rendering, the cell is a cell in the volume, for particles a cell is a single particle – in the latter case the size of the cell is zero, so it is often ambigious to pick, the **Point** mode is best for picking particles.
- **Point** mode is also slow, and is best suited for picking **Particles**. It is important to keep in mind that IFrIT only picks an **OpenGL** point, which may or may not have any relation to the underlying data

structure. For example, if IFrIT picks a point on the isosurface, the point most likely will not correspond to any vertex of the undelying grid. In addition, for a solid object (i.e. object consisting of the polygonal OpenGL mesh), only a vertex of a polygon can be picked. Thus, it is possible that you point the cursor on the solid surface, but nothing gets picked, because all polygon vertices are too far from the imaginary line. In that case, either use a **Cell** mode, or adjust the **Accuracy** property to reduce the maximum distance from the line−of−sight.
- **Object** mode is fast, because in this mode IFrIT uses the hardware to pick an object. However, the hardware picker may not pick translucent (non−opaque) objects correctly.

Short form: **pi**

Available properties:

- **Accuracy** (short form: **a**; type: `float`; # of arguments: `1`)
  This property specifies the distance (tolerance) within which the **Picker** will search for the nearest point or cell. If this distance is too small, Picker may not find even one point or cell to pick. If this distance is too large, the picked point may be too far from the cursor position, and may not be the point you intended to pick. Adjust this parameter to get the best results for you data. This property has no effect in the **Object** mode.
- **PickMethod** (short form: **m**; type: `int`; # of arguments: `1`)
  This property sets the picker mode: 0 is for **Cell** mode, 1 is for **Point** mode, and 2 is for **Object** mode.
- **PointSize** (short form: **ps**; type: `float`; # of arguments: `1`)
  The size of the point the Picker places at the picked position, in OpenGL coordinates (the length of the bounding box is 2).

3.18 Surface object

Surface object represents a two−dimensional surface that samples the three−dimensional scalar data. The surface can be either an isosurface of a particular scalar variable, or a specified geometric shape like a sphere or a plane. A surface can be painted by varied colors that correspond to a value of one of scalar variables via a specified palette. Multiple instances of a **Surface** object can co−exist at the same time − for example, several isosurfaces corresponding to different values of a scalar variable can be used to represent the three−dimensional structure of the data.

Because a surface has two sides, IFrIT uses the following rule to determine which side is called "outside" and which is called "inside" for an isosurface:

1. For the **first** variable the outside side is the side where the value of the variable is **larger** than the level of the isosurface.
2. For the **second** and the **third** variables the outside side is the side where the value of the variable is **smaller** than the level of the isosurface.

It might help you to put more meaning in the words "inside" and "outside" for your isosurface by appropriately ordering variables in the data file.

Common **Color**, **Opacity**, and **Palette** properties are 2−component arrays, with the first component refererring to the outside of the surface, and the second component referring to the inside of the surface.

Short form: **s**

Available properties:

- **AlternativeIsoSurfaceReductionMethod** (short form: **rda**; type: **`bool`**; # of arguments: **`1`**)
  If this property is set to 1, an alternative method for reducing the number of polygons in the isosurface will be used. Smetimes, the alternative method may be faster or may produce better results.
- **Ambient** (short form: **ma**; type: **`float`**; # of arguments: **`1`**)
  Properties **Ambient**, **Diffuse**, **Specular**, and **SpecularPower** control material properties of the object, as specified in OpenGL. The first three take a floating−point value between 0 and 1, while the **SpecularPower** property, which controls the OpenGL *shininess* property (it is called *specular power* in VTK), takes a floating−point value between 0 and 128. You need to know how OpenGL handles material properties to understand what these values mean.
- **AttachProbeToMarker** (short form: **am**; type: **`int`**; # of arguments: **`1`**)
  If this property is set to the value of the index of an existing **Marker**, then the center of the probe surface (i.e. not an isosurface) will be "attached" to a newly created **Marker**. If the **Marker** is moved, the surface will move with it too.
- **ClipPlane** (short form: **cp**; type: **`bool`**; # of arguments: **`1`**)
  A switch activating or deactivating the clipping plane. Takes a boolean (0/1) value.
- **Color** (short form: **c**; type: **`color`**; # of arguments: **`2`**)
  Properties **Color**, **Opacity**, and **Palette** set these three properties for **View Objects**, which are represented as solid surfaces (OpenGL polygonal mesh) − which are all View Objects except the **Volume** object. These three properties can be either scalar (1−component) or arrays (multi−component), depending on the type of the **View Objects** they belong to (check descriptions for specific objects for more details).The **Color** property takes a 3−component RGB value `int.int.int` (like 255.0.0 for red). The **Opacity** properties takes a floating point number between 0 and 1, and the **Palette** takes the integer value, that is interpreted as follows: if the value is positive, it is taken to be the index of the specific palette; if the value is negative, it is taken as minus the index of the palette, and the palette is reversed. For example, a value of 1 will select a traditional rainbow palette, while a value of −1 will select the rainbow palette but in the reversed order (pink is on the left, dark blue is on the right). Finally, a value of zero will select the special "brightness" palette, with which the object is colored by shades of its current color, with brightness of the shade related to the value of scalar variable used to color the object. The brightness palette cannot be reversed, and **Cross Section** and **Volume** objects cannot be colored with a brightness palette (they do not have a regular color), so the value 0 is ignored for them.
- **DataPresent** (short form: **dp**; type: **`bool`**; # of arguments: **`1`**)
  A read−only boolean property returning 1 if the object has the data and 0 otherwise.
- **DataType** (short form: **dt**; type: **`int`**; # of arguments: **`1`**)
  This property specifies the type of data a given object uses. It has no effect for the basic edition of IFrIT, but extensions to IFrIT use multiple data types and this property is used to select among different types of data.
- **Diffuse** (short form: **md**; type: **`float`**; # of arguments: **`1`**)
  See Ambient.
- **IsoSurfaceLevel** (short form: **l**; type: **`float`**; # of arguments: **`1`**)
  Sets the value for the isosurface level in the isosurface mode.
- **IsoSurfaceMethod** (short form: **im**; type: **`int`**; # of arguments: **`1`**)
  Selects the method for creating the isosurface. The **IsoSurfaceMethod**=1 selects a commonly used Marching Cubes method, but it may not be available in all versions of VTK. A value of **IsoSurfaceMethod**=0 selects an alternative method, which may vary from one VTK version to another. Method **0** should *always* work, while method **1** may not always work due to VTK bugs. IFrIT attempts to choose the best method as a default setting.

- **IsoSurfaceOptimization** (short form: **op**; type: **`bool`**; # of arguments: **`1`**)
  A boolean property that toggles whether the isosurface is optimized after construction. Optimizing the isosurface takes time, but the optimized isosurface will render faster. Setting this property to `true` only makes sense if you plan to send long time working with a given isosurface.
- **IsoSurfaceReduction** (short form: **rd**; type: **`int`**; # of arguments: **`1`**)
  Sets the level of reduction in the number of polygons that represent the isosurface. This property takes values from 0 to 3: value 0 means no reduction, 1 attempts to reduce by 75%, 2 by 90%, and 3 by 99%. Reduction does not change the topology of the isosurface (at least, should not), so the target level may not be achievable in practice.
- **IsoSurfaceSmoothing** (short form: **sm**; type: **`int`**; # of arguments: **`1`**)
  The integer factor from 0 to 10 that controls the degree of additional smoothing for the isosurface. The value of zero switches smoothing off. Smoothing an isosurface takes time, but makes it look better.
- **IsoSurfaceVar** (short form: **v**; type: **`int`**; # of arguments: **`1`**)
  The scalar variable whose isosurface is created.
- **Method** (short form: **m**; type: **`int`**; # of arguments: **`1`**)
  Specifies the method for creating the surface. Three methods are currently supported:
  - **0**: isosurface of the scalar variable **IsoSurfaceVar** at the value **IsoSurfaceLevel**,
  - **1**: sphere of radius **Size** at the position **Position**,
  - **2**: plane with normal **PlaneDirection** at the position **Position**.
- **NormalsFlipped** (short form: **nf**; type: **`bool`**; # of arguments: **`1`**)
  This boolean property, if set to `true`, reverses the direction of normals (i.e. the sense of inside–outside for the surface) relative to the default settings.
- **NumReplicas** (short form: **nr**; type: **`int`**; # of arguments: **`6`**)
  Some of **View Objects** can be replicated, i.e. their identical replicas placed outside the bounding box. This 6–dimensional integer property specifies how many replicas should be placed in –X, +X, –Y, +Y, –Z, and +Z directions, respectively. This property only has effect if the data are periodic in at least some of the directions, and if the object can be replicated (not all **View Objects**.
- **Opacity** (short form: **o**; type: **`float`**; # of arguments: **`2`**)
  See Color.
- **PaintVar** (short form: **pv**; type: **`int`**; # of arguments: **`2`**)
  This two–component integer array sets the variable to paint the outside (first component) or the inside (the second component) of the surface with. The two–component **Palette** property sets the respective palettes
- **Palette** (short form: **p**; type: **`int`**; # of arguments: **`2`**)
  See Color.
- **PlaneDirection** (short form: **pd**; type: **`float`**; # of arguments: **`3`**)
  The direction (specified as a 3–component double array) of the surface in plane mode.
- **Position** (short form: **x**; type: **`double`**; # of arguments: **`3`**)
  The position of the fixed probe surface (plane or sphere).
- **ProbePositionToPickedPoint** (short form: **p2p**; type: **`bool`**; # of arguments: **`1`**)
  This is an "action" (write–only) property that moves the location of the fixed probe surface (sphere or plane) to the currently picked position. If the surface is attached to a marker, the attachment is broken.
- **Ready** (short form: **r**; type: **`bool`**; # of arguments: **`1`**)
  A read–only boolean value for checking whether the object is ready to be shown (i.e. has the data and all the correct settigs). An attept to show an object that is not ready will not succeed.
- **Size** (short form: **s**; type: **`double`**; # of arguments: **`1`**)
  The radius of the sphere in the sphere mode.
- **Specular** (short form: **ms**; type: **`float`**; # of arguments: **`1`**)
  See Ambient.

