The Concepts of VISH

Werner Benger[1] and Georg Ritter[2] and
René Heinzl[3]

[1]Center for Computation and Technology, Louisiana State University, USA

[2]Institute for Astro- and Particle Physics, University of Innsbruck

[3]Institute for Microelectronics, TU Wien, Vienna, Austria

werner@cct.lsu.edu, georg.ritter@uibk.ac.at,
heinzl@iue.tuwien.ac.at

May 15, 2007

Abstract

VISH is a novel application interface aiming at the separation of algorithm implementation from the software environment they run in. It provides different layers of abstraction to shield algorithms from application-specific details. On the coarsest level, these are generic objects with parameters, on the finest level, this is a concrete model for scientific data covering a wide range of data types. Special attention is given to algorithms for scientific visualization. The objective is to have algorithms only implemented once and share them, together with the data, among applications, even in binary form. This paper presents the concepts and current implementation status in C++.

Contents

1 Introduction

1.1 Motivation

Although Scientific Visualization is an established field, the gap between algorithm development and accessibility of new visualization techniques to a wider audience, especially application scientists as end-users, is still a frequent hurdle. New techniques are either implemented as minimal standalone versions using libraries such as GLUT or as a plugin to a larger, sometimes proprietary, software environment. Both methods serve well for the development process, but complicate the wider deployment of the algorithms because they are bound to their runtime environment. Therefore they do not necessarily integrate well into the daily work flow of an application scientists due to limitations of that specific environment for his own purpose.

We envision a software environment that allows to develop and implement visualization algorithms with

- independence from a specific application; ideally, such that even precompiled algorithms can be shared among applications and users;
- minimal overhead on dependencies and external components;
- easy deployment to end users, i.e. support for application-specific file formats and user-friendly GUI without impacting the visualization algorithm itself;
- accessibility to all levels of interfacing hardware;
- high reusability of algorithms, in particular also including I/O layers (support for diverse file formats, remote data access etc.).

This may be achieved via some visualization microkernel that may serve as an "operating system for visualization algorithms". A minimal abstract API shall serve as a framework for development and allow sharing of plugins throughout applications, which provide implementations of the same interfaces.

This is the vision of VISH as a "visualization shell" ("shell" in the sense of a "structural work" or "skin"). It is not an another application by itself, but an implementation of the infrastructure necessary for scientific visualization and

therefore a collection of interfaces in the form of libraries. The implementation of such interfaces (such as an input method for an integer) is left to a specific application environment. Algorithms will just see the VISH API (such as to formulate the request for an integer value) and remain application independent.

VISH encompasses different levels of abstraction: the VISH kernel (section 2) provides only interface functionality for general purpose objects with parameters. A data model for scientific visualization is provided by the FiberLib2 (section 3) and complements the VISH kernel. While the VISH kernel basically abstracts event handling and data flow, the FiberLib2 functionality also allows to handle and share data sets among applications. Both components can be used independently, but are designed to integrate well with another, thus forming the FiberLib2-VISH (or shortly "FISH") environment (see Fig. 1 for a depiction of the relationships of the VISH components).

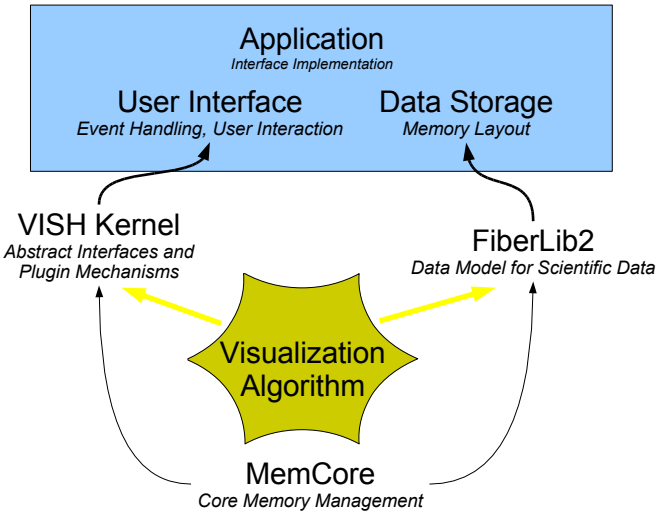

Figure 1: Based on the memory management facilities of MemCore, the VISH and FiberLib2 libraries provide abstraction interfaces for object operations and formulating scientific data, respectively. A newly implemented visualization algorithm would only communicate with these layers, whereas the concrete application behind builds on them or implements the interfaces.

