11 11 TEX LATEX

# DataViewer Scene File Format Documentation

Dana Vrajitoru, danav@masg1.epfl.ch v0.1, Fri Aug 25 15:39:36 CEST 2000

This is the user documentation of the file for constructing geometrical scenes inside the DataViewer scientific visualization system.

## Contents

# 1 Introduction

This section presents the general notions about the scene file (*.scn).

All lines starting with ***** are messages that will be printed on the screen (without the *) and have no other effect on the scene.

All lines starting with / are comments that will simply be ignored. Longer comments (on more than 1 line) can also be enclosed by /* and */.

The file can contain a number of objects and properties. For each object or property, the file must contain a command made of a name and a body.

The **name** of the any command must be one of the legal tokens. Although it is recommended that the whole name of the command should be typed, the program only needs the information that can identify the command from other tokens. For example, for the box command, the b letter would be enough, as there is no other command starting with b. For the triangle, 3 letters are needed, because the translation has the first 2 letters in common with it.

The **body** of any command starts with { and ends with }. It contains all the information concerning the object. This information must be given in the following order:

- 1D, 2D or 3D data defining the object,
- properties and parameters specific to the object,
- general properties

Any number of properties can be attached to a body (color, translations, rotations, etc.) and will be applied to the objects who's body they are in.

There is a special category of objects, which inherit from the class **DVpick_components**. In the syntax summary they are denoted as "complex objects". For these objects, the data concerning them can be followed by any of the "pick_components" properties. In the syntax summary, these properties are called "comples properties".

The objects that use 2D or 3D data inherit from this class: the boxes, the lines, the rotation surfaces, the spheres, the surfaces, the triangle strips and grids. The properties related to this class concern especially how to use the coordinates of the points, and is designed for the case when the points have more than 3 coordinates.

# 2 Syntax summary

# 3 Describing data

## 3.1 Contents



## 3.2 1D data

A 1D data set represents a point. It can have any number of coordinates, including 0 (working theoretically, not yet tested). If the number of expected coordinates is known (3 by default), and the point has this number of specified coordinates, then the values of the coordinates can be listed without any other . The 1D data set must be enclosed by $<$ and $>$ in the following cases :

- the number of expected coordinates in not known, and we want it to be different from 3; the number of expected coordinates is set by the first point of the 3.3 (2D) or 3.4 (3D) data sets (see next sections);
- the number of expected coordinates is known, but the number of specified coordinates for this point is less than the expected one; the missing coordinates are set to 0 by default; some objects might set the missing coordinates otherwise.

| **Name** | **Outline** | **Meaning** |
|---|---|---|
| / | / text | comment |
| | * text | print text |
| 4.2 ({ } ) | { object_ data list_ of_ properties } | body of an object |
| | { list_ of_ objects_ or_ properties } | node object |
| | { 2D data }{ 2D data } **...** { 2D data } | 3.4 (3D data) |
| < > | < some_ int_ or_ float_ values> | 3.2 (1D data ) |
| | 1D data 1D data ... 1D data | 3.3 (2D data ) |
| 4.3 (animate) | **animate** { | animation node |
| | firstFrame number | |
| | stepFrame number | |
| | notAnimate number | |
| | optimize | |
| | lastFrame number | |
| | speed number | |
| | list_ of_ objects_ or_ properties } | |
| 4.4 (axes) | **axes** { | axes object |
| | center_ data | |
| | directions_ data | |
| | list_ of_ properties | |
| | } | |
| 4.5 (boxes) | **boxes** { | set of boxes: |
| | 3D_ corners_ data | a complex object |
| | list_ of_ properties | |
| | } | |
| 5.3 (ccoords) | **ccoords** { color_ position } | set color coordinate: |
| | | a complex property |
| 5.2 (color) | **color** { < red green blue blend > which} | color property |
| color | **color** { red green blue blend } | simple color property |
| 5.4 (color_ map) | **color_ map** | color map: |
| | a list of real numbers | a complex property |
| | a list of simple color properties | |
| | } | |
| 4.6 (cones) | **cones** { | set of cones: |
| | 3D_ corners_ data | a complex object |
| | ray { column_ number } | |
| | list_ of_ properties | |
| | } | |
| 4.7 (frlines) | **frlines** } | frames lines: |
| | 3D_ points_ data | a complex object |
| | cylinder | |
| | ribbon { rx_ loc ry_ loc rz_ loc rcolor_ loc } | |
| | ribbon { asymetric } | |
| | ribbon { vector } | |
| | list_ of_ properties | |
| | } | |
| 5.6 (gcomp) | **gcomp** { value } | geometry complexity property |
| 5.7 (lcomp) | **lcomp** { value } | light complexity property |
| 4.8 (lines) | **lines** { | set of lines: |
| | 3D_ points_ data | a complex object |
| | **ribbon** { rx_ loc ry_ loc rz_ loc } | |
| | **cylinder** | |
| | list_ of_ properties | |
| | } | |
| 4.9 (points) | **points** { | set of points: |
| | 2D_ points_ data | a complex object |
| | list_ of_ properties | |
| | } | |
| 4.10 (poly) | **poly** { | set of polygons: |
| | 3D_ points_ data | a complex object |
| | **norms** 3D_ norms_ data | |
| | list_ of_ properties | |
| | } | |