- **SpecularPower** (short form: **mp**; type: **`float`**; # of arguments: **`1`**)
  See Ambient.
- **Visible** (short form: **vis**; type: **`bool`**; # of arguments: **`1`**)
  A read–only boolean value for checking whether the object is visible. It has a value of **true** (1) is the **View Object** is visible, and **false** otherwise. This property cannot be used to show/hide the object, only to test the visibility state.

3.19 Tensor object

TensorField object represents a Tensor Field. Currently, the only supported method for Tensor Field visualization is a "tensor glyph": a collection of ellipsods, which, at every point is oriented along the eigenvectors of the tensor and (optionally) scaled in proportion to tensor eigenvalues.

Short form: **t**

Available properties:

- **Ambient** (short form: **ma**; type: **`float`**; # of arguments: **`1`**)
  Properties **Ambient**, **Diffuse**, **Specular**, and **SpecularPower** control material properties of the object, as specified in OpenGL. The first three take a floating–point value between 0 and 1, while the **SpecularPower** property, which controls the OpenGL *shininess* property (it is called *specular power* in VTK), takes a floating–point value between 0 and 128. You need to know how OpenGL handles material properties to understand what these values mean.
- **ClipPlane** (short form: **cp**; type: **`bool`**; # of arguments: **`1`**)
  A switch activating or deactivating the clipping plane. Takes a boolean (0/1) value.
- **Color** (short form: **c**; type: **`color`**; # of arguments: **`1`**)
  Properties **Color**, **Opacity**, and **Palette** set these three properties for **View Objects**, which are represented as solid surfaces (OpenGL polygonal mesh) – which are all View Objects except the **Volume** object. These three properties can be either scalar (1–component) or arrays (multi–component), depending on the type of the **View Objects** they belong to (check descriptions for specific objects for more details).The **Color** property takes a 3–component RGB value `int.int.int` (like 255.0.0 for red). The **Opacity** properties takes a floating point number between 0 and 1, and the **Palette** takes the integer value, that is interpreted as follows: if the value is positive, it is taken to be the index of the specific palette; if the value is negative, it is taken as minus the index of the palette, and the palette is reversed. For example, a value of 1 will select a traditional rainbow palette, while a value of –1 will select the rainbow palette but in the reversed order (pink is on the left, dark blue is on the right). Finally, a value of zero will select the special "brightness" palette, with which the object is colored by shades of its current color, with brightness of the shade related to the value of scalar variable used to color the object. The brightness palette cannot be reversed, and **Cross Section** and **Volume** objects cannot be colored with a brightness palette (they do not have a regular color), so the value 0 is ignored for them.
- **ColorBy** (short form: **cb**; type: **`int`**; # of arguments: **`1`**)
  The integer property specifying the scalar variable which is used to color the object. If this property is set to 0, no coloring by scalar variable is performed (all glyphs have the same color). The **Vector Field** object accepts, in addition, the following 3 values:
  - **–3**: color by the vector field magnitude;
  - **–2**: color by the vector field vorticity;

- **−1**: color by the divergence of the vector field.

- **ConnectedToScalars** (short form: **cs**; type: **`bool`**; # of arguments: **`1`**)
  A read−only boolean property specifying whether the object can be colored by a scalar variables. If this property is `false`, setting **ColorBy** property to a positive value will have no effect.
- **DataPresent** (short form: **dp**; type: **`bool`**; # of arguments: **`1`**)
  A read−only boolean property returning 1 if the object has the data and 0 otherwise.
- **DataType** (short form: **dt**; type: **`int`**; # of arguments: **`1`**)
  This property specifies the type of data a given object uses. It has no effect for the basic edition of IFrIT, but extensions to IFrIT use multiple data types and this property is used to select among different types of data.
- **Diffuse** (short form: **md**; type: **`float`**; # of arguments: **`1`**)
  See Ambient.
- **GlyphSampleRate** (short form: **gr**; type: **`int`**; # of arguments: **`1`**)
  This property accepts integer positive numbers that specify the sampling rate for the object. If this property is set to 1, every point in the data will be shown. For values larger than 1, only every **GlyphSampleRate** value in every direction will be shown.
- **GlyphSize** (short form: **gs**; type: **`double`**; # of arguments: **`1`**)
  The uniform size to scale all glyphs with.
- **Method** (short form: **m**; type: **`int`**; # of arguments: **`1`**)
  This property is reserved for future use. Currently, only one methd − tensor glyph − is supported.
- **NumReplicas** (short form: **nr**; type: **`int`**; # of arguments: **`6`**)
  Some of **View Objects** can be replicated, i.e. their identical replicas placed outside the bounding box. This 6−dimensional integer property specifies how many replicas should be placed in −X, +X, −Y, +Y, −Z, and +Z directions, respectively. This property only has effect if the data are periodic in at least some of the directions, and if the object can be replicated (not all **View Objects**.
- **Opacity** (short form: **o**; type: **`float`**; # of arguments: **`1`**)
  See Color.
- **Palette** (short form: **p**; type: **`int`**; # of arguments: **`1`**)
  See Color.
- **Ready** (short form: **r**; type: **`bool`**; # of arguments: **`1`**)
  A read−only boolean value for checking whether the object is ready to be shown (i.e. has the data and all the correct settigs). An attept to show an object that is not ready will not succeed.
- **ScalingOn** (short form: **so**; type: **`bool`**; # of arguments: **`1`**)
  The boolean property that specifies whether the tensor glyphs should be scaled by the tensor eigenvalues. If it is `false`, all glyphs will have the same volume, but different shapes and orientations.
- **Specular** (short form: **ms**; type: **`float`**; # of arguments: **`1`**)
  See Ambient.
- **SpecularPower** (short form: **mp**; type: **`float`**; # of arguments: **`1`**)
  See Ambient.
- **Visible** (short form: **vis**; type: **`bool`**; # of arguments: **`1`**)
  A read−only boolean value for checking whether the object is visible. It has a value of **true** (1) is the **View Object** is visible, and **false** otherwise. This property cannot be used to show/hide the object, only to test the visibility state.

3.20 Vector object

VectorField object represents a vector field, like the flow of the fluid. Vector field can be represented as a "vector glyph" – a set of straight lines pointing in the direction of the Vector Field at every point with lengths proportional to the magnitude of the vector, or as a collection of stream lines – lines which would correspond to the fluid flow lines if the Vector Field was a real fluid velocity field. Streamlines originate at a **source object**, which can be a plane, a sphere, a disk, or a all existing **Marker**s. Common **Color**, **Opacity**, and **Palette** properties are 2–component arrays, with the first component refererring to the VectorField itself, and the second component referring to the streamline source object.

