11 11 TEX LATEX

DataViewer's Developers Reference Manual

Randy Paffenroth redrod@acm.org v0.2, Mon Dec 13 17:23:23 CET 1999

This is the developers reference manual for DataViewer. It provides a general description of the implementation of the package. It also contains a reference to all the classes in DataViewer More information on this software may be found at *http://lcvmwww.epfl.ch/DV*.

Contents

1 Introduction

DataViewer is a scientific visualization library created to allow the easy development of efficient visualization programs for those who are not graphics programming experts. It consists of a high-level set of routines for rendering geometric objects and utilizes the OpenGL graphics API for its low level rendering. DataViewer has been used for several different visualization tasks, e.g. most of *VBM*'s visualization modules have been developed using it. The goals that have driven the design of DataViewer are to be efficient, easy to use, portable, closely connectable to numerical computations, and available free of charge to the mathematical and scientific community.

1.1 License

1.2 Implementation

DataViewer2.0 is written entirely in C++, and uses many of the language's object oriented features, such as templates and virtual functions. Each class in DataViewer2.0 inherits from a single base class called

DVobject. Each DVobject defines a *draw* routine which canonically contains a set of OpenGL commands that create the graphical representation of the object. In addition, each DVobject contains a set of properties, such as color, translation, and rotation. These properties will be discussed in greater detail below.

Python gives access to the Tk widget library, which DataViewer utilizes for the design of GUIs. Tk is a library of widgets which can be accessed from a wide variety of languages, including Tcl, Python, C/C++, and Perl. It is a robust and mature code which has implementations on all Unix systems as well as Microsoft Windows. In our opinion, Tk is much easier to use than Motif and makes the GUI more flexible. With Tk and Python it is quite easy for the user of DataViewer to create custom GUIs for their problems.

Finally, DataViewer2.0 provides several features which are accessible in any application which uses the library. For example, DataViewer provides stereo viewing capabilities, using hardware such as Crystal Eyes and access to six degree-of-freedom controls, such as Logitech's Magellan.

2 History

Many software packages and algorithms have been developed for the rendering of three dimensional graphics on modern computers. Only certain of these algorithms are of interest in our work because of our interactivity requirement. For example, ray-tracing performs well under many circumstances, but is too slow for our purposes.

Commonly, interactive graphics require some type of hardware assistance for the rendering process. Geometry rendering is notoriously slow, but since it is a repetitive process it can be efficiently implemented in hardware. Many companies produce hardware accelerators for graphical rendering, but each accelerator tends to use its own proprietary library for accessing the hardware. Obviously, none of these libraries are appropriate for the development of general visualization software, since the software would only be usable on the specific hardware platform.

Many attempts have been made to place software layers on top of the specific hardware implementations to make applications portable. These libraries define an API, and software vendors are expected to provide implementations of this API for their hardware. Today, the situation has become clearer, and it appears that OpenGL is the preferred solution on Unix systems and either OpenGL or Direct3D is the preferred solution on Microsoft Windows. These libraries offer the advantage that an application may be developed using them, and this application will then be portable to any hardware which has an implementation of the appropriate API.

To achieve portability and maintain efficient access to hardware renderers, Paffenroth implemented the first version of DataViewer (DataViewer1.0) using OpenGL. Several useful applications were developed using DataViewer1.0, but a few problems were noticed with this approach. First, the development of DataViewer1.0 applications required the user to develop OpenGL code themselves. OpenGL is very general, but quite low level. It only provides simple geometrical primitives (such as lines and triangles), and more high level objects must be implemented separately. Accordingly, writing applications directly in OpenGL can be complex and time consuming. Second, the interface in DataViewer1.0 was written in Motif, a graphical user interface (GUI) library for Unix. Motif, as OpenGL, is very general but low level. Thus, it was not easy to develop custom GUIs in DataViewer. Also, Motif is proprietary software, and even though it is widely available, there exist Unix based systems for which it is either not available or only available at a significant cost.

We decided that our needs called for some type of layer on top of OpenGL, or some similar library, to provide a higher level of abstraction and which would be easier to use. We examined the libraries and packages that were available in late 1996, and found several candidates. Unfortunately, none of the available software exactly met our needs.

Hence we were left with the prospect of creating our own software library. DataViewer2.0 was developed by R. Paffenroth, T. Stone, D. Vrajitoru and A. Ahearn; it is our attempt to create a library which encompasses all of our needs, based on our experiences in the development of DataViewer1.0.

3 General notions

In this section we will describe the basic organization of a DataViewer scene.

DataViewer allows one to easily display and interact with data on a computer screen. DataViewer can display, rotate, zoom, etc., one image, or use animation to display multiple (time–dependent) images.