| **Name** | **Outline** | **Meaning** |
|---|---|---|
| 5.8 (rotate) | **rotate** {<br>axis_ x axis_ y axis_ z<br>angle<br>**radians**<br>} | rotation property |
| 4.11 (rsurface) | **rsurface** {<br>2D_ points_ data<br>angle an_ angle<br>list_ of_ properties<br>} | rotation surface object:<br>a complex object |
| 5.9 (scale) | **scale** { x_ value y_ value z_ value} | scaling property |
| 4.12 (spheres) | **spheres** {<br>2D_ centers_ data<br>**ray** { ray_ value }<br>list_ of_ properties<br>} | set of spheres:<br>a complex object |
| 4.13 (surface) | **surface** {<br>point_ number<br>{ list_ of_ point_ numbers }<br>{ 2D_ points_data}<br>**norms** { 2D_ norms_data }<br>{ list_ of_ patch_ sizes }<br>{ 2D_ int_ patch_ point_ numbers_ data }<br>**smooth**<br>list_ of_ properties<br>} | surface:<br>a complex object |
| 4.14 (texture) | **texture** {<br>4_ by_ 3_ 2D_ data<br>a_ file_ name<br>list_ of_ properties<br>} | texture quadrilateral object |
| 5.10 (translate) | **translate** { x y z} | translation property |
| 4.15 (triangle) | **triangle** { nodes_ norms_ data list_ of_ properties } | triangle object |
| 4.16 (tgrid) | **tgrid** {<br>3D_ nodes_ data<br>**norms** 3D_ norms_ data<br>list_ of_ properties<br>} | triangle grid:<br>a complex object |
| 4.17 (tstrip) | **tstrip** {<br>3D_ nodes_ data<br>**norms** 3D_ norms_ data<br>list_ of_ properties<br>} | triangle strip:<br>a complex object |
| 5.5 (xyz) | **xyz** { x_ position y_position z_ position } | set coordinates:<br>a complex property |
| 5.11 (width) | **width** { value location} | width property |

The 1D data set can be enclosed by $<$ and $>$ whenever you want, even if it is not necessary.

**Examples**:

0 0 0

0 0 1 3.0 -1

$<$1 5 7.2 .9 -3.2$>$

## 3.3 2D data

A 2D data set represents a set of points, each being described as a 3.2 (1D data set). They are used by various objects to represent string of points related in a certain way. For example, the DVlines object used 2D data to represent a string of lines, so there's a line between any 2 adjacent points.

For a stand alone 2D data set (not part of a 3D data set) and for the first 2D data set out of a 3D data set, the first point defines the number of expected coordinates. If the first point is not enclosed by $<$ and $>$, the number of expected coordinates is 3. Otherwise, the number of real values enclosed by $<$ and $>$ for the first point will set the expected number of coordinates for all the points that follow.

If the 2D data set is enclosed by { and } , then it is considered as a part of a 3D data set, even if there is only 1 2D data set.

**Examples**:

A simple 2D set with 3 coordinates.

25.2 0 9.3

25.2 5 9.3

20 0 11.4

20 5 11.4

A set with 7 coordinates. The 4th point has only 5 coordinates.

$<$4 1 0 0 1 0 0$>$

4 1 2 1 1 0 0

4 3 5 2 1 0 0

$<$4 1 7 3 1$>$

**Note**. A 1D data set is a particular case of 2D data set and can be used whenever a 2D data set is required.

### 3.4 3D data

A 3D data set is a list of 3.3 (2D) data sets enclosed by { and }. As mentioned before, the first point of the first 2D data set defines the expected number of coordinates for the entire 3D data set.