Short form: **v**

Available properties:

- **Ambient** (short form: **ma**; type: **`float`**; # of arguments: **`1`**)
  Properties **Ambient**, **Diffuse**, **Specular**, and **SpecularPower** control material properties of the object, as specified in OpenGL. The first three take a floating–point value between 0 and 1, while the **SpecularPower** property, which controls the OpenGL *shininess* property (it is called *specular power* in VTK), takes a floating–point value between 0 and 128. You need to know how OpenGL handles material properties to understand what these values mean.
- **AttachSourceToMarker** (short form: **am**; type: **`int`**; # of arguments: **`1`**)
  If this property is set to the value of the index of an existing **Marker**, then the center of the streamline source object will be "attached" to a newly created **Marker**. If the **Marker** is moved, the source object will move with it too.
- **ClipPlane** (short form: **cp**; type: **`bool`**; # of arguments: **`1`**)
  A switch activating or deactivating the clipping plane. Takes a boolean (0/1) value.
- **Color** (short form: **c**; type: **`color`**; # of arguments: **`2`**)
  Properties **Color**, **Opacity**, and **Palette** set these three properties for **View Objects**, which are represented as solid surfaces (OpenGL polygonal mesh) – which are all View Objects except the **Volume** object. These three properties can be either scalar (1–component) or arrays (multi–component), depending on the type of the **View Objects** they belong to (check descriptions for specific objects for more details).The **Color** property takes a 3–component RGB value `int.int.int` (like 255.0.0 for red). The **Opacity** properties takes a floating point number between 0 and 1, and the **Palette** takes the integer value, that is interpreted as follows: if the value is positive, it is taken to be the index of the specific palette; if the value is negative, it is taken as minus the index of the palette, and the palette is reversed. For example, a value of 1 will select a traditional rainbow palette, while a value of –1 will select the rainbow palette but in the reversed order (pink is on the left, dark blue is on the right). Finally, a value of zero will select the special "brightness" palette, with which the object is colored by shades of its current color, with brightness of the shade related to the value of scalar variable used to color the object. The brightness palette cannot be reversed, and **Cross Section** and **Volume** objects cannot be colored with a brightness palette (they do not have a regular color), so the value 0 is ignored for them.
- **ColorBy** (short form: **cb**; type: **`int`**; # of arguments: **`1`**)
  The integer property specifying the scalar variable which is used to color the object. If this property is set to 0, no coloring by scalar variable is performed (all glyphs have the same color). The **Vector Field** object accepts, in addition, the following 3 values:
  - **–3**: color by the vector field magnitude;
  - **–2**: color by the vector field vorticity;
  - **–1**: color by the divergence of the vector field.
- **ConnectedToScalars** (short form: **cs**; type: **`bool`**; # of arguments: **`1`**)
  A read–only boolean property specifying whether the object can be colored by a scalar variables. If this property is `false`, setting **ColorBy** property to a positive value will have no effect.

- **DataPresent** (short form: **dp**; type: **`bool`**; # of arguments: **`1`**)
  A read–only boolean property returning 1 if the object has the data and 0 otherwise.
- **DataType** (short form: **dt**; type: **`int`**; # of arguments: **`1`**)
  This property specifies the type of data a given object uses. It has no effect for the basic edition of IFrIT, but extensions to IFrIT use multiple data types and this property is used to select among different types of data.
- **Diffuse** (short form: **md**; type: **`float`**; # of arguments: **`1`**)
  See Ambient.
- **GlyphSampleRate** (short form: **gr**; type: **`int`**; # of arguments: **`1`**)
  This property accepts integer positive numbers that specify the sampling rate for the object. If this property is set to 1, every point in the data will be shown. For values larger than 1, only every **GlyphSampleRate** value in every direction will be shown.
- **GlyphSize** (short form: **gs**; type: **`double`**; # of arguments: **`1`**)
  The uniform size to scale all glyphs with.
- **LineDir** (short form: **ld**; type: **`int`**; # of arguments: **`1`**)
  The direction of the streamlines relative to the source. The following values are accepted:
  - **0**: the up–stream direction (the direction the **Vector Field** is point to);
  - **1**: the down–stream direction (the direction opposite to the one the **Vector Field** is point to);
  - **2**: the forward direction (the direction of increasing coordinate);
  - **3**: the backward direction (the direction of decreasing coordinate);
  - **4**: both directions (the streamline is drawn from both sides of the source object).
- **LineLength** (short form: **ll**; type: **`float`**; # of arguments: **`1`**)
  The maximum length for the streamline to draw.
- **LineQuality** (short form: **lq**; type: **`int`**; # of arguments: **`1`**)
  The quality of the streamline. If the value of this property is zero, the streamline may appear "jaggy". Positive values of this property make the streamline smoother, at the expense of longer rendering time.
- **Method** (short form: **m**; type: **`int`**; # of arguments: **`1`**)
  The following modes for visualizing the vector field are supported:
  - **Glyph (0)**: a set of straight lines pointing in the direction of the **Vector Field** at every point with lengths proportional to the magnitude of the vector;
  - **Stream line (1)**: a set of lines, which would correspond to the fluid flow lines if the **Vector Field** was a real fluid velocity field;
  - **Stream tube (2)**: a stream line that has a finite thickness that varies according to the flow speed (as if the mass flow is conserved);
  - **Stream band (3)**: a pair of nearby stream lines with the surface between them. This is useful to visualize diverging or converging flows.

  In the last three modes, streamlines are
- **NumReplicas** (short form: **nr**; type: **`int`**; # of arguments: **`6`**)
  Some of **View Objects** can be replicated, i.e. their identical replicas placed outside the bounding box. This 6–dimensional integer property specifies how many replicas should be placed in –X, +X, –Y, +Y, –Z, and +Z directions, respectively. This property only has effect if the data are periodic in at least some of the directions, and if the object can be replicated (not all **View Objects**.
- **NumberOfStreamLines** (short form: **nl**; type: **`int`**; # of arguments: **`1`**)
  The number of streamlines to generate. If the source object is **Markers**, then this property has no effect, as the number of streamlines is equal to the number of available markers.
- **Opacity** (short form: **o**; type: **`float`**; # of arguments: **`2`**)
  See Color.
- **Palette** (short form: **p**; type: **`int`**; # of arguments: **`2`**)
  See Color.

- **Ready** (short form: **r**; type: **`bool`**; # of arguments: **`1`**)
  A read–only boolean value for checking whether the object is ready to be shown (i.e. has the data and all the correct settigs). An attept to show an object that is not ready will not succeed.
- **ShowSourceObject** (short form: **sso**; type: **`bool`**; # of arguments: **`1`**)
  A boolean property that toggles whether the source object is shown in the visualization scene.
- **SourceDirection** (short form: **sd**; type: **`float`**; # of arguments: **`3`**)
  The direction (specified as a 3–component double array) of the source object when it is a plane or a disk.
- **SourceOpacity** (short form: **so**; type: **`float`**; # of arguments: **`1`**)
  The opacity of the source object (between 0 and 1), if it is shown.
- **SourcePosition** (short form: **sx**; type: **`double`**; # of arguments: **`3`**)
  The position of the source object.
- **SourcePositionToPickedPoint** (short form: **p2p**; type: **`bool`**; # of arguments: **`1`**)
  This "action" (write–only) property, when assigned a `true` value, moves the center of the source object to the currently picked position.
- **SourceSize** (short form: **ss**; type: **`double`**; # of arguments: **`1`**)
  The size of the source if it is a disk or a sphere.
- **SourceType** (short form: **st**; type: **`int`**; # of arguments: **`1`**)
  The type of a geometric shape of the source object:
  - **0**: disk;
  - **1**: plane;
  - **2**: size.

  The size of the disk or the sphere is set by the **SourceSize** property, the orientation of the disk or the plane is set by the **SourceDirection** property, and the center of the source object is set by the **SourcePosition** propery.
- **Specular** (short form: **ms**; type: **`float`**; # of arguments: **`1`**)
  See Ambient.
- **SpecularPower** (short form: **mp**; type: **`float`**; # of arguments: **`1`**)
  See Ambient.
- **TubeRangeFactor** (short form: **tr**; type: **`float`**; # of arguments: **`1`**)
  TubeSize, **TubeRangeFactor**, and **TubeVariationFactor** control the shape of streamtubes. The **TubeSize** property sets the minimum diameter of the tube; all linear scales in the tube are proportional to the value of this property. The **TubeRangeFactor** property is the ratio of the maximum to the minimum sizes of the tube. The **TubeVariationFactor** determines how sensitive the size of the tube is to the value of the vector field visualized. It may take some experimentation with these three parameters to achieve a satisfactory–looking tubes for your data.
- **TubeSize** (short form: **ts**; type: **`int`**; # of arguments: **`1`**)
  See TubeRangeFactor.
- **TubeVariationFactor** (short form: **tv**; type: **`float`**; # of arguments: **`1`**)
  See TubeRangeFactor.
- **Visible** (short form: **vis**; type: **`bool`**; # of arguments: **`1`**)
  A read–only boolean value for checking whether the object is visible. It has a value of **true** (1) is the **View Object** is visible, and **false** otherwise. This property cannot be used to show/hide the object, only to test the visibility state.

3.21 ViewModule object

The **View Module** object is a single visualization window that displays the whole scene. Several **ViewModule** objects can exist, and can either be independent, or be "clones" of another **ViewModule**. In the latter case they share the data with the "parent" window but in all other respects behave as separate windows. All other objects except the **ControlModule** object and the **ImageComposer** object belong to one of **ViewModule** objects.The **View Module** object is directly responsible for showing the **Bounding Box**, **Record Label**, **Clipping Plane**, and several other controls, but it delegates visualization of the data to its **View Objects**.