1.2 Previous Work

The IBM Data Explorer [Treinish, 1997] has a long history as a tool for scientific visualization. Its design concepts (in particular its data model based on fiber bundles) were so well founded, that it is still actively used even though its implementation concepts are outdated and active development has ceased.

Both Amira [Stalling et al., 2005] and Ensight [CEI, 2007] are widely used visualization environments, but are proprietary, whereas the open source model offers many benefits to both academia and industry, and to both researchers and end users [Johnson et al., 2006]. The visualization toolkit [Kitware, 2005] is a well known open source collection of algorithms, but is far from being a microkernel. SciRun [SCI, 2007] is a recently released open source solution, but a complex application by itself and quite intrusive for visualization algorithms. GLUT[1] is a convenience library based on OpenGL frequently used to demonstrate proof-of-concept, but is not aiming at providing end user friendly applications.

2 The VISH Kernel

The VISH kernel is the API that plugins and application code see. All interactions from application code and with algorithms are through this API. This kernel itself consist of a collection of libraries which are contained in a common folder called "`ocean`" (the ocean is what is required to let fish swim). The main library is the "`plankton`" library (plankton are the smallest animals in the ocean and are essential to nourish fishes). Dependencies of the `plankton` library to external libraries are kept to a minimum, the only requirement is the "`memcore`" library that provides reference pointer and similar functionality in a generic way. Another component of the VISH kernel is the GLvish library, which adds objects with OpenGL rendering capabilities to the VISH kernel. The `vscript` library implements one possible scripting interface to the VISH objects; other scripting languages such as tcl or python are possible as well. However, these libraries will not be discussed here.

2.1 MemCore Functionality

The MemCore library implements means for automatic dynamic memory management. Similar to e.g. the Boost smart pointers [Colvin, 1994], it supports the concept of weak and strong pointers (implementation details are given on p.81 – p.83 in [Benger, 2004]). In addition, these pointers allow implicit up- and downcasting within the same class hierarchy, thereby enabling very compact code when querying objects at runtime such as in:

```
struct A {};
struct B : A {};

void f(const RefPtr<A>&a)
{
      if (RefPtr<B> b = a ) { /* it's a B object*/ }
}
```

In the MemCore library, any strong pointer is automatically a weak pointer as well. As a consequence, if the referenced object is destroyed explicitly, then

[1] `http://www.opengl.org/documentation/specs/glut/spec3/spec3.html`

all strong pointers become invalid. Thus, reference pointers may as well point to automatic, static or dynamically allocated objects that are deleted explicitly.

Another feature of the MemCore library are *typemaps*, that allow to associate objects with C++ type information. The resulting objects can then be indexed using `typeid()`:

```
std::map<type_info, string>  TypeName;
TypeName[ typeid(int) ] = "Integer";
```

In practice, the `type_info` type is not directly suitable as a key to STL maps and some intermediate class is employed. Note that only some ordering relationship among type information instances is necessary. This is provided natively by any C++ implementation and guarantees a bijective association in a platform-independent way.