**Example**:

{<42 1> 1 0 0 28 41 29 42 2 }

{4 28 3 3 14 2 }

{7 3 8 10 }

{4 10 6 9 }

The number of points in each 2D data set from the 3D data set can be different, but the number of coordinates is the same everywhere.

**Note**. A 2D data set is a particular case of 3D data set and can be used whenever a 3D data set is required. recursively, this also applies to a 1D data set.

## 4 Objects

### 4.1 Contents

## 4.2 Node

The node object has no name field. The body can contain any number of objects and properties. The command creates a new node of type container ( DVcontainer) . This node is the child of the node corresponding to the { } which enclose the command, or the dv.top, if the command is not enclosed by any other.

All the objects created within { and } are related to the created node. All the properties declared within { and }apply to the node, which means that they apply to all the objects related to it, except if the object redeclares the same property.

The set of { } pairs in the scene actually create a tree structure.

## 4.3 Animation node

Name field:

**animate**

The animation node object is similar to the node object, with the only difference in the way the objects inside it are rendered. While the objects depending on a simple node are all rendered simultaneously, the objects contained in an animation node are rendered one by one, depending on the frame number. This has the effect of creating an animation from the objects contained in the animation node.

The body of the animation node can contain any number of objects and properties. The command creates an object of type DVselection_container. This kind of node is attached to the scene tree the same way as any other type of node.

The animation node has the following specific properties:

**firstFrame number**

This sets the first frame for the animation. Giving this parameter a value greater than 1, you can start the animation with any of the objects contained in the animation node.

**stepFrame number**

This property sets a culling factor for the animation. Thus, if this parameter is greater than 1, only each number objects will be rendered.

**notAnimate number**

This parameter allows you to visualize all the animation node as a usual, non animated node. If the value of this parameter is 1, then all the objects in the node will be rendered simultaneously, and the animation window will have no effect on them anymore. If this property is present without a number, then it is assigned the 1 value by default.

**optimize**

If this property is turned on, then all the other nodes in the tree structure will be turned into animation nodes, and the objects inside them will be multiplied by the number of frames in the animation. This choice is not recommended unless the objects that are static in the scene have a small size compared to the others.

**lastFrame number**

This property sets the last frame in the animation. By default the last frame is set to the maximal number of objects in all the animation nodes from the tree. If this property is set to a lower number than the number of objects, then all the animations will be cut after this number of frames. If this number is bigger than the number of objects, then the last part of the animation will render empty objects.

**speed number**

This property is linked to the animation speed. The number specifies the number of frames per second in the animation.

### 4.3.1 Example:

animate {

firstFrame 1

stepFrame 1

notAnimate 0

optimize

lastFrame 10

speed 20

lines {...}

}

## 4.4 Axes

Name field:

**axes**

The axes object is normally used for a frame in the 3D space and is defined by a center and 3 directions. Thus, the body of the axes object can contain:

- nothing, in which case, the orthonormal frame is defined:

  center: 0 0 0

  Ox: 1 0 0

  Oy: 0 1 0

  Oz: 0 0 1

- 3 real numbers representing the center. In this case, the orthonormal frame is translated with the coordinates of the center

- 12 real numbers representing the center plus the 3 directions
- any number of simple properties.

By the fact that the center of the axes is in the first place of the body, if you want to specify some non default direction, you must also include the center explicitly, or else the first direction will be read as the center.

The properties used by the axes objects are:

- The **color**: this object uses the 3 colors that can be attached to any object. If the 3 colors are defined, each vector composing the axes object will use its own color. If there are only less than 3 colors attached to the object, each direction will use the color corresponding to it, or if this color doesn't exist, the closest previous color. If there is no previous color, it will use the color of the node containing the object. By default the top node has 1 white color attached to it.

- The **width**:

  width of 0 (the default option) defines the width of the lines as number of pixels

  width of 1 defines a scaling factor that will be applied to each coordinate of each of the 3 axes. A more precise scaling can be obtained by the "scale" property.

### 4.4.1 Example:

```
axes {
    1 0 0
    0 1 1
    1 0 1
    1 1 0
    color { <0 0 1> 0}
    color { <1 0 0> 1}
    color { <1 1 0> 2}
    width { 2 }
}
```

The scale, translate, and rotate properties also apply to the axes object in the usual way.