Short form: **vm**

Available properties:

- **AnimationOutput** (short form: **ao**; type: **`int`**; # of arguments: **`1`**)
  The boolean property that controls the final output of an animation. If it is set to 0, the animation is stored in a sequence of files; for other values the animation is saved into a movie file: value of 1 creates an MPEG2 movie, and a value of 2 creates an AVI movie (the latter option requires special libraries and may not be available on all platforms). The movie option is only available under VTK 5.
- **Antialiasing** (short form: **a**; type: **`bool`**; # of arguments: **`1`**)
  The boolean property that toggles OpenGL antialiasing in the scene.
- **AxesBoxLabels** (short form: **abl**; type: **`string`**; # of arguments: **`3`**)
  When the **Bounding Box** is displayed as axes, properties **AxesBoxLabels** and **AxesBoxRanges** specify labels and ranges for the three axes respectively. The former property takes 3 string–values arguments, while the latter takes 6 floating point numbers as `Xmin`, `Xmax`, `Ymin`, etc.
- **AxesBoxRanges** (short form: **abr**; type: **`float`**; # of arguments: **`6`**)
  See AxesBoxLabels.
- **BackgroundColor** (short form: **bg**; type: **`color`**; # of arguments: **`1`**)
  Sets the background color of the scene, specified as a 3–component RGB value `int.int.int` (like 255.255.255 for white).
- **BoundingBox** (short form: **bb**; type: **`bool`**; # of arguments: **`1`**)
  The boolean property that toggles showing of the bounding box – the frame that encloses (but not truncates) the main visualizatin scene. The bounding box serves as a useful reference frames for placing other objects in relation to each other.
- **BoundingBoxType** (short form: **bbt**; type: **`int`**; # of arguments: **`1`**)
  The type of the **Bounding Box**, as follows:
  - **0**: the default IFrIT–style bounding box (red–blended–into–blue);
  - **1**: the classic bounding box (red in X–direction, green in Y, blue in Z);
  - **2**: the hair–thin bounding box (one–pixel wide lines);
  - **3**: the classic bounding box with X, Y, and Z axes shown as coordinate axes, with arrows at the end, labels, and ranges. This type is useful for showing 3D scatter plots.
- **BoxSize** (short form: **bs**; type: **`float`**; # of arguments: **`1`**)
  This property sets the size of the **Bounding Box** in physical units. Chaging this property does not actually change the visualization scene and it does not affect the relation of objects with respect to each other, but it useful for assigning physical meaning to the distances in the scene. This property is only meaningful if the **OpenGL Coordinates** property is set to `false`. Otherwise, the **Box Size** is always 2 and coordinates go from –1 to 1.
- **CameraAlignmentLabel** (short form: **ca**; type: **`bool`**; # of arguments: **`1`**)
  Toggles showing of coordinate axis when the came orientation is orthogonal, i.e. the direction of viewing and the view up vector are both parallel to coordinate axes. This is useful when visualizing **Cross Section** objects.

- **ClipPlane** (short form: **cp**; type: **`bool`**; # of arguments: **`1`**)
  Toggles showing of the clipping plane. The clipping plane cuts off a portion of the scene, thus allowing to look inside a visuaized object. It is controled by the **ClipPlaneDirection** and **ClipPlaneDistance** properties.
- **ClipPlaneDirection** (short form: **cpd**; type: **`float`**; # of arguments: **`3`**)
  ClipPlaneDirection and **ClipPlaneDistance** properties control the orientation and location of the . The **ClipPlaneDirection** property specifies the direction of the normal to the , while the The **ClipPlaneDistance** property sets the distance from the plane to the center of the bounding box in OpenGL coordinates.
- **ClipPlaneDistance** (short form: **cpl**; type: **`float`**; # of arguments: **`1`**)
  See ClipPlaneDirection.
- **CloneOfWindow** (short form: **co**; type: **`int`**; # of arguments: **`1`**)
  A read–only property that returns `true` if this **View Module** is a clone of another one, and `false` if it owns its data.
- **ColorBars** (short form: **cb**; type: **`bool`**; # of arguments: **`1`**)
  Toggles showing of **Color Bar**s.
- **CrossSectionCurrent** (short form: **oxc**; type: **`int`**; # of arguments: **`1`**)
  This property specifies the index of the current cross section. Must take value between 1 and the value of Cross Section Max property.
- **CrossSectionMax** (short form: **oxm**; type: **`int`**; # of arguments: **`1`**)
  The read–only property that returns the number of existing cross sections.
- **DumpImage** (short form: **di**; type: **`bool`**; # of arguments: **`1`**)
  An "action" (write–only) property, setting `true` to which creates the image file with the current visualization scene (see **Image Composer** for more details what can be output as an image).
- **FontScale** (short form: **fs**; type: **`int`**; # of arguments: **`1`**)
  FontScale and **FontType** properties set the font size and type of text displayed along **Color Bar**s, a **Ruler**, etc. The scale value of 0 means the default font size, negative values make the font smaller, positive values make it larger.
- **FontType** (short form: **ft**; type: **`int`**; # of arguments: **`1`**)
  See FontScale.
- **GlassClipPlane** (short form: **gcp**; type: **`bool`**; # of arguments: **`1`**)
  Toggles whether a highly transparent (a–la glass) plane showing the location and orientation of the is shown in the scene. It is usually helpful in placing the in exact location.
- **ImageFormat** (short form: **if**; type: **`int`**; # of arguments: **`1`**)
  The format of the image file produces. The accepted values are:
  - **0**: Portable Network Grapics(.png);
  - **1**: Joint Photographic Experts Group (.jpg);
  - **2**: Portable pixmap (.ppm);
  - **3**: Windows bitmap (.bmp);
  - **4**: Tag Image File Format (.tif);
  - **5**: Encapsulated Postrscript (.eps).

  Frankly speaking, I have no idea why anyone would want all of them...
- **ImageMagnification** (short form: **ix**; type: **`int`**; # of arguments: **`1`**)
  The magnification factor of the image. If the produces image does not have to be the same as the image on the screen, but can be any factor larger. I have a 30,000 by 30,000 image on the wall of my office.
- **InteractorStyle** (short form: **is**; type: **`int`**; # of arguments: **`1`**)
  The style of the visualization window *interactor*, i.e. the component that control how mouse clicks and keyboard keys are affect the visualization scene. IfrIT used four different interactor styles:
- **0** or **1**: use **Display** mode described in Mouse and Keyboard Controls. The difference between values 0 and 1 is that the value 0 uses a *trackball* method of interaction, when you need to click on a mouse

button and move the mouse to change the scene; the value of 1 uses a *joystick* mode, when just clicking on mouse button causes the scene to spin or zoom (depending on the button clicked) with a rate that depends on how far away the mouse cursor is from the center of the screen;

- **2**: use **Fly−by** mode described in Mouse and Keyboard Controls.
- **3**: use **Keyboard Interactor** mode described in Mouse and Keyboard Controls.
- **Label** (short form: **lb**; type: **`bool`**; # of arguments: **`1`**)
  The boolean property that shows/hides a record label that displays a properly modified current record of an animatable data file. If the curent data file is ot animatable, the label is not displayed.

  The specific contents of the label is controled by **LabelName**, **LabelScale**, **LabelUnit**, and **LabelDigits** properties. Namely, the label is shown as the following equation:

  ```
  name=value unit
  ```

  where *name* and *unit* are strings specified by string−values properties **LabelName** and **LabelUnit**, and *value* is the mathematical product of the property **LabelScale** and the value of the current file record. The number of decimal places to show in the *value* is set by the **LabelDigits** property.

  For example, if **LabelName**="Power", **LabelUnit**="Watts", and **LabelScale**=0.01, then a file with the record 1234 will have a label

  ```
  Power=12.34Watts
  ```

  A special value **LabelScale**=0 forms a label with name "z", no unit, and the value equal to 1e4/*record*−1, so that a file with the record 1234 will have a label

  ```
  z=7.10
  ```

  As you see, IFrIT remembers its origin!
- **LabelDigits** (short form: **ld**; type: **`int`**; # of arguments: **`1`**)
  See Label.
- **LabelName** (short form: **ln**; type: **`string`**; # of arguments: **`1`**)
  See Label.
- **LabelScale** (short form: **ls**; type: **`float`**; # of arguments: **`1`**)
  See Label.
- **LabelUnit** (short form: **lu**; type: **`string`**; # of arguments: **`1`**)
  See Label.
- **LightAngles** (short form: **la**; type: **`float`**; # of arguments: **`2`**)
  Two angles (elevation and azimuth) that specify the location of the main light. See descrion of IFrIT lights for more details on how light are controlled in IFrIT.
- **LightIntensity** (short form: **li**; type: **`float`**; # of arguments: **`3`**)
  IFrIT uses **LightKit** object from VTK to control light. Below is the (slightly edited) description of lights borrowed from VTK documentation.

  LightKit is designed to make general purpose lighting of VTK scenes simple, flexible, and attractive (or at least not horribly ugly without significant effort).