Currently DataViewer allows images to be made up of the following basic graphics shapes: spheres, polygons, lines, cylinders, and cones. It also provides some basic use of texture mapping.

More specifically, DataViewer (currently) has the following graphical types:

- DVaxes
- DVboxes
- DVcones
- DVframed_lines
- DVlines
- DVpolygons
- DVquadrilateral
- DVrotation_surface
- DVspheres
- DVsurface
- DVtextureQuadrilateral
- DVtriangle
- DVtriangleGrid
- DVtriangleStrips

More types may be added later.

3.1 The basic object

All the graphical classes in DataViewer are derived from one class, **DVobject**.

This class defines a number of properties common to all the objects in DataViewer. The translation and rotation properties are directly defined in the DVobject class. As DVobject is derived from the **DVselect-Property** class, all its properties are available in any DataViewer object:

- **color** describing a direct color in RGB terms; has 2 components, the second being used as the ribbon color by DVlines
- **width** describing an object width; has 2 components, and each object can chose the way to use it
- **geometry_complexity** from 1 to 9
- **light_complexity** from 1 to 9
- **scale** a scaling with x, y, and z values, does not deform the spheres and cylinders
- **color_by_data** sets one of the points coordinates to be the color; is related to color_mappings
- **lights** add lights to the object
- **material** defines the properties of the material

All these properties will be described in more details further in the document.

3.2 The scene graph

The most important feature of DataViewer2.0 is its scene graph. In a scene graph, a geometrical object is represented in a hierarchical fashion using a tree. A given object is divided into pieces, each of which is either a *leaf node* (i.e. a geometrical object which can be represented as an atomic data type) or a *group node* which itself contains other nodes. DataViewer2.0 implements a scene graph by using two main types of nodes.

The first type of node in DataViewer2.0 is the *geometry* . Geometry nodes are the leaf nodes of DataViewer2.0 and they represent geometrical objects. DataViewer2.0 differs from most libraries that use scene graphs in that DataViewer2.0 not only provides simple leaf nodes, but also leaf nodes which are quite complex geometric objects.

The second type of node in DataViewer2.0 is the *container* node. Container nodes group together other nodes into a set, which may be used as a single node. In a container node, the other nodes which are contained within the container node are known as its *children*. Similarly, the container node itself is known as the *parent* of the nodes which it contains.

DataViewer divides container nodes into two classes. The first class is the *normal containers*, with the property that each time the draw function of the parent is called the draw functions of all of the children are called.

The second class is the *selection containers*, with the property that each time the draw function of the parent is called only some of the children's draw functions are called. Generally, the selection containers are used for special affects, such as animation and level-of-detail.

4 Common routines

This section describes the set of routines and operators that can apply to any class in DataViewer, be it a geometrical class or a simple property class.

4.1 Any DataViewer class

Let `DVclass` be any class in DataViewer. Then we can define the following routines:

The default destructor. It is declared virtual so that the correct descendant destructor will get called.

```
virtual ~DVclass();
```

A routine that returns a pointer to a copy of the object created with new.

```
virtual DVclass *copy();
```

The copy constructor and the assignment operator.

```
DVclass(const DVclass &data);
const DVclass &operator=(const DVclass &data);
```

Operators to check for equality and for non equality.

```
int operator==(const DVclass &data);
int operator!=(const DVclass &data);
```

Virtual print function so that we can print instantiations of the various classes derived from DVclass.

```
virtual void print(ostream &);
```

A friend routine to overload the << operator.

```
friend ostream &operator<<(ostream &, DVclass &);
```

4.2 General geometrical routines

The draw routine that uses the geometrical stack must be defined by each geometrical class. It has to be virtual so that the correct descendant routine gets called.

```
virtual void draw(LCVMstack<DVproperty*>&);
```

4.3 Complex geometrical classes

This paragraph concerns the geometrical classes corresponding to data organized in a matrix of several lines and columns, or even several matrices of various sizes. The geometrical objects in this category inherit both from the DVobject class and from the DVpick_components class. This category includes the following classes:

- DVboxes
- DVcones
- DVframed_lines
- DVlines
- DVpolygons
- DVrotation_surface
- DVspheres
- DVsurface
- DVtriangleGrid
- DVtriangleStrips

4.4 Container routines

All the containers must know how to add and to delete a child from their list of descendants.

```
DVobject &copy_child(DVobject &child);
DVobject &delete_current_child();
```

4.5 LCVMarray classes

In general, a LCVMarray is a class containing a normal array together with information about its size. All the LCVMarray classes are instantiations of the LCVMarray template class, or of other template classes derived from it.

Let `T` be the basic class for the LCVMarray template class. The most important methods that can be applied to this class are:

Constructors

The constructor with a given size, which is 0 by default.