## 4.5 Boxes

Name field:

**boxes**

A set of boxes is defined by a number of boxes strings. Each box is defined by two opposite corners which will be linked by the x, y and z planes. If the corners are in the same x, y or z plane, then the box is a rectangle.

The body of the set of boxes can contain nothing, in which case the boxes are set by default to 1 box string containing 1 box defined by the corners:

0.0, 0.0, 0.0,

1.0, 1.0, 1.0

If the set of boxes is defined by the user, then the first element of the boxes object body must be a 3D data set.

If the number of coordinates in the boxes is 2, the z coordinate is set by default alternatively to the 0 and 1 values, so that all the boxes have a depth of 1.0.

### 4.5.1 Example:

boxes {

{<42 1>

1 0

0 28

41 29

42 2 }

{ 4 28

3 3

14 2 }

}

The boxes do not have any particular property, but inherit from the class DVpick_components.

## 4.6 Cones

Name field:

**cones**

A set of cones is defined by a number of cones strings. Each point of a cone string must contain in principle 4 values, 3 for the center, and the fourth for the radius. For each string, each two adjacent cone points define a cone with orientation along the line passing by the two centers.

The body of the set of cones can contain nothing, in which case the cones are set by default to 1 cone string containing 1 cone defined by the centers:

0.0, 0.0, 0.0, 1.0

1.0, 0.0, 0.0, 1.0

As the two points define the same radius, the default cone is a cylinder.

If the set of cones is defined by the user, then the first element of the cones object body must be a 3D data set. We remind you that if each point has more than 3 coordinates, which is normally the case for a cone object, then the first point in the 3D data set must be enclosed by < and > .

By default the fourth coordinate represents the radius. If you want to use another coordinate for the radius, you can add, anywhere after the data section, the following specific property:

ray {column_number}

Note that the column numbers start from 0. In the example below, the column number 4 is actually the fifth column.

### 4.6.1 Example:

cones {

<0 0 0 1 1>

1 0 0 2 0.5

2 0 0 3 1

3 0 0 4 0.5

ray {4}

}

The cones class inherits from the class DVpick_components.

## 4.7 Set of framed lines with multiple ribbons

Name field:

**frlines**

A set of framed lines with multiple ribbons is defined by some 3D data followed by the set properties, and, at last, by some general object properties, and each of them is independently optional.

The points data is a 3D data set. If no information is present, a default line is constructed:

0 0 0 -> 1 0 0.

Each 2D data set represents a string of lines, and several of them can be present in one lines object.

The properties of the set of framed lines can be the cylinder and any number of ribbon properties.

- By default the centerline is drawn as simple OpenGL lines. The cylinder property is set by the keyword cylinder and causes the centerline to be drawn as cylinders. Note that this is done by setting the line status of the object to DV_CYLINDER. In this case, width {value} sets the ray of the cylinder which is equal to 1.0 by default.
- There are 3 kinds of ribbon properties. They are set by the keyword ribbon, followed by {} containing:
  - 4 or less integer numbers representing the columns used for the coordinates of the ribbon plus the color. Any number of ribbons can be added in this way, and all of them will be displayed.
  - the asymetric keyword, making all the ribbons to be displayed only on one side of the centerline. By default the ribbons are mirrored according to the centerline.
  - the vector_field keyword, which causes all the ribbons to be rendered as a vector field instead of a surface which is the default option.

The width property applies to the framed lines in the following manner:

- if the line is rendered as a simple line, the width of 0 (default) represents the number of pixels in the line
- if the line is rendered as a cylinder, the width of 0 (default) represents the ray of the cylinder

  If there is are some ribbons defined, the width of 1 (width {value 1}) scales the ribbons with this value.

### 4.7.1 Example:

frlines {

<19 0.895419 -0.527986 0.95 0.95 0.895419 -0.527986 0.95 0.95 0.95 -0.0920471 -0.715969 0.95 0.95 0.95 0.95 0.95>

20 0.890337 -0.601784 1 1 0.890337 -0.601784 1 1 1 0.000252651 -0.76089 1 1 1 1 1

xyz {1 2 3}

ccoord {4}

width {0.5 1}

ribbon { 5 6 7 8 }

ribbon { 9 10 11 12 }

ribbon { 13 14 15 16 }

ribbon { asymetric }

ribbon { vector_field }

}

The framed lines class also inherits from the DVpick_components class.