  A **LightKit** consists of three lights, a **main** light, a **fill** light, and a **headlight**. The main light is usually positioned so that it appears like an overhead light (like the sun, or a ceiling light). It is generally positioned to shine down on the scene from about a 45 degree angle vertically and at least a little offset side to side. The main light usually at least about twice as bright as the total of all other

lights in the scene to provide good modeling of object features.

The other lights in the kit (the fill light, headlight, and a pair of back lights) are weaker sources that provide extra illumination to fill in the spots that the main light misses. The fill light is usually positioned across from or opposite from the main light (though still on the same side of the object as the camera) in order to simulate diffuse reflections from other objects in the scene. The headlight, always located at the position of the camera, reduces the contrast between areas lit by the main and fill light. The two back lights, one on the left of the object as seen from the observer and one on the right, fill on the high–contrast areas behind the object. The **LightIntensity** property sets the intensities of the main, fill, and head lights. The back lights intensities are set automatically.

All lights are directional lights (infinitely far away with no falloff). Lights move with the camera. For simplicity, the position of lights in the **LightKit** can only be specified using two angles: the elevation (latitude) and azimuth (longitude) of the main light with respect to the camera, expressed in degrees. (Lights always shine on the camera's lookat point.) For example, a light at (elevation=0, azimuth=0) is located at the camera. A light at (elevation=90, azimuth=0) is above the lookat point, shining down. Negative azimuth values move the lights clockwise as seen above, positive values counter–clockwise. So, a light at (elevation=45, azimuth=–20) is above and in front of the object and shining slightly from the left side.

- **MarkerCurrent** (short form: **omc**; type: **`int`**; # of arguments: **`1`**)
  This property specifies the index of the current marker. Must take value between 1 and the value of Marker Max property.
- **MarkerLegend** (short form: **ml**; type: **`bool`**; # of arguments: **`1`**)
  A boolean property that toggles showing the legend for the existing markers.
- **MarkerLegendPosition** (short form: **mlp**; type: **`int`**; # of arguments: **`1`**)
  A position on the screen of the **Marker Legend**. This property takes only 2 values: 0 (for lower–left corner) or 1 (for lower–right corner).
- **MarkerMax** (short form: **omm**; type: **`int`**; # of arguments: **`1`**)
  The read–only property that returns the number of existing markers.
- **MeasuringBox** (short form: **mb**; type: **`bool`**; # of arguments: **`1`**)
  The boolean property that toggles showing the measuring box. The measuring box is a semi–transparent cube that can be moved in the visualization scene. Its size can be adjusted, and is always displayed on the screen, so the size of features that fit int the box are always known. When shown, the measuring box takes over the mouse and keyboard interaction.
- **MoveMarkerCaption** (short form: **mmc**; type: **`bool`**; # of arguments: **`1`**)
  A boolen property that toggles a **Caption Move** mode. In this mode the mouse interaction with the visualization scene is disabled. The only mouse interaction that is allowed is to click on the marker caption and drag it around the screen.
- **NoClone** (short form: **nc**; type: **`bool`**; # of arguments: **`1`**)
  A read–only property that returns `true` if this **View Module** owns its data, or `false` if it is a clone of other view module.
- **OpenGLCoordinates** (short form: **glc**; type: **`bool`**; # of arguments: **`1`**)
  This boolean property specifies the coordinate system used to labels scales and locations in the visualization scene. If this property is set to `true`, then the coordinate within the bounding box go from –1 to 1 in each of the 3 directions; the center of the bounding box is at (0,0,0), and the linear size of the bounding box is 2. If this property is `false`, then the value of the **Box Size** property is used to set the size of the bounding box, and the coordinates go from 0 to **Box Size** in all 3 directions.
- **ParticlesCurrent** (short form: **opc**; type: **`int`**; # of arguments: **`1`**)
  This property specifies the index of current particles. Must take value between 1 and the value of Particles Max property.

- **ParticlesMax** (short form: **opm**; type: **`int`**; # of arguments: **`1`**)
  The read–only property that returns the number of existing particles.
- **PlaceMarker** (short form: **pm**; type: **`bool`**; # of arguments: **`1`**)
  A boolen property that toggles a **Marker Placement** mode. In this mode the mouse cursor gets an outline bounding box, axes–aligned cross–hairs, and axes shadows that can be used to move the current marker interactively.
- **Position** (short form: **p**; type: **`int`**; # of arguments: **`2`**)
  The position of the visualization on the screen (returned as two–component integer array). This property is read–only – use the mouse to move the windows on the screen.
- **PostScriptOrientation** (short form: **pso**; type: **`int`**; # of arguments: **`1`**)
  This property sets the orientation of the PostScript images as portrait (0) or landscape (1).
- **PostScriptPaperFormat** (short form: **psf**; type: **`int`**; # of arguments: **`1`**)
  Sets the paper format for the PostScript image format. The valid values are from 0 to 10, which select the followinf formats respectively: A0, A1, A2, A3, A4, A5, A6, A7, A8, Letter, 11x17.
- **Ruler** (short form: **rl**; type: **`bool`**; # of arguments: **`1`**)
  This property toggles showing the ruler on top of the visualization window. The ruler is only visible in the parallel projection, because in the perspective projections the same distance projects on the screen differently depending on ots location in the scene.
- **RulerScale** (short form: **rs**; type: **`float`**; # of arguments: **`1`**)
  Sets the scale shown on the ruler. The scene will be zoomed in/out appropriatey to maintain consistency between the scale of the scene and the value shown on the ruler.
- **Size** (short form: **s**; type: **`int`**; # of arguments: **`2`**)
  This property controls the size of the visualization window on the screen. Opposite to the **Position** property, this property is writeable, i.e. changing this value also changes the size of the window on the screen.
- **Stereo** (short form: **ss**; type: **`bool`**; # of arguments: **`1`**)
  The boolean property that toggles the stereo mode. The same effect is obained by pressing a key "3" in the visualization window in the display mode.
- **StereoAlignmentMarkers** (short form: **sam**; type: **`bool`**; # of arguments: **`1`**)
  The boolean property that toggles showing alighnment markers for dual window stereo mode. This property has no effect if stereo is not used, or if the **Stereo Type** mode is not *dual windows*.
- **StereoType** (short form: **st**; type: **`int`**; # of arguments: **`1`**)
  Specifies the method to use to display stereo image. The following values are accepted:
  - **0**: Two eyes in two separate windows (for geowall–like set up).
  - **1**: Crystal Eyes special purpose hardware.
  - **2**: Simple blue–red stereo.
  - **3**: The interlaced render stereo type is for output to a VRex stereo projector. All of the odd horizontal lines are from the left eye, and the even lines are from the right eye. The user has to make the render window aligned with the VRex projector, or the eye will be swapped.
  - **4**: Left eye only.
  - **5**: Right eye only.
  - **6**: Dresden Display special purpose hardware.
- **SurfaceCurrent** (short form: **osc**; type: **`int`**; # of arguments: **`1`**)
  This property specifies the index of the current surface. Must take value between 1 and the value of Surface Max property.
- **SurfaceMax** (short form: **osm**; type: **`int`**; # of arguments: **`1`**)
  The read–only property that returns the number of existing surfaces.
- **UpdateRate** (short form: **r**; type: **`int`**; # of arguments: **`1`**)
  Specifies the interactive update rate. The main feature of VTK is that it can adjust rendering of a complex visualization scene to achieve a request update rate. If the rendering of one frames takes to long to keep up with the requested rate, some of the objects in the scene will be simplified temporarily

to maintain interactive rate. The rate is measured in frames per second, rounded to the nearest integer.

- **WindowNumber** (short form: **wn**; type: **`int`**; # of arguments: **`1`**)
  This read–only property returns the index of the current **View Module**.

3.22 Volume object

Volume object uses the volume rendering method to represent the full three–dimensional structure of the data. Several methods for volume rendering are available.