Typemaps are useful to enable an dynamic interface mechanism to objects, similar to the interface concept in Java as an alternative to multiple inheritance. The MemCore library allows to add and remove interfaces to a base class "Intercube" (inspired by the notion that a cube has many faces) at runtime. It is basically a type map that associates a type domain to some interface object. Interface objects are reference counted objects that define such a type domain; the schematic structure looks as follows:

```
struct  InterfaceBase // reference countable
{};

template <class X> struct Interface : InterfaceBase
{
   typedef X InterfaceDomain;
};

struct Intercube
{
     typemap<RefPtr<InterfaceBase> > interfaces;

  template <class X>
     void addInterface(const RefPtr<Interface<X> >&InterfacePtr)
     {
        interfaces[ typeid(X) ] = InterfacePtr;
     }
};
```

To retrieve the interface of a certain object, the MemCore library provides a syntax that is similar to the native C++ pointer conversions:

```
struct  A : Interface<A>
{};
Intercube Object;
    Object.addInterface( new A() );

RefPtr<A> aptr = interface_cast<A>( Object );
```

Dynamic interfaces are very useful to allow plugins to add properties to existing objects that are managed in a central library.

One `Intercube` object can hold an unlimited number of interfaces, but only one interface per domain. These interfaces may well be of different type and stem from another class inheritance, as long as they refer to the same interface domain:

```
struct  SecondA : Interface<A>
{};
RefPtr<A> aptr = interface_cast<A>( Object );
RefPtr<SecondA> aptr2 = interface_cast<A>( Object );
```

In this example, the `aptr` will be invalid, whereas the `aptr2` will be a valid pointer to an object derived from the same interface domain.

Note that in the following section we will use native pointers in the code examples for illustration, though in the actual implementation only strong and weak pointers are employed.

2.2 The Plankton Library

2.2.1 Object Management in VISH

The VISH `plankton` library provides a common pool of "managed objects" (class `VManagedObject`). They are contained in a global associative container that allows to address each object via some type ID (the managed object's *domain*) and a user-defined unique text. Schematically, the pool of managed object is of a signature such as (using the STL standard map):

```
map<type_info, map<string, VManagedObject*> >  ObjectPool;
```

An arbitrary object may now be retrieved via some type information, and an arbitrary name:

```
VManagedObject*MyObject = ObjectPool[ typeid(Domain) ][ "Name"];
```

An example for managed objects are Creators. Their constructor inserts them automatically into the ObjectPool using a "Creator" type domain. Within this domain they may be accessed via a a unique textual identifier. They may well be implemented as static objects within a dynamic library, such that this Creator object is visible in the ObjectPool once the library is loaded. Upon unloading the dynamic library, when the Creator's destructor is called, it will automatically become invisible in the ObjectMap due to the use of MemCore's reference pointer scheme. The base class of these Creator object comes with a virtual function that allows to create a certain category of objects, which are introduced in the next section.

2.2.2 Objects, Parameters and Inputs

The basic instance in VISH are abstract objects that may perform some operation based on some input parameters and may serve as input themselves; these

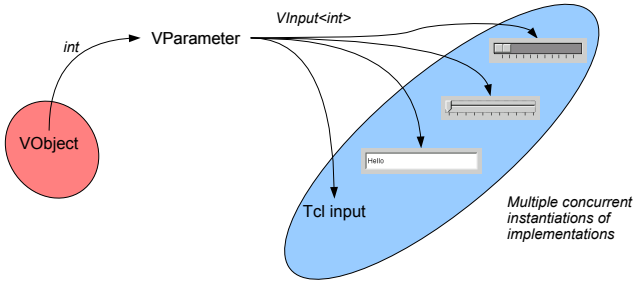

Figure 2: `VObject`s request `VParameter`s via intrinsic C++ types, which are implemented via one or many `VInput`s.

are called `VObject`s. VISH provides means to expose these input parameters, equip them with appropriate input objects, and to connect or share them with other `VObject`s. The relationship among objects, their parameters, and inputs of them defines the core functionality of VISH (see Fig. 2 for illustration):

- A `VObject` is an abstract class that implements some functionality controlled by input parameters. Its results are provided as output parameters.
- A `VParameter` provides means to retrieve numerical values. Multiple `VObject`s may refer to the same `VParameter` such that they can be easily coupled (for instance, different visualization algorithms operating on the same slice of the same volume, but different data fields).
- A `VInput` implements an actual numerical value and the means to modify it. A typical example is a slider within a graphical user interface. Alternative representations of the same value are desirable, such that a problem-specific representation can be chosen (e.g. a dial widget instead of a slider). Since such a representation may well depend on the context, also multiple representations of the same value are allowed (all referring to the same `VParameter`).

Considering these requirements, we may illustrate the central VISH classes schematically via native C pointers (note that the integer data type is only used for exemplification, the actual code uses weak and strong pointers on on abstract type-independent base classes):

```
struct VObject     // some VISH object doing something
{
     int **red;    // a parameter to the VObject

     void update() // the object's operation
     {
         printf("My red value is %d\n", **red);
     }
};
```

I.e., the numerical value of the object is not contained in the object itself, but only retrieved indirectly. The `VObject` needs to dereference the pointer twice to get the actual value. This indirection allows multiple objects to share the same value:

```
VObject A, B;
   A.red = B.red;
```

The *double* indirection allows to change the implementation of a certain value for all instances simultaneously:

```
int a, b;
int *value;
VObject A, B;

   B.red = &value;
   A.red = B.red;
   value = &a; // drive A and B objects via value a
   value = &b; // drive A and B objects via value b
```

VISH generalizes this basic idea through abstract objects. They are template instantiations over arbitrary C++ types and are derived from abstract base classes. Conceptually, the `int`s in the above example correspond to the `VInput` objects, and the first indirection, the `int*` to the `VParameter`s:

- Both input and output parameters of `VObject`s are bound to an intrinsic C++ type.
- A specific `VObject` may
  - request `VParameter`s, each such request is specified via a C++ type info plus an associated text. Such *input parameters* may be shared among `VObject`s.
  - provide `VParameter`s, which are then bound to this certain object. It is the duty of the respectively owning `VObject` to update those output`VParameter`s with the correct numerical values. Such *output parameters* may well be used as input parameters of other `VObject`s. This relationship among `VObject`s thus forms a graph among `VObject`s, called the *data flow* graph since it corresponds to the flow of data among `VObject`s.

- One `VParameter` may be represented by one or many `VInputs`. If one `VInput` is modified, then its corresponding `VParameter` will be notified about this change and forward this notification to all other associated `VInputs`. Thus different representations have the means to update themselves to display the changed value. The relationships among `VParameter` and `VInput` forms a graph, which is called the *control flow* graph since its values controls (parameterizes) the behavior of the `VObjects`.

The control flow graph is executed synchronously to the user interaction. In contrast, the data flow graph is only traversed when data are requested by some *data sink* (a `VObject` that resides at the end of the data flow graph, i.e. it has only inputs, but no outputs), i.e. asynchronously to user interaction. For instance, within a graphical display, such a data sink may be implemented by some OpenGL viewer that requests new data to be updated at 30 frames per second, or it may be some `VObject` which saves data to a file.

```
struct VInput
{
    VParameter*owner;
    int value;
};

struct VParameter
{
    VObject*owner;
    std::vector<VInput> inputs;
    int getValue();
};

struct VObject
{
    map<std::string, VParameter*> inputParams;
    std::vector<VParameter>       outputParams;

    void update();
};
```

Figure 3: Schematic structure of the relationships among VObject, VParameter and VInputs.

In the data flow model, VISH implements the push model (the data sink determines traversal of the data flow graph), like VTK [Kitware, 2005], not the pull model (the data source drives traversal through the data flow graph), like Amira [Stalling et al., 2005]. [2] Only those data that are supposed to be displayed (in a visualization context) are to be loaded on-demand from the

[2] TODO: Find more references and examples here, relate to literature, such as `http://www.usenix.org/events/sruti05/tech/full_papers/duan/duan_html/node4.html`

disk, for instance one time slice out of an evolution of a dynamic data set. As a drawback, a data set that is not yet displayed is also not yet loaded from disk, i.e. it cannot be analyzed by modules which are not related to an active data sink that pulls the data. It left to the application using the VISH/plankton library to decide which data sink is active and when.

2.2.3 Data Flow vs. Control Flow

The relationships among `VObject`s implement the data flow graph, which might involve computationally intensive operations and large data transfers. In contrast the control flow graph is implemented by the various `VInput` objects, that are intended for user interaction and is thus to be traversed immediately once an user input event occurs. The control flow graph will thus be executed *synchronously*, whereas the data flow graph is executed *asynchronously*, i.e. at some later time when actual data are requested. Such a data request may be driven by a time signal, that feeds some OpenGL renderer multiple times per second.

In both case, `VParameter` objects serve to link the various nodes within the data flow graph, the control flow graph, and the interaction among both types of graphs.

2.2.4 Context-relative Value Retrieval

Given a certain `VParameter`, its value may be retrieved relative to a certain context. Such a context provides a pool of variable values, each of them (a "shadow" of the global value) is associated with a `VParameter`. Schematically the code to retrieve a value then looks like this:

```
void example(ValuePool*Context, VParameter*param)
{
int value = param->getValue(Context);
    printf("Value in pool is %d\n", value);
}
```

An application case are multiple instances of 3D viewers in a graphical environment: the value of a parameter may be different for each viewer (each viewer provides its own pool of variables). Most important case will be the location of the observer such that different viewers may show different views. The concept of context-relative value retrieval is a generalization allowing to extend this functionality to arbitrary parameters. Different viewers may thus display different time steps of an evolution.

Alternative to the explicit call of a member function `getValue()` the overloading of the shift operators provides a more compact syntax:

```
void example(ValuePool*Context, VParameter*param)
{
int value;
```

```
    param << Context >> value;
    printf("Value in pool is %d\n", value);
}
```

hereby indicating the coupling of a parameter with an intrinsic variable via some context.

2.2.5 Slots

Within a certain VObject, VParameters are retrieved by a textual description. Since string lookup is expensive, an intermediate class VSlot is provided that refers to a named VParameter. Remembering fig. 3, a VSlot incorporates

```
map<std::string, VParameter*>::value_type
```

and thus can be seen as an additional (but mostly invisible) indirection:

```
class VObject
{
int ***red;
};
```

This additional indirection not only allows to share different input representations of the same parameters, but also of sharing parameters themselves among VObject. [...] (good explanation yet missing)

Given the "<<" and ">>" operators, the context-relative value of a slot can be evaluated. The return value of this operation is a boolean that tells if the certain variable was existent in the given context. The return value may be ignored if this existence is not important, e.g. because the value has a meaningful default. In such a case, the retrieval operators will leave the value untouched.

```
Slot  mySlot;
Context myContext;
int Value;
   if ( mySlot << myContext >> Value )
 {
       printf("Yes, got a value, and its value is %d!\n", Value);
   }
   else
 {
       printf("Sorry, could not retrieve the desired input value...\n");
   }
```

2.2.6 Input Creators

An input creator is a static object that allows to create inputs for a certain type.

This type can well be an arbitrary class. Let us exemplify the functionality of an input create on behalf of a practical case from qVISH. Here, a new type `qVTimerWindow` is defined in a library common to all plugins – this is

the qvish/aqua library; all qVISH plugins will link against this library. This qVTimerWindow defines an abstract interface, in this very case its purpose is to draw a graphical representation of a time slider (which by itself will not be discussed here):

```
struct qVTimerWindow : QFrame
{
   qVTimerWindow(QWidget*parent);
   virtual ~qVTimerWindow() = 0;
};
```

Associate with this type we require a derived input creator which allows to forward constructor arguments from a virtual create() function:

```
struct qVTimerWindowCreator : VInputCreator<qVTimerWindow>
{
   qVTimerWindowCreator(const string&name,
                        int prior=VManagedObject::DEFAULT_PRIORITY,
                        const RefPtr<VCreationPreferences>&prop = NullPtr() )
   : VInputCreator<qVTimerWindow>(name, prior, prop)
   {}

   virtual qVTimerWindow*create(QWidget*parent) = 0;
};
```