## 4.8 Set of lines, ribbons or cylinders

Name field:

**lines**

A set of lines, ribbons or cylinder is defined by some points data followed by the set properties, and, at last, by some general object properties, and each of them is independently optional.

The **points data** is a 3D data set. If no information is present, a default line is constructed:

0 0 0 -> 1 0 0.

Each 2D data set represents a string of lines, and several of them can be present in one lines object.

The **properties** of the set of lines can be : the cylinder, and the ribbon property.

- The cylinder property is set by the keyword **cylinder** and sets the line status of the object to DV_CYLINDER. If DV_CYLINDER is on, then the centerline is drawn as cylinders, otherwise as lines.
- The ribbon property is set by the keyword **ribbon** and sets the ribbon status of the object to DV_RIBBON. The ribbon property can be followed by 3 integer values inside curly braces indicating the indexes of the x y z coordinates of the ribbon vector.

### 4.8.1 Example:

lines {

<1 0 0 1 1 1>

1.5 0 0 1 1 1

2 0 0 1 1 1

cylinder

ribbon {3 4 5}

}

See the width section for its application to lines. The lines class also inherits from the DVpick_components class.

## 4.9 Set of points

Name field:

**points**

This class defines a list of simple points. If the body of the object is empty, the default points object will contain the origin point (0, 0, 0). The data in the points object is two dimensional. They are drawn as simple OpenGL points, whose number of pixels can be defined by the width property. The points class also inherits its properties from the DVpick_components class.

### 4.9.1 Example:

points {

<0 1 0 0>

0.01 0.972464378 0.185507501 0.025

0.02 0.911180956 0.360762062 0.05

width {3}

}

## 4.10 Set of polygons

Name field:

**poly**

The set of polygons can be set by default ( an empty body :{ } ) or be specified by a list of nodes ( at least 3) , which can be followed by the **norms** keyword and by a list of norms.

Each of the 2 lists is a 3D data set. The norms data must have the same structure as the nodes data: the number 2D data subsets must be the same, and each 2D data subset must have the same number of points as the nodes data, but the number of coordinates by point must be 3.

The default set of polygons is the same as the default triangle.

### 4.10.1 Example:

```
poly {
    { -6 1 -3
    -6 1 2
    0 1 6
    6 1 2
    6 1 -3
    0 1 -6 }
    { -4 5 -3
    -4 5 2
    0 5 4
    4 5 2
    4 5 -3
    0 5 -4 }
    color { 1 1 0 }
}
```

As explained in the beginning, the information concerning the nodes and norms of the set of polygons can be followed by any number of properties. The set of polygons also inherits from the DVpick_components class.

## 4.11 Rotation surface

Name field:

**rsurface**

A rotation surface is described by a 2D data set representing a line that will be rotated around the Oz axis. The line is supposed to be in the xOz plane, and if it isn't, only the x and z coordinates will be considered for the rotation. The 2D data set can be followed by the angle keyword and by the rotation angle.

### 4.11.1 Example:

rsurf {

<0 0 5> <1.2 0 5> <1.3 0 5.1> <1.3 0 5.4>

<1.2 0 5.5> <0 0 5.5>

angle 360

color { 0.3 0.18 0.1 }

xyz { 0 2 1 }

translate { 5.5 0 0 }

}

**Note**: The rotation surface points are calculated immediately after the information about the angle. Any property that is given afterwards does not affect the construction of the surface, but only the way it is drawn. If you want a different precision in the number of steps for the rotation, you should include the rotation surface in a node object, and set its geometric complexity before declaring the rotation surface. This doesn't work yet very well, but I'll check on it as soon as possible.

The rotation surface is derived from the triangle strips class, and so is also a subclass of DVpick_components.

## 4.12 Set of spheres

Name field:

**spheres**

A set of spheres is defined by a number of centers followed by a ray, and each of them is independently optional.

Each center is defined by 3 real values ( its coordinates) . If no center information is present, the set of spheres will contain one default sphere with the center:

0.0, 0.0, 0.0

If the ray information is not present, the default ray is 1.0. The ray applies to all the spheres in the set.

#### 4.12.1 Example:

spheres {

1 0 0

1.5 0 0

2 0 0

ray {0.5}

}

As explained in the beginning, the information concerning the centers and the ray of the set of spheres can be followed by any number of properties. The spheres class inherits from the DVpick_components class.