Short form: **w**

Available properties:

- **Ambient** (short form: **ma**; type: **`float`**; # of arguments: **`1`**)
  Properties **Ambient**, **Diffuse**, **Specular**, and **SpecularPower** control material properties of the object, as specified in OpenGL. The first three take a floating–point value between 0 and 1, while the **SpecularPower** property, which controls the OpenGL *shininess* property (it is called *specular power* in VTK), takes a floating–point value between 0 and 128. You need to know how OpenGL handles material properties to understand what these values mean.
- **BlendMode** (short form: **bm**; type: **`int`**; # of arguments: **`1`**)
  The **Blend Mode** is only meaningful for the Raycast and VolumePro methods. It accepts two values:
  - **0**: composite mode, when all values along a line–of–sight direction are composed together depending on the opacity function;
  - **1**: "maximum intensity" mode, when the maximum value along a line–of–sight direction is used to represent the opacity along the line–of–sight.
- **ClipPlane** (short form: **cp**; type: **`bool`**; # of arguments: **`1`**)
  A switch activating or deactivating the clipping plane. Takes a boolean (0/1) value.
- **CompositeMethod** (short form: **cm**; type: **`int`**; # of arguments: **`1`**)
  The **Composite Method** property is only meaningful for the Raycast method and composite blend mode; a value 0 of this property corresponds to the "classify first" method, a value 1 corresponds to "interpolate first" method. You need to know the basic of ray casting to understand what these methods mean.
- **DataPresent** (short form: **dp**; type: **`bool`**; # of arguments: **`1`**)
  A read–only boolean property returning 1 if the object has the data and 0 otherwise.
- **DataType** (short form: **dt**; type: **`int`**; # of arguments: **`1`**)
  This property specifies the type of data a given object uses. It has no effect for the basic edition of IFrIT, but extensions to IFrIT use multiple data types and this property is used to select among different types of data.
- **Diffuse** (short form: **md**; type: **`float`**; # of arguments: **`1`**)
  See Ambient.
- **InterpolationType** (short form: **it**; type: **`int`**; # of arguments: **`1`**)
  The value of 0 sets the nearest neighbor interpolation; the value of 1 switches to linear interpolation.
- **Method** (short form: **m**; type: **`int`**; # of arguments: **`1`**)
  Specifies the method for volume rendering:
  - **0**: *Raycast* method computes volume properties along rays cast from every point on the screne into the volume;

- **1**: *2D texture* method replaces the volume with a set of semi–transparent textures; this can be fast for small volumes and high quality videocards, although the quality of rendering is usually worse then in the *Raycast* method;
- **2**: *3D texture* method uses 3–dimensional textures, similar to 2D textures, except the visual quality is usually much higher even for small volumes. This method is only available under VTK 5 and only if your hardware supports it – not all videocards can handle 3D textures, but most of the most recet ones can;
- **3**: *VolumePro* method uses the VolumePro 1000 hardware volume rendering board; you need to have this expensive board to use this method.

- **NumReplicas** (short form: **nr**; type: **`int`**; # of arguments: **`6`**)
  Some of **View Objects** can be replicated, i.e. their identical replicas placed outside the bounding box. This 6–dimensional integer property specifies how many replicas should be placed in –X, +X, –Y, +Y, –Z, and +Z directions, respectively. This property only has effect if the data are periodic in at least some of the directions, and if the object can be replicated (not all **View Objects**.
- **OpacityFunction** (short form: **of**; type: any; # of arguments: **`1`**)
  The piece–wise function that specifies the opacity of a cell as a function of the variable value in the cell. The opacity function is specified in the same as as the **ParticleGroup:SizeFunction** property.
- **Palette** (short form: **p**; type: **`int`**; # of arguments: **`1`**)
  Sets the palette to render the volume with.
- **Ready** (short form: **r**; type: **`bool`**; # of arguments: **`1`**)
  A read–only boolean value for checking whether the object is ready to be shown (i.e. has the data and all the correct settigs). An attept to show an object that is not ready will not succeed.
- **Shading** (short form: **mw**; type: **`bool`**; # of arguments: **`1`**)
  A boolean switch that can be used to switch off shading of light on the material (so that a surface looks the same no matter what its orientation relative to lights and a camera is).
- **Specular** (short form: **ms**; type: **`float`**; # of arguments: **`1`**)
  See Ambient.
- **SpecularPower** (short form: **mp**; type: **`float`**; # of arguments: **`1`**)
  See Ambient.
- **Var** (short form: **v**; type: **`int`**; # of arguments: **`1`**)
  Sets the scalar variable to do the volume rendering of.
- **Visible** (short form: **vis**; type: **`bool`**; # of arguments: **`1`**)
  A read–only boolean value for checking whether the object is visible. It has a value of **true** (1) is the **View Object** is visible, and **false** otherwise. This property cannot be used to show/hide the object, only to test the visibility state.
- **XYResample** (short form: **rxy**; type: **`float`**; # of arguments: **`1`**)
  Properties **XYResample** and **ZResample** specify resampling factors in the plane of the screen (X–Y plane) and along the line–of–sight (Z direction). These properties are only meaningful for the Raycast and VolumePro methods.
- **ZResample** (short form: **rz**; type: **`float`**; # of arguments: **`1`**)
  See XYResample.

A Appendices

A.1 Codes For Writing IFrIT Data Files

A.1.1 Code Examples

Examples of computer codes for writing IFrIT data files are available in `docs` directory of IFrIT source distribution.

A.1.2 Fortran

```
C
C  Write to text uniform scalars data file
C
      subroutine WriteIFrITUniformScalarsTxtFile(n1,n2,n3,var1,var2,var3,
     .     filename)
      integer n1, n2, n3  ! Size of the computational mesh in 3 directions
      real*4 var1(n1,n2,n3)
      real*4 var2(n1,n2,n3) ! Three scalar variables
      real*4 var3(n1,n2,n3)
      character*(*) filename  ! Name of the file
      open(unit=1, file=filename)
      write(1,*) n1, n2, n3
      do k=1,n3
         do j=1,n2
            do i=1,n1
               write(1,*) var1(i,j,k), var2(i,j,k), var3(i,j,k)
            enddo
         enddo
      enddo
      close(1)
      return
      end
C
C  Write to binary uniform scalars data file
C
      subroutine WriteIFrITUniformScalarsBinFile(n1,n2,n3,var1,var2,var3,
     .     filename)
      integer n1, n2, n3  ! Size of the computational mesh in 3 directions
      real*4 var1(n1,n2,n3)
      real*4 var2(n1,n2,n3) ! Three scalar variables
      real*4 var3(n1,n2,n3)
      character*(*) filename  ! Name of the file
      open(unit=1, file=filename, form='unformatted')
      write(1) n1, n2, n3
      write(1) (((var1(i,j,k),i=1,n1),j=1,n2),k=1,n3)
      write(1) (((var2(i,j,k),i=1,n1),j=1,n2),k=1,n3)
      write(1) (((var3(i,j,k),i=1,n1),j=1,n2),k=1,n3)
      close(1)
      return
      end
C
C  Write to text uniform vectors data file
```

```
C
      subroutine WriteIFrITUniformVectorsTxtFile(n1,n2,n3,vect,filename)
      integer n1, n2, n3  ! Size of the computational mesh in 3 directions
      real*4 vect(3,n1,n2,n3)  ! Vector field
      character*(*) filename  ! Name of the file
      open(unit=1, file=filename)
      write(1,*) n1, n2, n3
      do k=1,n3
         do j=1,n2
            do i=1,n1
               write(1,*) vect(1,i,j,k), vect(2,i,j,k), vect(3,i,j,k)
            enddo
         enddo
      enddo
      close(1)
      return
      end
C
C  Write to binary uniform vectors data file
C
      subroutine WriteIFrITUniformVectorsBinFile(n1,n2,n3,vect,filename)
      integer n1, n2, n3  ! Size of the computational mesh in 3 directions
      real*4 vect(3,n1,n2,n3)  ! Vector field
      character*(*) filename  ! Name of the file
      open(unit=1, file=filename, form='unformatted')
      write(1) n1, n2, n3
      write(1) (((vect(1,i,j,k),i=1,n1),j=1,n2),k=1,n3)
      write(1) (((vect(2,i,j,k),i=1,n1),j=1,n2),k=1,n3)
      write(1) (((vect(3,i,j,k),i=1,n1),j=1,n2),k=1,n3)
      close(1)
      return
      end
C
C  Write to text uniform tensors data file
C
      subroutine WriteIFrITUniformTensorsTxtFile(n1,n2,n3,tens,filename)
      integer n1, n2, n3  ! Size of the computational mesh in 3 directions
      real*4 tens(6,n1,n2,n3)  ! Tensor field
      character*(*) filename  ! Name of the file
      open(unit=1, file=filename)
      write(1,*) n1, n2, n3
      do k=1,n3
         do j=1,n2
            do i=1,n1
               write(1,*) tens(1,i,j,k), tens(2,i,j,k), tens(3,i,j,k),
     .              tens(4,i,j,k), tens(5,i,j,k), tens(6,i,j,k)
            enddo
         enddo
      enddo
      close(1)
      return
      end
C
C  Write to binary uniform tensors data file
C
      subroutine WriteIFrITUniformTensorsBinFile(n1,n2,n3,tens,filename)
      integer n1, n2, n3  ! Size of the computational mesh in 3 directions
      real*4 tens(6,n1,n2,n3)  ! Tensor field
      character*(*) filename  ! Name of the file
      open(unit=1, file=filename, form='unformatted')
      write(1) n1, n2, n3
      write(1) (((tens(1,i,j,k),i=1,n1),j=1,n2),k=1,n3)
```

```
      write(1) (((tens(2,i,j,k),i=1,n1),j=1,n2),k=1,n3)
      write(1) (((tens(3,i,j,k),i=1,n1),j=1,n2),k=1,n3)
      write(1) (((tens(4,i,j,k),i=1,n1),j=1,n2),k=1,n3)
      write(1) (((tens(5,i,j,k),i=1,n1),j=1,n2),k=1,n3)
      write(1) (((tens(6,i,j,k),i=1,n1),j=1,n2),k=1,n3)
      close(1)
      return
      end
C
C  Write to text basic particles data file
C
      subroutine WriteIFrITBasicParticlesTxtFile(n,xl,yl,zl,xh,yh,zh,
     .     x,y,z,attr1,attr2,attr3,filename)
      integer n  ! Number of particles
      real*4 xl, yl, zl, xh, yh, zh  ! Bounding box
      real*4 x(n), y(n), z(n)  ! Particle positions (can be real*8)
      real*4 attr1(n), attr2(n), attr3(n)  ! Particle attributes
      character*(*) filename  ! Name of the file
      open(unit=1, file=filename)
      write(1,*) n
      write(1,*) xl, yl, zl, xh, yh, zh
      do i=1,n
         write(1,*) x(i), y(i), z(i), attr1(i), attr2(i), attr3(i)
      enddo
      close(1)
      return
      end
C
C  Write to binary basic particles data file
C
      subroutine WriteIFrITBasicParticlesBinFile(n,xl,yl,zl,xh,yh,zh,
     .     x,y,z,attr1,attr2,attr3,filename)
      integer n  ! Number of particles
      real*4 xl, yl, zl, xh, yh, zh  ! Bounding box
      real*4 x(n), y(n), z(n)  ! Particle positions (can be real*8)
      real*4 attr1(n), attr2(n), attr3(n)  ! Particle attributes
      character*(*) filename  ! Name of the file
      open(unit=1, file=filename, form='unformatted')
      write(1) n
      write(1) xl, yl, zl, xh, yh, zh
      write(1) (x(i),i=1,n)
      write(1) (y(i),i=1,n)
      write(1) (z(i),i=1,n)
      write(1) (attr1(i),i=1,n)
      write(1) (attr2(i),i=1,n)
      write(1) (attr3(i),i=1,n)
      close(1)
      return
      end
```