Then, application code will use the input creator's findBest() function to query VISH for an creator for the very type:

```
QWidget*parent;
qVTimerWindow*timerwindow = 0;

RefPtr<qVTimerWindowCreator>
       TimerWindowCreator = qVTimerWindowCreator::findBest()
       if (TimerWindowCreator)
       {
        timerwindow = TimerWindowCreator->create(parent);
       }
```

Now the *implementation* of the qVTimerWindow interface is done in a runtime plugin, in this case the qvish/chron module. This one implements a graphical representation of the time slider. It is derived from the qVTimerWindow interface class, its details and additional members plus the actual code doing the GUI is not shown:

```
class TimerWindow : public qVTimerWindow
{
public:
    TimerWindow(QWidget*parent);
};
```

What is left is providing an input creator for the interface type to VISH . This is done by deriving from the interface creator class in order to create a new instance of the interface implemention in the virtual `create()` function:

```
struct MyTimerCreator : qVTimerWindowCreator
{
    MyTimerCreator(const string&name,
                   int prior=VManagedObject::DEFAULT_PRIORITY,
                   const RefPtr<VCreationPreferences>&prop = NullPtr() )
    : qVTimerWindowCreator(name, prior, prop)
    {}

    qVTimerWindow*create(QWidget*parent)
    {
       return new TimerWindow(parent);
    }
};

static MyTimerCreator MyqVTimerWindowCreator("TimerWindow");

VISH_DEFAULT_INIT
```

The input creator function of VISH will now provide `TimerWindow` objects when an input of type `qVTimerWindow` is requested. Note that multiple such input creators may peacefully co-exist. Which one is chosen will be random, as all have to conform to the interface specification. However, `VCreationPreferences` may be used to give a preference to a certain creator over another one.

2.3 Eagle

The eagle library is an independent library implementing operations on small arrays of fixed size, vector spaces and geometric algebra. It is delivered with the VISH kernel and used in the context of the GLVish library (section 2.4).

The relationships among small arrays of fixed types, vector spaces and multivectors (the fundamental elements of geometric algebra) form a template class hierarchy. All instantiations over the same base type share the same memory layout, but provide different algebraic properties. Accompanied with these types is a type trait class that allows to described these properties for an arbitrary type.

2.4 GLVish

3 The FiberBundle Data Model

The fiber bundle data model is a generic scheme to cover a wide range of scientific data types through a specific data structure that is inspired by the mathematics of fiber bundles. It draws upon concepts of differential geometry and topology. The original ideas have been laid out by [Butler & Pendley, 1989] and later refined by [Haber et al., 1991]. While many modern visualization environments do not implement a data model at all and rather implement the various data types on an ad-hoc basis (with more or less random overlap of properties), the IBM Data Explorer (now OpenDX) [Treinish, 1997] has successfully proved the benefits of the fiber bundle data model. The model developed in [Benger, 2004] has extended the concepts found there in order to further systematize and reuse the concept of a fiber bundle.

The `FiberLib2` is a new implementation of the fiber bundle data model from [Benger, 2004]. Fundamental to it are the six hierarchy levels `Bundle, Slice, Grid, Topology, Representation, Field`. Given one hierarchy level, the next one is accessed via some identifier that is specifically appropriate for this level:

hierarchy object	identifier type	identifier semantic
`Bundle`	floating point number	time value
`Slice`	string	grid name
`Grid`	integer set	topological properties
`Topology`	pointer	relationship map
`Representation`	string	field name
`Field`	multidimensional index	array index

Only two of these identifier types are strings, and of arbitrary value. The semantics of the grid and the field names are left to the application code and the user. All other identifiers do have specific meanings in the fiber bundle data model and are used to determine the specific properties of a data set. In order to get from one hierarchy level to the next one, the "`[]`" and "`()`" operators are used for modified indexing (the return value is guaranteed to exist) and unmodified indexing (the return value will be zero if no subhierarchy entry exists for the given index):

```
Bundle MyBundle;
Slice&S = MyBundle[ 1.0 ];
RefPtr<Slice> MySlice = MyBundle( 1.0 );
```

In this example, the operator "`[]`" will create an entry for the time $t = 1.0$ in the `MyBundle` object, if it does not exist yet. It provides a reference to the Slice object that contains all data for $t = 1.0$. The operator "`()`" will query the `MyBundle` object whether data exist for $t = 1.0$, and if not, return an invalid pointer.

3.1 Topological Properties

The `Topology` level of the fiber bundle hierarchy describes a certain topological property. This can be the vertices, the cells, the edges etc. . It is loosely connected to the skeletons of a cw-complex, but in this context also used to specify different mesh refinement levels and agglomerations of certain elements. Details are given in [Benger, 2004], whereas here the only property of relevance is that all data fields that are stored within a `Topology` level have the same number of elements. I.e., they share their index space (a data space in HDF5 terminology). Moreover, each `Topology` object within a `Grid` object is uniquely identified via a set of integers, which are the *dimension* (e.g., the dimension of a k-cell), *index depth* (how many dereferences are required to access coordinate information in the underlying manifold) and *refinement level* (a multidimensional index, in general).

3.2 Relationship Maps

Numerical values within a `Topology` level are grouped into `Representation` objects, which hold all information that is *relative* to a certain "representer". Such a representer may be a coordinate object that refers to some cartesian (or polar) chart, or it may well be another `Topology` object, either within the same `Grid` object or even within another one. Given a `Topology` object called `Vertices` and a chart object `CartesianChart3D`, we may retrieve the representation of the Vertices in cartesian chart using the operator syntax:

```
Topology Vertices;
Chart    CartesianChart3D;

Representation& CartesianVertices = Vertices[ CartesianChart3D ];
```

Given a `Topology` object describing the triangles of a Grid, we may retrieve triangle information in cartesian coordinates in a similar way (e.g. surface normal vectors of the triangles), but as well retrieve the information on how the triangles relate to the vertices:

```
Topology Vertices, Triangles;
Representation & TrianglesAsVertices = Triangles[ Vertices ];
```

The inversion, the representation of the vertices via triangles, e.g. for determining which triangles share the same vertex, is easily accessed by the inverse operation:

```
Representation & VerticesAsTriangles = Vertices[ Triangles ];
```

Each `Representation` is a collection of `Field`s, which are basically multidimensional arrays that are accessed via some textual identifier. The value of this textual identifier is left to the application, with the mere exception of the "`Positions`" entry. This specific field describes the locality of the elements of one index space within the domain of the representer (e.g., within a chart, or as set of indices).

4 Example Application

The following example demonstrates how to set up a VISH object that defines a time-dependent uniform vector field. It involves infrastructure components from VISH , Memcore, and FiberLib2.

```
class   Vectorfield : public VObject // Implement VISH object
{
   Bundle B;
   RefPtr<VParameter> TimeParameter;  // refer to some parameter
public:

  Vectorfield()
  {
      // request floating point parameter
      TimeParameter = addParameter("time", 0.0);
  }

  void update()   // virtual VISH function
  {
  double time = TimeParameter->getValue() // request parameter value

       Slice &S = B[ time ];        // operating on FiberLib2 from here
       Grid  &G = S[ "unigrid" ];
       RefPtr<Skeleton> Vertices = G.makeVertices(3);
       RefPtr<Chart> myCartesian = G.makeChart( typeid(Fiber::CartesianChart3D) );
       Representation&R = (*Vertices)[ myCartesian ];
       RefPtr<Field> Coords  = R[ "Positions" ];
       MultiIndex<3> Dims = MIndex(31,43,53); // define size of the grid

       typedef MemArray<3, tvector3> VectorArray_t;
       RefPtr<Field> Vectors = R[ "vectors" ];
       RefPtr<VectorArray_t> VectorData = new VectorArray_t(Dims);
          Vectors->setData( VectorData, MemCache() );

      ProcArray_t*PCrds = new ProcArray_t();   // define uniform coordinates
         Coords->setData( PCrds, MemCache() ); // (details not shown)

      MultiArray<3,tvector3>     Vec = *VectorData;
  /* Vec is a multidimensional array of vectors and
     can be modified now, e.g. at index 4,4,4 with some
     time-dependent value (in practice, all elements need to be set) */
     Vec[ MultiIndex(4,4,4) ] = tvector3(1.0, 2.0, time);
  }
};
```

5 Results

VISH is still under an experimental development state, but the evolution of the core infrastructure has widely settled down already. Using this infrastructure, a simple visualization algorithm like rendering vector arrows of slice from a time-dependent uniform vector field can be realized in less then 300 lines of source code, including low-level OpenGL calls and parameter steering. A reference implementation employing QT of a user interface as a plugin to the VISH kernel

has been developed. Alternatively there exists a preliminary interface for the Amira visualization software, such that plugins can be shared in binary form among these two environments.

6 Conclusion

Practical experience shows that newly developed visualization techniques are not easily and quickly deployed by end-users. There exist complex – and frequently proprietary – applications that are problematic to adapt to a custom problem on one side, and there exist separate stand-alone versions of highly specialized algorithms that cannot be used in general context on the other side. The concepts of VISH intend to close this gap.

As part of the ongoing evolution, the VISH kernel has proved to suit well the needs of an abstraction layer, since the requirements for further kernel modifications have decreased as more functionality has been added to the kernel's periphery. Complemented with the fiber bundle component VISH is able to cover a wide range of scientific data types as well, even in its early phase. Future efforts therefore will now focus on plugin components rather than on the kernel itself.

It will be part of future investigation also to test whether the VISH abstraction is sufficient to encapsulate even contradictory concepts such as the push and pull model within the same application. The envisioned "worst-case" scenario here is to have the entire VISH kernel appear as a single object within another application without exposing its internal structure (i.e., the collection of VISH objects and their relation to parameters).

References

[Benger, 2004] Benger, W. (2004). *Visualization of General Relativistic Tensor Fields via a Fiber Bundle Data Model*. PhD thesis, FU Berlin.

[Butler & Pendley, 1989] Butler, D. M. & Pendley, M. H. (1989). A visualization model based on the mathematics of fiber bundles. *Computers in Physics*, 3(5), 45–51.

[CEI, 2007] CEI, C. E. I. (last visited 4/30/2007). EnSight - FEA and CFD post-processing visualization. URL: `http://www.ensight.com/ensight.html`.

[Colvin, 1994] Colvin, G. (1994). Exception safe smart pointers. URL: `http://std.dkuug.dk/jtc1/sc22/wg21/docs/papers/1994/N0555.pdf`.

[Haber et al., 1991] Haber, R. B., Lucas, B., & Collins, N. (1991). A data model for scientific visualization with provisions for regular and irregular grids. In *VIS '91: Proceedings of the 2nd conference on Visualization '91* (pp. 298–305). Los Alamitos, CA, USA: IEEE Computer Society Press.

[Johnson et al., 2006] Johnson, C. R., Moorehead, R., Munzner, T., Pfister, H., Rheingans, P., & Yoo, T. S., Eds. (2006). *NIH-NSF Visualization Research Challenges Report*. Los Alamitos, CA, USA: IEEE Press, 1st edition. URL: `http://tab.computer.org/vgtc/vrc/NIH-NSF-VRC-Report-Final.pdf`.

[Kitware, 2005] Kitware (2005). Visualization toolkit. Last visited 4/25/2007. URL: `http://www.kitware.org/`.

[SCI, 2007] SCI (last visited 4/30/2007). SCIRun: A Scientific Computing Problem Solving Environment, Scientific Computing and Imaging Institute (SCI). URL: `http://software.sci.utah.edu/scirun.html`.

[Stalling et al., 2005] Stalling, D., Westerhoff, M., & Hege, H.-C. (2005). Amira - an object oriented system for visual data analysis. In C. R. Johnson & C. D. Hansen (Eds.), *Visualization Handbook*: Academic Press. URL: `http://www.amiravis.com/`.

[Treinish, 1997] Treinish, L. A. (1997). Data explorer data model. `http://www.research.ibm.com/people/l/lloydt/dm/dx/dx_dm.htm`.