### 4.13 Surface

Node field:

**surface**

A surface corresponding to the DVsurface class is a complex object that needs several field in its description. So, the body of a surface must contain the following information: the point numbers, the points data, the norms data, the patches size, and the point numbers in each patch. The surface body can be empty, in which case the surface will contain nothing ( sorry, I was lazy on this part) . From all the items of the surface, only the norms data is optional. All the other items either do not appear at all, either they must all be given.

#### 4.13.1 Point numbers

The point numbers can be implicit or explicit.

The **implicit** point numbers are given only by their number, are are set by default to 0...number.

The **explicit** point number must be given one by one in an increasing order. To distinguish them from the implicit form, they must be enclosed in curly braces { } . If the numbers are not in order, the unordered part will be replaced by a default sequence. For example, the list of point numbers:

{ 1 3 4 6 7 0 0 0}

will be replaces by

{ 1 3 4 6 7 8 9 10}

Each number in this list corresponds to the point data with the same index (and not the same number !). For example, in the list above, the number '4' denotes the 3rd point in the point data list.

### 4.13.2 Points data

This part of the surface is a simple 2D data list, with the only difference that is enclosed by curly braces. Only one string of points is admitted here. This data will not be repeated afterwards, but will be accessed by the point numbers.

### 4.13.3 Norms data

This is the only optional item of the surface. If it is present, it must be preceded by the token norms and has the same as the points data: a 2D data list enclosed in curly braces. Note that if the norms are given, the surface will be smooth.

**Patch sizes**

This is a list of simple integer numbers enclosed in curly braces. The surface is composed of a number of patches, each being a polygon. Each number in this list represent the number of points in a patch.

### 4.13.4 Patch points

This is a special 2D data list, in which each point has its own length ( number of coordinates) . The numbers in this list are integers and each of them must also be found in the point numbers list. Each such list of integers must be sized according to the size of the patch defined in the previous item.

### 4.13.5 Specific properties

The surface has only one specific property, which specifies if the surface is smooth or not and only need the token **smooth** with no other argument. If the norms data is explicitly given, then this property has no effect.

The class DVsurface is also derived from the DVpick_components class and inherits its properties.

### 4.13.6 Example:

```
surface {
        12
        {<-6 1 -3 0>
        -6 1 2 1
        0 1 6 2
        6 1 2 3
        6 1 -3 4
        0 1 -6 5
```

-4 5 -3 6

-4 5 2 7

0 5 4 8

4 5 2 9

4 5 -3 10

0 5 -4 11 }

{ 6 6 4 }

{< 0 1 2 3 4 5 >

< 6 7 8 9 10 11 >

< 0 1 7 6> }

smooth

}

## 4.14 Texture quadrilateral

Name field:

**texture**

The texture quadrilateral is defined by 4 points with 3 coordinates each, and a file name containing a ppm image. If the 4 points are not specified, the default quadrilateral is set to:

<-1.0 -1.0 0.0 >

< 1.0 -1.0 0.0 >

< 1.0 1.0 0.0 >

<-1.0 1.0 0.0 >

The ppm image must be in a ascii rgb ppm . The width and height of the image must both be a power of 2 (like 256 / 1024), but the image does not have to be a square. Images of less that 64 / 64 pixels can cause problems.

If the proportion of the image is different than that of the quadrilater, the image will be scaled to fit in the quadrilateral.

#### 4.14.1 Example:

texture {

-4.0 4.0 0.0

-4.0 -4.0 0.0

4.0 -4.0 0.0

4.0 4.0 0.0

Moto.ppm

}

The DVtexture_quadrilateral class can only have simple properties.

### 4.15 Triangle

Name field:

**triangle**

The triangle can be set by default (an empty body : { } ) or be specified by the three nodes, by a norm or by 3 norms. According to this the body of the triangle can contain:

- nothing, in which case the default triangle is constructed:

  1.0 0.0 0.0

  0.0 1.0 0.0

  0.0 0.0 1.0

- 3 times 3 real values, representing its nodes,
- 4 times 3 real values, representing its nodes and one norm,
- 6 times 3 real values, representing its nodes and 3 norms.

#### 4.15.1 Example:

triangle {

0.0 0.0 0.0

1.0 0.0 0.0

0.0 1.0 0.0

}

As explained in the beginning, the information concerning the nodes and norms of the triangle can be followed by any number of properties.

## 4.16 Triangle grid

Name field:

**tgrid**

The triangle grid can be set by default (an empty body :{} ) or be specified by a list of nodes (at least 3), which can be followed by the **norms** keyword and by a list of norms.