A.1.3 C

```
#include <stdio.h>

/*  Write to text uniform scalars data file  */

int WriteIFrITUniformScalarsTxtFile(
    int n1, int n2, int n3, /* Size of the computational mesh in 3 directions */
    float *var1, float *var2, float *var3, /* Three scalar variables */
```

```
    char *filename) /* Name of the file */
{
   int i, j, k; FILE *F;
   F = fopen(filename,"w"); if(F == NULL) return 1;
   fprintf(F,"%d %d %d\n",n1,n2,n3);
   for(k=0; k<n3; k++)
   {
      for(j=0; j<n2; j++)
      {
         for(i=0; i<n1; i++)
         {
            fprintf(F,"%g %g %g\n",var1[i+n1*(j+n2*k)],
                                   var2[i+n1*(j+n2*k)],
                                   var3[i+n1*(j+n2*k)]);
         }
      }
   }
   fclose(F);
   return 0;
}

/*  Write to binary uniform scalars data file  */

int WriteIFrITUniformScalarsBinFile(
    int n1, int n2, int n3, /* Size of the computational mesh in 3 directions */
    float *var1, float *var2, float *var3, /* Three scalar variables */
    char *filename) /* Name of the file */
{
   int ntemp; FILE *F; /* ntemp should be declared long on a 16-bit machine */
   F = fopen(filename,"w"); if(F == NULL) return 1;
   ntemp = 12;
   fwrite(&ntemp,4,1,F);
   fwrite(&n1,4,1,F);
   fwrite(&n2,4,1,F);
   fwrite(&n3,4,1,F);
   fwrite(&ntemp,4,1,F);
   ntemp = 4*n1*n2*n3;
   fwrite(&ntemp,4,1,F); fwrite(var1,4,n1*n2*n3,F); fwrite(&ntemp,4,1,F);
   fwrite(&ntemp,4,1,F); fwrite(var2,4,n1*n2*n3,F); fwrite(&ntemp,4,1,F);
   fwrite(&ntemp,4,1,F); fwrite(var3,4,n1*n2*n3,F); fwrite(&ntemp,4,1,F);
   fclose(F);
   return 0;
}


/*  Write to text uniform vectors data file  */

int WriteIFrITUniformVectorsTxtFile(
    int n1, int n2, int n3, /* Size of the computational mesh in 3 directions */
    float *vect, /* Vector field */
    char *filename) /* Name of the file */
{
   int i, j, k; FILE *F;
   F = fopen(filename,"w"); if(F == NULL) return 1;
   fprintf(F,"%d %d %d\n",n1,n2,n3);
   for(k=0; k<n3; k++)
   {
      for(j=0; j<n2; j++)
      {
         for(i=0; i<n1; i++)
         {
            fprintf(F,"%g %g %g\n",vect[0+3*(i+n1*(j+n2*k))],
```

```
                                    vect[1+3*(i+n1*(j+n2*k))],
                                    vect[2+3*(i+n1*(j+n2*k))]);
         }
      }
   }
   fclose(F);
   return 0;
}

/*  Write to binary uniform vectors data file  */

int WriteIFrITUniformVectorsBinFile(
    int n1, int n2, int n3, /* Size of the computational mesh in 3 directions */
    float *vect, /* Vector field */
    char *filename) /* Name of the file */
{
   int i, ntemp; FILE *F; /* ntemp should be declared long on a 16-bit machine */
   F = fopen(filename,"w"); if(F == NULL) return 1;
   ntemp = 12;
   fwrite(&ntemp,4,1,F);
   fwrite(&n1,4,1,F);
   fwrite(&n2,4,1,F);
   fwrite(&n3,4,1,F);
   fwrite(&ntemp,4,1,F);
   ntemp = 4*n1*n2*n3;
   fwrite(&ntemp,4,1,F);
   for(i=0; i<n1*n2*n3; i++) fwrite(vect+3*i+0,4,1,F);
   fwrite(&ntemp,4,1,F);
   fwrite(&ntemp,4,1,F);
   for(i=0; i<n1*n2*n3; i++) fwrite(vect+3*i+1,4,1,F);
   fwrite(&ntemp,4,1,F);
   fwrite(&ntemp,4,1,F);
   for(i=0; i<n1*n2*n3; i++) fwrite(vect+3*i+2,4,1,F);
   fwrite(&ntemp,4,1,F);
   fclose(F);
   return 0;
}

/*  Write to text uniform tensors data file  */

int WriteIFrITUniformTensorsTxtFile(
    int n1, int n2, int n3, /* Size of the computational mesh in 3 directions */
    float *tens, /* Tensor field */
    char *filename) /* Name of the file */
{
   int i, j, k; FILE *F;
   F = fopen(filename,"w"); if(F == NULL) return 1;
   fprintf(F,"%d %d %d\n",n1,n2,n3);
   for(k=0; k<n3; k++)
   {
      for(j=0; j<n2; j++)
      {
         for(i=0; i<n1; i++)
         {
            fprintf(F,"%g %g %g %g %g %g\n",tens[0+6*(i+n1*(j+n2*k))],
                                            tens[1+6*(i+n1*(j+n2*k))],
                                            tens[2+6*(i+n1*(j+n2*k))],
                                            tens[3+6*(i+n1*(j+n2*k))],
                                            tens[4+6*(i+n1*(j+n2*k))],
                                            tens[5+6*(i+n1*(j+n2*k))]);
         }
      }
```

```
   }
   fclose(F);
   return 0;
}

/*  Write to binary uniform tensors data file  */

int WriteIFrITUniformTensorsBinFile(
    int n1, int n2, int n3, /* Size of the computational mesh in 3 directions */
    float *tens, /* Tensor field */
    char *filename) /* Name of the file */
{
   int i, ntemp; FILE *F; /* ntemp should be declared long on a 16-bit machine */
   F = fopen(filename,"w"); if(F == NULL) return 1;
   ntemp = 12;
   fwrite(&ntemp,4,1,F);
   fwrite(&n1,4,1,F);
   fwrite(&n2,4,1,F);
   fwrite(&n3,4,1,F);
   fwrite(&ntemp,4,1,F);
   ntemp = 4*n1*n2*n3;
   fwrite(&ntemp,4,1,F);
   for(i=0; i<n1*n2*n3; i++) fwrite(tens+6*i+0,4,1,F);
   fwrite(&ntemp,4,1,F);
   fwrite(&ntemp,4,1,F);
   for(i=0; i<n1*n2*n3; i++) fwrite(tens+6*i+1,4,1,F);
   fwrite(&ntemp,4,1,F);
   fwrite(&ntemp,4,1,F);
   for(i=0; i<n1*n2*n3; i++) fwrite(tens+6*i+2,4,1,F);
   fwrite(&ntemp,4,1,F);
   fwrite(&ntemp,4,1,F);
   for(i=0; i<n1*n2*n3; i++) fwrite(tens+6*i+3,4,1,F);
   fwrite(&ntemp,4,1,F);
   fwrite(&ntemp,4,1,F);
   for(i=0; i<n1*n2*n3; i++) fwrite(tens+6*i+4,4,1,F);
   fwrite(&ntemp,4,1,F);
   fwrite(&ntemp,4,1,F);
   for(i=0; i<n1*n2*n3; i++) fwrite(tens+6*i+5,4,1,F);
   fwrite(&ntemp,4,1,F);
   fclose(F);
   return 0;
}

/*  Write to text basic particles data file  */

int WriteIFrITBasicParticlesTxtFile(
    int n, /* Number of particles */
    float xl, float yl, float zl, float xh, float yh, float zh, /* Bounding box */
    float *x, float *y, float *z, /* Particle positions (can be double) */
    float *attr1, float *attr2, float *attr3, /* Particle attributes */
    char *filename) /* Name of the file */
{
   int i; FILE *F;
   F = fopen(filename,"w"); if(F == NULL) return 1;
   fprintf(F,"%d\n",n);
   fprintf(F,"%g %g %g %g %g %g\n",xl,yl,zl,xh,yh,zh);
   for(i=0; n>i; i++)
   {
      fprintf(F,"%g %g %g %g %g %g\n",x[i],y[i],z[i],
                           attr1[i],attr2[i],attr3[i]);
   }
   fclose(F);
```

```
   return 0;
}

/*  Write to binary basic particles data file  */

int WriteIFrITBasicParticlesBinFile(
    int n, /* Number of particles */
    float xl, float yl, float zl, float xh, float yh, float zh, /* Bounding box */
    float *x, float *y, float *z, /* Particle positions (can be double) */
    float *attr1, float *attr2, float *attr3, /* Particle attributes */
    char *filename) /* Name of the file */
{
   int i, ntemp; FILE *F;
   F = fopen(filename,"w"); if(F == NULL) return 1;
   ntemp = 4;
   fwrite(&ntemp,4,1,F); fwrite(&n,4,1,F); fwrite(&ntemp,4,1,F);
   ntemp = 24;fwrite(&ntemp,4,1,F);
   fwrite(&xl,4,1,F);
   fwrite(&yl,4,1,F);
   fwrite(&zl,4,1,F);
   fwrite(&xh,4,1,F);
   fwrite(&yh,4,1,F);
   fwrite(&zh,4,1,F);
   fwrite(&ntemp,4,1,F);
   ntemp = sizeof(x[0])*n;
   fwrite(&ntemp,4,1,F); fwrite(x,sizeof(x[0]),n,F); fwrite(&ntemp,4,1,F);
   fwrite(&ntemp,4,1,F); fwrite(y,sizeof(y[0]),n,F); fwrite(&ntemp,4,1,F);
   fwrite(&ntemp,4,1,F); fwrite(z,sizeof(z[0]),n,F); fwrite(&ntemp,4,1,F);
   ntemp = 4*n;
   fwrite(&ntemp,4,1,F); fwrite(attr1,4,n,F); fwrite(&ntemp,4,1,F);
   fwrite(&ntemp,4,1,F); fwrite(attr2,4,n,F); fwrite(&ntemp,4,1,F);
   fwrite(&ntemp,4,1,F); fwrite(attr3,4,n,F); fwrite(&ntemp,4,1,F);
   fclose(F);
   return 0;
}
```