`LCVMarray(const int = 0)`

The constructor with a given size, and initialized with a value of the basic class.

`LCVMarray(const int, const T&)`

Copy constructor.

`LCVMarray(const LCVMarray&)`

Copy constructor from a normal array of the basic type. It needs the size of the array.

`LCVMarray(const T* const, const int)`

Set the data in the array

For each constructor, there is a similar set_data method, with the same arguments. These methods first delete the existing array and the data it contains, then initialize the object again with the new data.

Without reallocating the array contained in the object, you can initialize each component of the array with the given value:

`LCVMarray (const T&)`

The data in a LCVMarray can be deleted (but not the object itself) with the following method, returning the object itself after the operation:

`LCVMarray ()`

Extract data from the array

To know the size of such an array, use the following method:

`int get_size() const`

The elements of a LCVMarray can be accessed as the elements of a normal array with the `[]` operator:

`T [](const int)`

or with the following method:

`T get_element(const int) const`

Two methods can copy the data in the LCVMarray to another LCVMarray:

`void get_data(LCVMarray&) const`

or to a normal array:

`void get_data(T*) const`

For more information on each specific LCVMarray class, see the 6 (class reference).

4.6 Scene file parser methods

In principle, the scene file parser is organized in the following way:

The input file

The scene file is opened in a global variable, managed by the `scene_data` module. No other module should have access to it.

The function

`char *GetNextWord();`

moves the input file pointer to the next word in the file, skipping all the blanc characters and the comments. Separators like { are considered entire words.

The function

`char *TheWord();`

returns the word in the input file at the current position.

Tokens

There is a set of tokens defined in the `scene_tokens` module. The function

`int CheckToken(char *aString, TokenName aToken);`

verifies that the given string represents the specified token.

If a new token is added to the file parser, all the modules that use tokens have to be recompiled. We recommend to do a `make clean` and then recompile everything again.

Constructors

Each geometrical class and several other classes have constructors taking a file option as input. The normal constructor from the global input file with a file option should receive the file pointer at the beginning of an object in the file of its class type. Usually the container constructor from a file option calls the appropriate constructors according to the current token in the file. The class creates an instance of itself using the information in the file, and then return the file pointer at a position immediately after the object.

5 DataViewer applications

This section introduces the basic notions to run and to create a DataViewer application.

5.1 Running a DataViewer application

To run a DataViewer application, you should run the following command:

```
DV.py python_script.py
```

The first argument is the name of the DataViewer executable. If it is not in the current directory, it must be preceded by the path. For example, if we were in the directory `~/Dataviewer/Python_demo_simple` , instead of `DV.py` we should type `/bin/DV.py`. The executable of DataViewer always has this name.

The second argument is the name of a Python script which contains the object (scene) we want to view with DataViewer. For example, from the same directory, we would type `/python_script/typein.py`. This argument must only end with `.py`.

For more details on running DataViewer, see the *DataViewer's User Guide* `<http://lcvmwww.epfl.ch/DV/documentation/>`.

5.2 Writing a DataViewer application

A DataViewer application consists in a Python script and a c++ function that communicate by means of strings.

The Python script

Write a Python script defining a class derived from GeomWindow. In this class, you must define two methods:

- `UserWidgets`

In this methods, you must declare the widgets that are necessary for your problem. These widgets will be included in a dialog window that is activated from the File/User menu.

- `Draw`

In this method, you must specify the c++ problem the following way:

`self.geom.set_problem("problem_name")`

and you must send all the information needed by the c++ function with

`self.geom.file("some data string")`

The c++ function

Write a c++ function that constructs the scene graph for your problem.

```
     int problem_function(DVcreate_image_arguments )
```

This function must always return the `int` type, and must always take an argument of type `DVcreate_image_arguments`.

Two things are important about this argument:

- `dv.input_string` which contains the string send by the Python script with the `self.geom.file` call;
- `dv.top` of type `DVcontainerbase`, which is the root of the scene graph, so that any geometrical object you create must be copied to it, or recursively copied to a container that will be directly or indirectly copied to it.

Finally, to link the Python script with the c++ function, you must proceed the following way:

- Add the prototype of the c++ function (`problem_function`) to `DVpython_problems.h`.
- Add the name of the problem as called in the Python script (`problem_name`) to DVpython_problems.cc, in the function `DVpython_problems`:

```
          if(strcmp(input_problem,"problem_name")==0)
            set->set_user_function(problem_function);
```

- Add the name of the c++ file containing your problem to the `Makefile`. If you have taken one of the existing applications as example, you must add your problem file to `$PROBLEM_OBJECTS`.

6 Class reference

The class reference is automatically generated from comments in the header files.

It can be found at the followiong address: *Online reference manual* `<http://lcvmwww.epfl.ch/DV/documentation/>`