Each of the 2 lists is a 3D data set. The norms data must have the same structure as the nodes data: the number 2D data subsets must be the same, and each 2D data subset must have the same number of points as the nodes data, but each points must have 3 coordinates.

The difference from the triangle strip is that the points must be rectangular, which means that each 2D data set in the triangle grid must have the same number of points.

The default triangle grid (corresponding to no data in the body) is:

{ 0 0 0

0 0 1 }

{ 0 1 0

0 1 1 }

### 4.16.1 Example:

tgrid {

{ 0 1 0

0 2 0

0 3 0 }

{ 1 1 0

2 2 0

3 3 0 }

{ 2 1 1

4 2 2

5 3 3 }

color { 1 0 0 }

}

As explained in the beginning, the information concerning the nodes and norms of the triangle grid can be followed by any number of properties. The triangle grid also inherits from the DVpick_components class.

## 4.17 Triangle strip

Name field:

**tstrip**

The triangle strip can be set by default (an empty body :{ } ) or be specified by a list of nodes (at least 3), which can be followed by the **norms** keyword and by a list of norms.

Each of the 2 lists is a 3D data set. The norms data must have the same structure as the nodes data: the number 2D data subsets must be the same, and each 2D data subset must have the same number of points as the nodes data.

The default triangle grid is the same as the default triangle.

### 4.17.1 Example:

tstrip {

0.0 0.0 0.0

1.0 0.0 0.0

0.0 1.0 0.0

0.0 0.0 1.0

0.0 0.0 0.0

1.0 0.0 0.0

}

This triangle strip builds a tetrahedron.

As explained in the beginning, the information concerning the nodes and norms of the triangle strip can be followed by any number of properties. The triangle strip also inherits from the DVpick_components class.

The triangle strip has only one special property, defined by the keyword "fan". This causes the list of points to be drawn as a triangle fan, in which each two points in the sequence, except for the first one, compose a triangle with the very first point in the sequence.

### 4.17.2 Example:

tstrip {

{0.0 0.0 0.0

1.0 1.0 1.0

-1.0 1.0 1.0

-1.0 -1.0 1.0

1.0 -1.0 1.0

1.0 1.0 1.0}

{1.0 1.0 1.0

-1.0 1.0 1.0

-1.0 -1.0 1.0

1.0 -1.0 1.0}

fan

}

This triangle fan builds a pyramid.

# 5 Properties

**Note**. Any property will be applied to the last object { geometrical or node) that has been created. If the last object that was created is a node object, the property will be applied to all the geometrical objects related to this node.

## 5.1 Contents



## 5.2 Color

Name field:

**color**

The body of the color should (but may not) contain 3 real values corresponding to the rgb values. If the rgb values are absent, the 1, 1, 1 values (white) are set by default. A 4th value can represent the blending (transparency) which is by default 1.0 (opaque).

Every object can have 3 color properties. Thus, a 5th value which must be an integer between 0 and 2, can indicate which of the 3 colors of the object will be set to this color. If this value is absent, the default color is the 0 one.

If you do not wish to set the blending, but want to set another color that the 0 one, you can put the rgb values between $<$ and $>$. For example,

```
color {<1 1 0> 2}
```

will set the 3rd color of the object to yellow.

## 5.3 Color coordinate

Name field:

**ccoord**

and it only applies to objects inheriting from the class DVpick_property.

This property sets the coordinate of the point that will be used as the color of the object in that point. The value of the color coordinate in each point will be used to compute the color from a color mapping. The minimal value of the color coordinate for the object will be set to the first color of the color mapping, and the maximal value to the last color of the mapping. All the other values are scaled in between.

### 5.3.1 Example:

ccoord { 5 }

If the body of the color coordinate is empty, then the color coordinate is set by default to 4. Note that at the creation of the object, this value is not set, and the object uses the general color property.

## 5.4 Color map

Name field:

**color_map**

The color map properties defines a color map and attaches it to the object inside whose body it is.

The color map is defined by a list of real numbers and a list of color. The list of real number can be shorter than the list of colors, in which case it will be completed by default as follows:

- If there are at least two numbers, the number list will be completed using thedifference between the last two numbers given by the user. For example, if there are 6 colors but only the list of numbers:

  0 0.2 0.5

  this list will be completed as:

  0 0.2 0.5 0.8 1.1

- If there is only one number the rest of the list will be completed by incrementing with 1.0 starting from this number. For example, the list:

  1.5

  will be completed as:

  1.5 2.5 3.5 4.5 5.5

- If the list of numbers is empty, it will be set by default to the sequence of integer numbers starting from 0. So, for 5 colors we would have the list :

  0 1 2 3 4

The colors in the color list have the same syntax as the color property, without the color choose (the 5th value in the color property) has no meaning and will be ignored.

If the color map property is empty, it will be set by default to the following color map:

color_map {

0.0 0.25 0.5 0.75 1.0

color {0 0 1}

color {0 1 1}

color {0 1 0}

color {1 1 0}

color {1 0 0}

}

### 5.4.1 Example:

color_map {

color {1 1 0}

color {1 0 0}

}

## 5.5 Coordinates setting

Name field:

**xyz**

and it only applies to objects inheriting from the class DVpick_property.

This property sets the indexes of the x y z coordinates of the drawn points, if the data points have more than 3 coordinates.

The body of this property can contain 3 values corresponding to the coordinates of the points which will be used for x, y and z coordinates of the image. If these values are not present, the coordinates are set to 0, 1 and 2 by default, which has no effect as they already have these values when the object is constructed.

### 5.5.1 Example:

xyz { 1 0 2 }

This example is just a change of base.

## 5.6 Geometry complexity

Name field:

**gcomp**

This property sets the complexity of the geometry for the object that contains this property in its definition. The body of the geometry complexity can contain an integer value from 1 to 9 signifying the complexity of the geometry. If this value is missing, the default value is 7.

## 5.7 Light complexity

Name field:

**lcomp**

This property sets the complexity of the light for the object that contains this property in its definition. The body of the light complexity can contain an integer value from 1 to 9 signifying the complexity of the light. If this value is missing, the default value is 7.

## 5.8 Rotation

Name field:

**rotate**

This property sets the rotation of the object. It has 4 parameters, the first 3 composing the rotation axis, and the forth being the rotation angle. A fifth optional parameter with the token **radians** can specify if the angle is given in radians. If this parameter is absent, the angle is by default in degrees.

If there are $0 <$ a number of real values $< 4$ in the rotation body, then they will receive the '0.0' value by default. As the values are set in order, then the angle will surely receive the 0.0 value. This means that the rotation will have no effect.

If no real value is present in the rotation body, then the 0.0, 1.0, 0.0, 90 values will be set by default. This means a rotation of 90 degrees around the 0y axis.

### 5.8.1 Example:

rotate { 0 0 1 1.5708 radians }

This is a rotation of around 90 degrees ( pi / 2 in radians) around the 0z axis.

## 5.9 Scaling

Name field:

**scale**

The body of the scaling should ( but may not) contain 3 real values corresponding to the xyz values. If the xyz values are absent, the 1, 1, 1 values ( no scaling) are set by default.

**Note**. The scaling only applies to object position, and does not affect their shape. The only exception is the lines object which has a ribbon. The length of the ribbon is scaled.

## 5.10 Translation

Name field:

**translate**

The body of the translation should ( but may not) contain 3 real values corresponding to the xyz coordinates of the translation. If these values are absent, the default null translation ( 0, 0, 0) is set.

## 5.11 Width

Name field:

**width**

The width property is a special one, because any object can have a width, but it effects are various from an object to another.

The width has two component, 0 and 1. The body of the width can contain a real value corresponding to the width and a a second argument representing the location value telling which of the components of the width will be set. If the first value is missing, it is set by default to 1.0. If the second value is missing, it is set by default to 0. The second argument can not appear without the first one.

### 5.11.1 Example:

width { 1.2 1}

In this example, we set the second width of an object to the value 1.2.

For the **axes** object, the first component of the width represents the width of the line, in number of pixels. The second component scales the length of the 3 directions.

For the **framed lines** and the **lines** objects, the first component of the width represents:

- the lines width ( number of pixels) if the line status of the object is DV_LINE,
- the cylinder ray if the line status of the object is DV_CYLINDER.

The second component of the width represents the ribbon width if the ribbon status of the object is D-V_RIBBON. For the framed lines, this value scales all the ribbons of the object.

For the **points** object, the width defines the number of pixels by which each point will be drawn.

For the **spheres** object, the first component of the width is the ray of the spheres. The second component of the width has no meaning.

For the **boxes** object, the width will soon scale the z coordinate, and can be used when the z coordinate is missing ( data has only 2 coordinates) and the z coordinate is alternatively set to 0 and 1.