A.1.4 IDL

```
;
;  Write to text scalar field data file
;
pro WriteIFrITUniformScalarsTxtFile, n1, n2, n3, var1, var2, var3, filename
; n1, n2, n3: size of the computational mesh in 3 directions
; var1, var2, var3: three scalar variables
; filename: name of the file
openw, 1, filename
printf, 1, n1, n2, n3
for k=0,n3-1 do $
for j=0,n2-1 do $
for i=0,n1-1 do $
printf, 1, var1[i,j,k], var2[i,j,k], var3[i,j,k]
close, 1
end
;
;  Write to binary scalar field data file
;
pro WriteIFrITUniformScalarsBinFile, n1, n2, n3, var1, var2, var3, filename
; n1, n2, n3: size of the computational mesh in 3 directions
; var1, var2, var3: three scalar variables
```

```
; filename: name of the file
openw, 1, filename, /F77_UNFORMATTED
writeu, 1, long([n1,n2,n3])
writeu, 1, var1
writeu, 1, var2
writeu, 1, var3
close, 1
end
;
;  Write to text uniform vectors data file
;
pro WriteIFrITUniformVectorsTxtFile, n1, n2, n3, vect, filename
; n1, n2, n3: size of the computational mesh in 3 directions
; vect[3,n1,n2,n3]: uniform vectors
; filename: name of the file
openw, 1, filename
printf, 1, n1, n2, n3
for k=0,n3-1 do $
for j=0,n2-1 do $
for i=0,n1-1 do $
printf, 1, vect[0,i,j,k], vect[1,i,j,k], vect[2,i,j,k]
close, 1
end
;
;  Write to binary uniform vectors data file
;
pro WriteIFrITUniformVectorsBinFile, n1, n2, n3, vect, filename
; n1, n2, n3: size of the computational mesh in 3 directions
; vect[3,n1,n2,n3]: uniform vectors
; filename: name of the file
openw, 1, filename, /F77_UNFORMATTED
writeu, 1, long([n1,n2,n3])
writeu, 1, vect[0,*,*,*]
writeu, 1, vect[1,*,*,*]
writeu, 1, vect[2,*,*,*]
close, 1
end
;
;  Write to text uniform tensors data file
;
pro WriteIFrITUniformTensorsTxtFile, n1, n2, n3, tens, filename
; n1, n2, n3: size of the computational mesh in 3 directions
; tens[6,n1,n2,n3]: uniform tensors
; filename: name of the file
openw, 1, filename
printf, 1, n1, n2, n3
for k=0,n3-1 do $
for j=0,n2-1 do $
for i=0,n1-1 do $
printf, 1, tens[0,i,j,k], tens[1,i,j,k], tens[2,i,j,k], $
           tens[3,i,j,k], tens[4,i,j,k], tens[5,i,j,k]
close, 1
end
;
;  Write to binary uniform tensors data file
;
pro WriteIFrITUniformTensorsBinFile, n1, n2, n3, tens, filename
; n1, n2, n3: size of the computational mesh in 3 directions
; tens[6,n1,n2,n3]: uniform vectors
; filename: name of the file
openw, 1, filename, /F77_UNFORMATTED
writeu, 1, long([n1,n2,n3])
```

```
writeu, 1, tens[0,*,*,*]
writeu, 1, tens[1,*,*,*]
writeu, 1, tens[2,*,*,*]
writeu, 1, tens[3,*,*,*]
writeu, 1, tens[4,*,*,*]
writeu, 1, tens[5,*,*,*]
close, 1
end
;
;  Write to text basic particles data file
;
pro WriteIFrITBasicParticlesTxtFile, n, xl, yl, zl, xh, yh, zh, $
x, y, z, attr1, attr2, attr3, filename
; n: number of particles
; xl, yl, zl, xh, yh, zh: bounding box (can be double)
; x, y, z: particle positions (can be double)
; attr1, attr2, attr3: particle attributes/
; filename: name of the file
openw, 1, filename
printf, 1, n
printf, 1, xl, yl, zl, xh, yh, zh
for i=0,n-1 do $
printf, 1, x[i], y[i], z[i], attr1[i], attr2[i], attr3[i]
close, 1
end
;
;  Write to binary basic particles data file
;
pro WriteIFrITBasicParticlesBinFile, n, xl, yl, zl, xh, yh, zh, $
x, y, z, attr1, attr2, attr3, filename
; n: number of particles
; xl, yl, zl, xh, yh, zh: bounding box (can be double)
; x, y, z: particle positions (can be double)
; attr1, attr2, attr3: particle attributes/
; filename: name of the file
openw, 1, filename, /F77_UNFORMATTED
writeu, 1, long(n)
writeu, 1, float([xl,yl,zl,xh,yh,zh])
writeu, 1, x
writeu, 1, y
writeu, 1, z
writeu, 1, attr1
writeu, 1, attr2
writeu, 1, attr3
close, 1
end
```

A.2 License Agreement

A.2.1 Overview

The standard edition of IFrIT is distributed under the GNU GPL License. Extensions of IFrIT may impose their own licenses.

For those not familiar with the GNU GPL, the license basically allows you to:

- Use the IFrIT software and source code at no charge.
- Distribute verbatim copies of the software in source form or as binaries you create.
- Sell verbatim copies of the software for a media fee, or sell support for the software.
- Distribute or sell your own modified version of IFrIT so long as the source code is made available under the GPL.

What this license **does not** allow you to do is make changes or add features to IFrIT and then sell a binary distribution without source code. You must provide source for any changes or additions to the software, and all code must be provided under the GPL.

A.2.2 GNU General Public License

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software−−to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it. For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSETERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
- b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
- c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

  These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

  Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

  In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

- a) Accompany it with the complete corresponding machine−readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine−readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty−free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to

refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR

REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS