Field Model: An Object-Oriented Data Model for Fields

Technical Report NAS-01-005

Patrick J. Moran*
NASA Ames Research Center
pmoran@nas.nasa.gov

Abstract

We present an extensible, object-oriented data model designed for field data entitled Field Model (FM). FM objects can represent a wide variety of fields, including fields of arbitrary dimension and node type. FM can also handle time series data. FM achieves generality through carefully selected topological primitives and through an implementation that leverages the potential of templated C++. FM supports fields where the nodes values are paired with any cell type. Thus FM can represent data where the field nodes are paired with the vertices (“vertex-centered” data), fields where the nodes are paired with the D dimensional cells in \( \mathbb{R}^D \) (often called “cell centered” data), as well as fields where nodes are paired with edges or other cell types. FM is designed to effectively handle very large datasets; in particular FM employs a demand-driven evaluation strategy that works especially well with large field data. Finally, the interfaces developed for FM have the potential to effectively abstract field data based on adaptive meshes. We present initial results with a triangular adaptive grid in \( \mathbb{R}^2 \) and discuss how the same design abstractions would work equally well with other adaptive-grid variations, including meshes in \( \mathbb{R}^D \).

CR Categories: E. Data (large); I.1.3 Languages and Systems, Evaluation strategies; I.3.8 Computer Graphics Applications

Keywords: data models, object-oriented, C++, templates, scientific visualization, demand-driven evaluation.

1 Introduction

Underlying virtually every object-oriented visualization system is a data model. The data model forms a key part of the system design, effectively spelling out the types of data that can be analyzed by the system. A well-designed data model component can significantly enhance the capabilities of the overall system. For example, the developers of OpenDX (formerly IBM Data Explorer) often cite the consistent, unified nature of the DX data model as one of the key reasons for the success of their system [13, 1]. For large data visualization, the data model can have a significant impact on system efficacy. Poorly chosen abstractions can lead to performance problems or make development awkward. Well-designed abstractions can enhance code reuse and enable the coupling of components in new and interesting ways.

A recent trend in numerical computing is the growing popularity of adaptive meshes. Adaptive meshes increase or decrease resolution automatically as required by a simulation code. Adaptive meshes free the scientist from having to construct a mesh initially that completely anticipates where high resolution will be required. Adaptive meshes are also a natural choice when the resolution required in various regions of the domain changes over the course of the simulation, for instance, following a shock wave. Adaptive mesh techniques are often implemented as parallel algorithms, requiring careful load balancing and communication strategies in order to be most effective. Unfortunately, adaptive meshes tend not to match well with the data models underlying current general visualization systems, prompting mesh library developers to resort to developing visualization modules custom to their mesh design.

For those in the visualization community, adaptive meshes offer the possibility of new and interesting research topics. For example, one might want to couple various multi-resolution visualization techniques with the adaptive mesh data structures. For visualization system developers, adaptive meshes are a challenge. There are a number of current adaptive mesh development efforts, each with its own custom algorithms and data structures. One would like to apply the wealth of visualization techniques that have already been developed, yet one is likely not to have the resources to devote to interfacing to each adaptive mesh variation. This is where a carefully designed data model comes in. With appropriately chosen abstractions, a data model can insulate the visualization techniques from the majority of the idiosyncrasies of the mesh and field data structures. A carefully designed model can also enhance modularity: newly added mesh and field types in the future should not require significant modifications to existing code.

In general, the advantages of a good data model are not limited to adaptive meshes alone. Overall, our goal is to provide a common model for field data that will enhance the sharing of data sets and of visualization technique implementations. In the next section we provide an overview of some of the key concepts in the FM design that are intended to take us towards our goal. Following that we survey related data model work within the visualization community. Next, we discuss key features of the FM design, and then present current results. Finally, we conclude with a discussion of future plans for the FM project.

2 Field Model Concepts

Field Model objects are embedded in \( \mathbb{R}^D \), also known as physical space. Objects in \( \mathbb{R}^D \) are also said to have a physical dimensionality of \( D \). The regions in \( \mathbb{R}^D \) where fields are defined are discretized by meshes, which in turn are composed of cells. A k-cell is a subset of \( \mathbb{R}^D \) that is homeomorphic (topologically equivalent) to \( k \)-ball. Cells in FM are currently all linear objects. A 0-cell is a vertex, a 1-cell is an edge, 2-cells include triangles, quadrilaterals, and other polygons. A hexahedron, tetrahedra, pyramids and prisms are all examples of 3 cells. Every cell \( \sigma \) has a set of vertices. We use a more general concept of face than some are familiar with: a face of \( \sigma \) is specified by a non-empty subset of the vertices of \( \sigma \). For example, a hexahedron has not only quadrilateral faces, but also vertex and edge faces. Every cell is also a face of itself. The general face definition enables us to develop a more uniform treatment of objects.

*Mail Stop T27A-2, Moffett Field, CA 94035

1If a cell \( \sigma \) is not a simplex, then not every subset of the vertices of \( \sigma \) constitutes a face. In practice it is clear which subsets define valid faces.
with general dimension. A mesh $\mathcal{M}$ is a finite set of cells such that if $\sigma \in \mathcal{M}$, and $\tau$ is a face of $\sigma$, then $\tau \in \mathcal{M}$. Typically, cells in a mesh share common faces, so for example two tetrahedra can share triangle, edge, and vertex faces. If the cells with the highest dimensionality in mesh $\mathcal{M}$ are $B$ cells, then $\mathcal{M}$ is a $B$ mesh, and $\mathcal{M}$ has a base dimensionality of $B$. The base dimensionality of a mesh must be less than or equal to its physical dimensionality. The shape of a mesh $\mathcal{M}$ is the space occupied by the union of all the cells of $\mathcal{M}$. In most cases, the shape of a $B$ mesh is equivalent to a $B$ manifold with boundary. In order to rule out some cell collections that do not have a manifold shape, we require that every cell in a $B$ mesh $\mathcal{M}$ must be the face of some $B$ cell in $\mathcal{M}$. This requirement, for instance, rules out cases where we have a surface ($B = 2$) with spurious edges and vertices that are not part of the surface.

Figure 1 illustrates example meshes that can be constructed in $\text{FM}$. Note that $\text{FM}$ meshes can represent familiar objects such as regular meshes, curvilinear meshes, and tetrahedral unstructured meshes. Note too that our definition is general enough that we can represent objects less commonly thought of as meshes, such as a collection of vertices and edges signifying the atoms and bonds of a molecule ($B = 1, D = 3$). Also, note that the molecular example is a case where the set of cells adheres to our mesh definition, but the shape of the mesh is non-manifold.

A field defines a function within a region of space. In $\text{FM}$, each field object has a set of values called nodes (which can be accessed on demand), a mesh, and a pairing between the $k$ cells in the mesh and the nodes. The value of $k$ for a particular field is known as its node association index. The base and physical dimensionalities of a field are the dimensionalities of its underlying mesh. For fields with base dimensionality $B$, the most common node association indices seen in visualization data are 0 ("vertex centered") and $B$ (typically called "cell centered"). Other node association indices tend to be underrepresented in visualization studies, though they are still important scientifically. Node association index 1 fields often occur in electromagnetics simulations as well as some adaptive mesh systems, where adaptation criteria are paired with the edges. Node association index 2 fields are useful in some flow studies, where fluxes are tracked at the 2-cells in order to verify the correctness of the simulation.

For a field with node association index $k$, the user can request a single value at a particular $k$-cell or request multiple values at a $j$-cell, $j \neq k$. We define later how the field selects node values in the case where $j \neq k$. The user can also request a field value at an arbitrary point in physical space, or for fields based on meshes with structured behavior, at an arbitrary point in base space. In response to such queries, fields return an integer code indicating whether the query was successful (e.g., depending upon whether the given point was within the part of the domain where the field is defined), and a field value. Appropriate interpolation techniques are fairly well agreed upon for fields with node association index 0; for other node association indices appropriate interpolation methods are still under investigation.

Before proceeding with a description of the $\text{FM}$ design and implementation, we review previous data model work.

### 3 Related Work

The importance of a well-designed data model has been recognized early on in the visualization community, and there have been a number of efforts to develop a general design with a strong, formal foundation. One of the earliest was the fiber bundle model by Butler and Pendley [5]. Their model was inspired the mathematical abstraction of the same name. Fiber bundles have proven to be somewhat difficult to implement in their pure form, though the concepts have inspired several follow-on efforts. The original fiber bundle abstractions did not provide a convenient means to access
the underlying discretization (mesh) of a data set. This was a problem since many visualization algorithms operate by iterating over various types of cells of the mesh.

One system in particular that has been influenced by fiber bundle concepts is OpenDX (formerly IBM Data Explorer[13, 1]). Beginning with Haber et al [8], the fiber bundle model was adapted into a model that would support a general-purpose data-flow visualization system. OpenDX can handle fields with node association indices 0 or $B$, where $B$ is the base dimensionality of the field. OpenDX does not support adaptive meshes, though more recent work by Treinish [23] describes a model that would accommodate such data.

Another field modeling effort was the Field Encapsulation Library (FEL) project, first presented at Visualization '95 [3]. FEL excelled with the multi-block curvilinear grids that are popular in computational fluid dynamics applications. FEL differed from most other modeling efforts in that it defined separate class hierarchies for meshes and fields, rather than a single combined object type. A second version of FEL, FEL2, followed after a basic redesign and total rewrite [16, 15]. FEL2 introduced fundamental design features that enabled the library to operate with far larger data sets, including a consistent demand-driven evaluation model [14] and the integration of demand-paging techniques [6]. FEL2, like the original version of FEL, assumed that all objects were in $\mathbb{R}^3$ physical space, and that all fields effectively had a node association index of 0.

The Visualization Toolkit (vtk) [20], like OpenDX, is an open source visualization system with a fairly general data model. The vtk data model uses an extended concept of cells, including such primitives as polylines and triangle strips as cell types. Recent extensions [12] have focused on enabling the data model (and thus the whole system) to handle large data. Like FM, vtk utilizes a demand-driven evaluation strategy. In vtk, visualization techniques negotiate with a data source in order to determine appropriate streaming parameters, then the streaming commences. FM demand-driven evaluation is maximally fine-grained: visualization techniques request data one cell at a time, and the lazy evaluation happens at the same granularity. The FM approach leads to more function calls between the data consumer and producer, while the vtk approach implies that the data consumer has to know more about the characteristics of the data set it is accessing.

Another object-oriented data flow visualization system intended for large data visualization is SCIRun [2, 19]. One distinguishing characteristic of the SCIRun development effort was the focus on computational steering, i.e., analyzing data from a simulation and modifying simulation parameters, as the simulation is running. SCIRun also allowed for some mesh adaptation during a simulation run. The data model was not the primary focus of the overall development effort.

VisAD [10, 9] is a relatively general, object-oriented model for numerical data. The user can construct data objects with a style similar to expressing mathematical functions. In contrast to the models described previously, VisAD is implemented in Java. The VisAD model is quite flexible, though the Java implementation makes it less suitable for very large data. The VisAD model does put more effort into the inclusion of metadata — data about data — than most other designs. For example, VisAD provides for the specification of the units of measurement. Thus, for example, VisAD users should be less likely to confuse distances measured in miles with distances measured in kilometers.

# 4 Design and Implementation

Object-oriented design is hard. As Gamma et al. point out:

Experienced object-oriented designers will tell you that a reusable and flexible design is difficult if not impossible to get “right” the first time. Before a design is finished, they usually try to reuse it several times, modifying it each time [7].

In the case of the design of FM, we benefit from our experience with the original [3] and second generation [16, 15] Field Encapsulation Library (FEL) projects. Both generations had relatively demanding performance requirements from applications such as Virtual Wind tunnel [4]. Both also faced large data challenges. The second generation FEL was used by several different applications, providing reuse cases that helped us refine the class interfaces.

In FM, as in FEL, the two main types of objects in the model are meshes and fields. We discuss key features of both types next.

## 4.1 Shared Mesh and Field Interface

Both mesh and field classes inherit interface from the class 
\[
FMFieldInterface{B, D, T, \phi}
\]
where the template arguments $B, D$ and $T$ specify the base dimensionality, physical dimensionality and node type, respectively. For meshes, the node type is the coordinate type: $FMVector{D, FMCoord}$. The interface class specifies the member functions at.cell, at.base, and at.phys. The at.cell call takes a cell argument and appends values to a C++ standard library vector [11] passed in by pointer. The at.base and at.phys member functions provide access to field values at a single point in base space or physical space, respectively. We provide detail on the access function semantics below.

## 4.2 Mesh Interface

In general an application can access two types of information from a mesh object: geometric and topological. Geometric information is accessed primarily through the at.cell call, which produces the coordinates of the vertices of its cell argument. The at.base call takes a point in base coordinates and produces physical coordinates, thus it provides a means to convert between the two coordinate systems. (There is also a routine to do the opposite conversion). The at.phys call may at first seem redundant for meshes, but via its integer return value it does provide a means for verifying whether a given physical point is within the region where the field is defined.
**FM mesh objects have several member functions that provide topological information. Here we focus on one particular method, faces, that is key to the general node association design. To illustrate the faces method, we consider the small 2-mesh in Figure 2. Below the the mesh is an incidence graph which captures all the face relationships of the mesh. Each row of nodes in the graph corresponds to a particular cell dimensionality, with the rows ordered by increasing dimensionality from bottom to top. The graph contains an edge between nodes representing a k-cell form and a (k + 1) cell \( \tau \) if \( \sigma \) is a face of \( \tau \). The faces methods takes a k-cell \( \sigma \) and an integer argument \( j \). If \( j < k \), then faces returns the j cells that are faces of \( \sigma \). If \( j > k \), then faces returns the j cells that \( \sigma \) is the face of. If \( j = k \), then faces returns \( \sigma \). In terms of the graph in Figure 2, the \( j < k \) case is equivalent to following all paths downward to the \( j \)-th row from the node corresponding to \( \sigma \); the \( j > k \) case is equivalent to following all paths upward instead of downward. For those familiar with algebraic topology, the functionality of the faces call is essentially equivalent to the closure and star operators combined [17]. The faces method has many uses, for example it may be used for obtaining the edges of a given hexahedron. We will see how faces is used in conjunction with general node association below in Section 4.4.**

The FM mesh interface also supports iterator functionality compatible with the C++ standard library [11]. Meshes behave as collections of cells, and one can iterate over the cells. Unlike standard library collections, mesh objects provide a richer set of iteration possibilities. Typically one wants to iterate over cells of a particular dimension, or some other subset of the total collection of cells. FM provides this control via optional arguments to the begin iterator initializer call. Other than that difference, FM iterator style is compatible with the standard library, and one should be able use any of the standard library algorithms that operate with a collection that provides a _const_iterator_.

### 4.3 Mesh Implementation

Figure 3 summarizes the _FM_ mesh hierarchy. All mesh objects share common interface defined by _FM_mesh_ and _FM_mesh_<B,D,_>. The subclasses also share implementation through inheritance. For example, topological methods such as faces are implemented in _FM_structuredmesh_<B,D,_> and used by all structured mesh subclasses. Meshes are also responsible for point location and contribute geometric information that is used for interpolation. Efficient point location is critical in a high-performance field model, as it is an intermediate step when computing field values at arbitrary points in space. Through the class hierarchy we are able to provide point location routines that exploit characteristics of various types of meshes in order to provide increased performance.

### 4.4 Field Interface

Fields are all templated on base dimensionality, physical dimensionality and node type. _FM_ uses the same source for scalar, vector and in general tensor fields — all are instantiated from the same class definitions. The fundamental field member function for obtaining field values is _at_cell_, which produces one or more field values, returning them in a C++ standard library vector object [11]. For a field with node association index \( k \), an _at_cell_ call with a \( k \)-cell argument will produce a single field value. The same call with a \( j \) cell argument, \( j \neq k \), first would use the faces call on the underlying mesh to convert the \( j \)-cell into a collection of \( k \)-cells. Then, for each of the resulting \( k \) cells the field would append a single value to the result collection. Thus, for example, a node association index 0 field given a hexahedron argument would produce \(^3\text{Note that graph nodes and field nodes are different concepts.}\)

[Diagram: Figure 3: A synopsis of the main mesh classes, with the inheritance hierarchy signified by indentation. The _B_ and _D_ template arguments have the same meaning as for meshes. The _T_ template argument specifies the field node type, e.g., _float_. The purpose of the _FM_field_. parent class is analogous to that of _FM_mesh_; it provides a convenient handle when an application only requires the portion of the field interface that is not dependent upon the template arguments.]

8 values, 1 for each vertex. Or, for example, a node association index 3 ("finite volume") field _at_cell_ call with a vertex argument would return in general 8 values. We say "in general" since a vertex at the boundary of the mesh is the face of fewer than 8 hexahedra.

The utility of the _at_cell_ definition becomes further apparent when we consider cases where we have a field with one particular node association index, but want it to behave like another. Our approach would be to define an adapter class [7], derived _FM_field_<B,D,T>, with its own _at_cell_ method implementation. For instance, consider the case where we have a visualization algorithm that expects a single value when calling _at_cell_ with a vertex, but our field has a node association index not equal to 0. The adapter would take an incoming vertex argument and call _at_cell_ on the adapted field. The multiple values received in response could be averaged (perhaps with some weighting factors) to produce the final single value response. Such an adapter would enable us to reuse some older visualization techniques that make vertex-centered data assumptions.

### 4.5 Field Implementation

The _FM_field_. class hierarchy is summarized in Figure 4. The subclasses are primarily responsible for providing implementations for the _at_cell_ member function. Core fields produce values from a memory buffer, _FM_multi_field_<B,D,T> represents fields consisting of multiple subfields; _at_cell_ calls are delegated to the appropriate subfield. The derived field classes produce values on demand, applying a mapping function to the values produced by...
2D Triangular Adaptive Grid (TAG) code that has served as the design, visualization techniques are free to request values in space and time as an added dimension, utilizing the general dimension mechanism. Time series objects use tile time member to index into tile series member. Objects that do not vm3 with trine ignore tiffs member. sition, physical position, and cell arguments all contain a trine field values in the neighborhood of tile point using at_cell, and then interpolating based on the geometry of the cell. Since both at_phys and at_base are implemented in terms of at_cell, field subclasses are not required to provide implementations of these two functions. Nevertheless, some subclasses do provide their own implementations in order to employ optimizations that are specific to certain field types.

4.6 Time

In the previous sections we have said little about time, but this is not because time-varying data is unimportant. To the contrary, many large data sets come in the form of a time series. There are two main strategies we could choose in order to address the needs of time-varying data in FM. One approach would be to simply treat time as an added dimension, utilizing the general dimension mechanisms in we have already developed. The alternative would be to treat time as special, distinct from the spatial coordinates. At first glance, the former strategy may seem more appealing – we would like to reuse implementation when we can – but we decided to go the latter route instead, for several reasons. First, the spatial and temporal resolutions of the data can be dramatically different. Especially in post-processing applications, what is saved of the simulation is typically down sampled in time from the resolution used during the run. This implies we may want to do spatial and temporal interpolation differently. Second, many visualization techniques are designed to work at some instance ill trine, and riley do not hanral illtelpolation differently. Second, many visualization techniques during tile ran. This implies we may want to do spatial and tempo resolutions of the data can be dramatically different. Es-

\[ \text{Mesh} \quad \text{Card}_0 \quad \text{Card}_1 \quad \text{Card}_2 \]

<table>
<thead>
<tr>
<th>Mesh</th>
<th>Card_0</th>
<th>Card_1</th>
<th>Card_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>14605</td>
<td>43009</td>
<td>28404</td>
</tr>
<tr>
<td>Level 1</td>
<td>26189</td>
<td>88991</td>
<td>59000</td>
</tr>
<tr>
<td>Level 2</td>
<td>62926</td>
<td>235331</td>
<td>156498</td>
</tr>
<tr>
<td>Level 3</td>
<td>169933</td>
<td>662344</td>
<td>441147</td>
</tr>
<tr>
<td>Level 4</td>
<td>380877</td>
<td>1505024</td>
<td>1003313</td>
</tr>
<tr>
<td>Level 5</td>
<td>488574</td>
<td>1935619</td>
<td>1291834</td>
</tr>
</tbody>
</table>

Table 1: Cell set cardinalities for the 0-cells, 1-cells, and 2-cells in the airfoil example illustrated in Figure 5. The levels denote the number of refinement steps taken.

at_cell calls on the underlying fields.

The FM_field<B,D,T> class provides default implementations for the at_phys and at_base methods. Both implementations operate by locating a cell containing the given point, obtaining field values in the neighborhood of the point using at_cell, and then interpolating based on the geometry of the cell. Since both at_phys and at_base are implemented in terms of at_cell, field subclasses are not required to provide implementations of these two functions. Nevertheless, some subclasses do provide their own implementations in order to employ optimizations that are specific to certain field types.

5 Results

5.1 2-D TAG

As a demonstration of the Field Model capabilities, we consider a 2-D Triangular Adaptive Grid (TAG) code that has served as the basis for previous research efforts on adaptive grid techniques [18]. The TAG system is designed to be relatively insulated from a particular flow solver. TAG provides mesh geometry and connectivity information used by the solver; the solver in turn computes field node values and adaptation criteria that are associated with the mesh edges. Based on the adaptation criteria, the TAG system refines or coarsens the mesh. Figure 5 illustrates the airfoil test case that we consider here. Table 1 quantifies the mesh size in terms of the number of k-cells, k = 0..2, for each level of refinement.

Our motivation for choosing the TAG 2-D example is to test extensibility, in particular, with an adaptive mesh object. It is neither feasible nor desirable for FM to provide built in support for every mesh data structure; the library implementation would become too bulky and difficult to maintain. Instead, our goal is a design that is modular enough that new types of meshes can be added without significant modification to existing parts of the model. To be successful in this endeavor, we have three criteria. First, the class interfaces should be general enough to be applicable to a variety of object types. So far we consider ourselves to have met this requirement. FM can represent a variety of objects, including structured and unstructured objects and multi-block objects. We have not encountered significant limitations due to the interfaces. Second, the interface abstractions should not cause us to suffer an unacceptable loss in performance. We address this issue below. Finally, the class design should support reuse of parts of the implementation, so that newly introduced mesh and field types do not have to reimplement common routines. Our design has been successful in this respect as well. For 2-D TAG, we defined a new class TAG2D.unstructured.mesh, which is derived from FM.unstructured.mesh<2,2>. Note that the TAG2D class is not templated; the base dimensionality and physical dimensionality are hard-coded in the 2-D TAG implementation that we obtained. Our TAG2D class must provide implementation of some basic member functions such as at_cell and faces, since these functions refer to TAG specific data structures. Other functionality, such as iterator support, is inherited from FM.unstructured.mesh<2,2>; our TAG class can reuse the existing code.

The version of the 2-D TAG code we adapted for our example here executes serially. Oliker and Biswas [18] also have versions of the same code designed for parallel architectures, including message passing systems. We do not have experience yet with how well FM would accommodate such generalizations, but we are interested in investigating this. There is also a 3-D version of the adaptive grid code, developed by the same research group, that is analogous in many respects to 2-D TAG. The 3-D version contains non-simplicial cells, including pyramids, prisms and hexahedra, which should provide some additional challenge, although we do not anticipate any fundamental problems adapting such objects.

5.2 Performance

**Field Model** at its heart is about abstractions, and it is natural to ask what cost one has to pay for the benefits of abstraction. This in general is a difficult question to answer, because:

- cost is relative to some alternative, and what alternatives we have vary from case to case;
- how much abstraction overhead is apparent depends on the balance between data access and computation using the data;
- with large data, access time can be significantly influenced by the locality or lack thereof in data access patterns.

Despite these difficulties, it is still important to quantify the performance of the data model. We present the results from two initial
Figure 5: The airfoil data set with 2-D TAG. At the upper left is a close-up of the whole airfoil. At the upper right is a much closer view of the leading edge of the airfoil. The two images in the lower row display successive refinement iterations within the same region.
tests based on the 2-D TAG example discussed in the previous section. Our first test involves computing the bounding box of the TAG mesh. This test is in many respects a worst case scenario because we compare the abstract FM method to a hand-coded C-style implementation that has direct access to the data buffers, the amount of computation using the data is minimal, and the data are not really large enough for cache-miss rates to dominate. The columns under “Bounding Box” in Table 2 summarize the results for the example airfoil data set, measured on a 195 MHz, dual processor SGI Onyx2 workstation with 512M of memory. The worst case does look pretty bad: the difference in total times in each case is over an order of magnitude slower. Still, depending on the application, the abstract performance may be fast enough.

As a second example, we consider a scenario where we generate postscript images consisting of the edges in the TAG mesh. We time the actual code we used to generate the images in Figure 5. Like the first scenario, we compare access through FM to hand-coded direct access to the data structures. Unlike the first scenario, the computation involves some simple transformations followed by a write to our postscript file. This is clearly more expensive than our bounding box computation. The columns under “Edge Drawing” summarize the results. The FM version runs slower, but by roughly only 5%. For this application the overhead is likely to be acceptable.

The timings in Table 2 clearly are not a thorough assessment of FM performance. Field Model is still relatively early in its development process, and we have done little performance tuning so far. Our plan is to port the VisTech library [21] to FM in the near future. VisTech consists of a collection of standard visualization algorithms, written in terms of FEL2 [16, 15]. We will be able to compare FM/VisTech performance with that of FEL2/VisTech, and in some cases with implementations hand coded for specific mesh and field types. VisTech applications will provide examples with more typical balance between data access and computation as well as relatively typical data access patterns for visualization applications.

6 Conclusion

We have presented an overview of Field Model (FM), an object-oriented data model for mesh and field data. FM benefits significantly from our experiences with FEL2 [16, 15], an earlier effort focused on the development of high-performance library for large data. FM goes beyond FEL2 in generality: FM can represent data with general base and physical dimensionality as well as fields with general node association. Furthermore, we anticipate that FM will be able to successfully handle adaptive mesh types. Our experience so far with the 2-D TAG [18] adaptive code confirms our expectations.

Two of the primary design goals of the FM project are modularity and extensibility. Our vision is that FM will serve as a common model where others in the community can contribute extensions specific to their mesh and field objects. The incentive would be that data brought into the shared model could be analyzed by what we hope will be a wide collection of analysis techniques written in terms of the model. Towards this end, we are working to establish FM as an Open Source [22] project, with its development home on SourceForge. We have established a site there (http://field-model.sourceforge.net), and we currently have a few initial files uploaded to the repository. We anticipate that by Vis’01 all the source used to create objects such as those displayed in this article will be available from our site.

Acknowledgements

We would like to thank Ernst Mücke for the interlinked tori point set used in Figure 1. We would also like to thank Rupak Biswas for providing the 2-D TAG code and example data used in Section 5.1 We are also grateful to Pete Vanderbilt and all the members of the Data Analysis Group for helpful insights. Finally, we would like to thank VA Linux for their ongoing support of the Open Source [22] software movement, and SourceForge in particular.

References


<table>
<thead>
<tr>
<th>Mesh</th>
<th>Bounding Box</th>
<th>Edge Drawing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Abstract</td>
<td>Direct</td>
</tr>
<tr>
<td>Initial</td>
<td>13.9</td>
<td>1.1</td>
</tr>
<tr>
<td>Level 1</td>
<td>23.5</td>
<td>1.8</td>
</tr>
<tr>
<td>Level 2</td>
<td>56.3</td>
<td>4.3</td>
</tr>
<tr>
<td>Level 3</td>
<td>152.4</td>
<td>11.7</td>
</tr>
<tr>
<td>Level 4</td>
<td>343.8</td>
<td>26.4</td>
</tr>
<tr>
<td>Level 5</td>
<td>462.8</td>
<td>35.7</td>
</tr>
</tbody>
</table>

Table 2: Initial FM example timings, in msec. The “Bounding Box” columns illustrate the worst case scenario for FM, we compare an algorithm written using FM to a hand-coded implementation that accesses the data structures directly, and the amount of compute relative to each data access is small. In this scenario the FM version comes out over an order of magnitude slower. The “Edge Drawing” columns illustrate a scenario that may be more typical. Once again we compare an algorithm written using FM to a hand-coded implementation that accesses the data structures directly, but in our second scenario the compute time is more significant. In this second scenario, the FM version is slower, but by roughly only 5%.


---

**Appendix A**

We provide the Field Model (FM) source for the examples presented in the body of this report in the following pages.
```cpp
#include "FM_vector.h"
#include "FMsubmesh_id.h"
#include "FM_time.h"

template <int B, typename T = FM coord>
class FM base : public FM vector<B,T>
{
public:
    FM time<T> time;
    FM submesh id submesh id;

    FM_base() {
    FM_base(const FM time<T>& t, const FM submesh id& sid) :
        time(t), submesh id(sid) {
    FM_base(const FM vector<B,T>& v) :
        FM vector<B,T>(v) {
    FM_base(const FM vector<B,T>& v, const FM time<FM u32>& t, const FM submesh id& sid) :
        FM vector<B,T>(v), time(t), submesh id(sid) {
    }
    }
    }

    template <typename T>
class FM base<l,T> : public FM vector<l,T>
    {
public:
    FM time<T> time;
    FM submesh id submesh id;
    FM_base() {
    FM_base(const FM time<T>& t, const FM submesh id& sid) :
        time(l), submesh id(l) {
    FM_base(const FM vector<l,T>& v) :
        FM vector<l,T>(v) {
    FM_base(T c0, T c1, const FM time<T>& t, const FM submesh id& sid) :
        FM vector<l,T>(c0, c1, time(t), submesh id(lid)) {
    }
    }
    }

    template <typename T>
class FM base<2,T> : public FM vector<2,T>
    {
public:
    FM time<T> time;
    FM submesh id submesh id;
    FM_base() {
    FM_base(const FM time<2,T>& t, const FM submesh id& sid) :
        time(2), submesh id(2) {
    FM_base(const FM vector<2,T>& v) :
        FM vector<2,T>(v) {
    FM_base(T c0, T c1, const FM time<2,T>& t, const FM submesh id& sid) :
        FM vector<2,T>(c0, c1, time(t), submesh id(lid)) {
    }
    }

    template <typename T>
class FM base<3,T> : public FM vector<3,T>
    {
public:
    FM time<T> time;
    FM submesh id submesh id;
    FM_base() {
    FM_base(const FM time<3,T>& t, const FM submesh id& sid) :
        time(3), submesh id(3) {
    FM_base(const FM vector<3,T>& v) :
        FM vector<3,T>(v) {
    FM_base(T c0, T c1, T c2, const FM time<3,T>& t, const FM submesh id& sid) :
        FM vector<3,T>(c0, c1, c2, time(t), submesh id(lid)) {
    }
    }

    template <int B, typename T = FM coord>
    std::ostream& operator<<(std::ostream& lhs, const FM base<B,T> rhs)
    {
        lhs << "{";
        for (int i = 0; i < B; i++)
            if (i > 0) lhs << ", ";
        lhs << rhs.time();
        if (rhs.time().defined())
            if (i++ > 0) lhs << ", time ";
            else time;
        if (rhs.submesh_id().defined())
            if (i++ > 0) lhs << ", submesh_id ";
            else submesh_id;
        return lhs << "}";
    }

    // Emacs mode * c++ *
    #ifndef FM_BASE_H
    #define FM_BASE_H
    /*
    • NAME: FMbase.h
    • WRITTEN BY:
    Patrick Moran
    pmoran@nas.nasa.gov
    */
    #include "FM_vector.h"
    #include "FMsubmesh_id.h"
    #include "FM_time.h"

template <int B, typename T = FM coord>
class FM base : public FM vector<B,T>
{
public:
    FM time<T> time;
    FM submesh id submesh id;
    FM_base() {
    FM_base(const FM time<T>& t, const FM submesh id& sid) :
        time(t), submesh id(sid) {
    FM_base(const FM vector<B,T>& v) :
        FM vector<B,T>(v) {
    FM_base(const FM vector<B,T>& v, const FM time<FM u32>& t, const FM submesh id& sid) :
        FM vector<B,T>(v), time(t), submesh id(sid) {
    }
    }
    }

    template <typename T>
class FM base<l,T> : public FM vector<l,T>
    {
public:
    FM time<T> time;
    FM submesh id submesh id;
    FM_base() {
    FM_base(const FM time<T>& t, const FM submesh id& sid) :
        time(l), submesh id(l) {
    FM_base(const FM vector<l,T>& v) :
        FM vector<l,T>(v) {
    FM_base(T c0, T c1, const FM time<T>& t, const FM submesh id& sid) :
        FM vector<l,T>(c0, c1, time(t), submesh id(lid)) {
    }
    }
    }

    template <typename T>
class FM base<2,T> : public FM vector<2,T>
    {
public:
    FM time<T> time;
    FM submesh id submesh id;
    FM_base() {
    FM_base(const FM time<2,T>& t, const FM submesh id& sid) :
        time(2), submesh id(2) {
    FM_base(const FM vector<2,T>& v) :
        FM vector<2,T>(v) {
    FM_base(T c0, T c1, const FM time<2,T>& t, const FM submesh id& sid) :
        FM vector<2,T>(c0, c1, time(t), submesh id(lid)) {
    }
    }

    template <typename T>
class FM base<3,T> : public FM vector<3,T>
    {
public:
    FM time<T> time;
    FM submesh id submesh id;
    FM_base() {
    FM_base(const FM time<3,T>& t, const FM submesh id& sid) :
        time(3), submesh id(3) {
    FM_base(const FM vector<3,T>& v) :
        FM vector<3,T>(v) {
    FM_base(T c0, T c1, T c2, const FM time<3,T>& t, const FM submesh id& sid) :
        FM vector<3,T>(c0, c1, c2, time(t), submesh id(lid)) {
    }
    }

    template <int B, typename T = FM coord>
    std::ostream& operator<<(std::ostream& lhs, const FM base<B,T> rhs)
    {
        lhs << "{";
        for (int i = 0; i < B; i++)
            if (i > 0) lhs << ", ";
        lhs << rhs.time();
        if (rhs.time().defined())
            if (i++ > 0) lhs << ", time ";
            else time;
        if (rhs.submesh_id().defined())
            if (i++ > 0) lhs << ", submesh_id ";
            else submesh_id;
        return lhs << "}";
    }

    // Emacs mode * c++ *
    #ifndef FM_BASE_H
    #define FM_BASE_H
    /*
    • NAME: FMbase.h
    • WRITTEN BY:
    Patrick Moran
    pmoran@nas.nasa.gov
    */
    #include "FM_vector.h"
    #include "FMsubmesh_id.h"
    #include "FM_time.h"

    template <int B, typename T = FM coord>
class FM base : public FM vector<B,T>
{
public:
    FM time<T> time;
    FM submesh id submesh id;
    FM_base() {
    FM_base(const FM time<T>& t, const FM submesh id& sid) :
        time(t), submesh id(sid) {
    FM_base(const FM vector<B,T>& v) :
        FM vector<B,T>(v) {
    FM_base(const FM vector<B,T>& v, const FM time<FM u32>& t, const FM submesh id& sid) :
        FM vector<B,T>(v), time(t), submesh id(sid) {
    }
    }
    }

    template <typename T>
class FM base<l,T> : public FM vector<l,T>
    {
public:
    FM time<T> time;
    FM submesh id submesh id;
    FM_base() {
    FM_base(const FM time<T>& t, const FM submesh id& sid) :
        time(l), submesh id(l) {
    FM_base(const FM vector<l,T>& v) :
        FM vector<l,T>(v) {
    FM_base(T c0, T c1, const FM time<T>& t, const FM submesh id& sid) :
        FM vector<l,T>(c0, c1, time(t), submesh id(lid)) {
    }
    }
    }

    template <typename T>
class FM base<2,T> : public FM vector<2,T>
    {
public:
    FM time<T> time;
    FM submesh id submesh id;
    FM_base() {
    FM_base(const FM time<2,T>& t, const FM submesh id& sid) :
        time(2), submesh id(2) {
    FM_base(const FM vector<2,T>& v) :
        FM vector<2,T>(v) {
    FM_base(T c0, T c1, const FM time<2,T>& t, const FM submesh id& sid) :
        FM vector<2,T>(c0, c1, time(t), submesh id(lid)) {
    }
    }

    template <typename T>
class FM base<3,T> : public FM vector<3,T>
    {
public:
    FM time<T> time;
    FM submesh id submesh id;
    FM_base() {
    FM_base(const FM time<3,T>& t, const FM submesh id& sid) :
        time(3), submesh id(3) {
    FM_base(const FM vector<3,T>& v) :
        FM vector<3,T>(v) {
    FM_base(T c0, T c1, T c2, const FM time<3,T>& t, const FM submesh id& sid) :
        FM vector<3,T>(c0, c1, c2, time(t), submesh id(lid)) {
    }
    }

    template <int B, typename T = FM coord>
    std::ostream& operator<<(std::ostream& lhs, const FM base<B,T> rhs)
    {
        lhs << "{";
        for (int i = 0; i < B; i++)
            if (i > 0) lhs << ", ";
        lhs << rhs.time();
        if (rhs.time().defined())
            if (i++ > 0) lhs << ", time ";
            else time;
        if (rhs.submesh_id().defined())
            if (i++ > 0) lhs << ", submesh_id ";
            else submesh_id;
        return lhs << "}";
    }
``
template <int B> class FM structured cell : public FM cell

friend class FM structured mesh 0 cell iter impl<B>;
friend class FM structured mesh B cell iter impl<B>;

public:
virtual FM u32 get submesh id() const { return submesh id; }
friend class FM structured mesh B cell iter impl<B>;
friend class FM structured mesh 0 cell iter impl<B>;

public:
virtual std::ostream& str(std::ostream& o) const = 0;
virtual bool is subsimplex() const { return false; }
virtual bool is equal to(const FM cell&) const = 0;
virtual FM cell type enum get type() const = 0;
virtual FM u32 get n faces(FM u32) const = 0;
virtual FM u32 get dimension() const = 0;
virtual void set submesh id(const FM submesh id& sid) { submesh id = sid; }
virtual FM submesh id get submesh id() const { return submesh id; }
virtual void set time(const FM time<FM u32>& t) { time = t; }
virtual FM time<FM u32> get time() const { return time; }
virtual "FM cell() {
FM cell(const FM t_ll& c) : time(c.time), submesh id(c.submesh id) {
FM cell(const FM time<FM u32>& t, const FM submesh id& s) :
FM time(t), submesh id(s) {
FM cell(const FM time<FM u32>& t) : time(t) {
FM cell() {

protected:
FM time<FM u32> t;
FM submesh id submesh id;
};

public:
FM cell() : time(0), submesh id(0) {
FM time<FM u32> t;
FM submesh id submesh id;
};

return !(lhs rhs);

return lhs.is_equal_to(rhs);

// switch get dimension() {
// case 0: o << "v";
// case 1: o << "E";
// case 2: o << "V";
// case 3: o << "F";
// case 4: o << "T";
// case 1: o << "Q";
// case 2: o << "P";
// case 3: o << "M";
// case 4: o << "H";
// case 5: o << "I";
// case 6: o << "G";
// case 7: o << "F";
// case 8: o << "E";
// case 9: o << "D";
// case 10: o << "C";
// case 11: o << "B";
// case 12: o << "A";
// case 13: o << "X";
// case 14: o << "Y";
// case 15: o << "Z";
// default: o << get dimension(); // d-cube
// o << ";";
PM u32 d = get dimension();
return PM phantom(id, d - h); // h-pow 2(d - h);

switch get dimension() {
// case 0: o << "v";
// case 1: o << "E";
// case 2: o << "V";
// case 3: o << "F";
// case 4: o << "T";
// case 1: o << "Q";
// case 2: o << "P";
// case 3: o << "M";
// case 4: o << "H";
// case 5: o << "I";
// case 6: o << "G";
// case 7: o << "F";
// case 8: o << "E";
// case 9: o << "D";
// case 10: o << "C";
// case 11: o << "B";
// case 12: o << "A";
// case 13: o << "X";
// case 14: o << "Y";
// case 15: o << "Z";
// default: o << get dimension(); // d-cube
// o << ";";
PM u32 d = get dimension();

for (int j = 0; j < B; j++) {
if (j > 0) o << ";",
if (j < 0) o << "*",
for (int j = 0; j < B; j++) {
if (j > 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j > 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "*",
if (j < 0) o << "}
};
* including without limitation the rights to use, *
* copy, modify, merge, publish, distribute, sublicense, *
* and/or sell copies of the Software, and to permit *
* persons to whom the Software is furnished to do so, *
* subject to the following conditions: *
* *
* The above copyright notice and this permission *
* notice shall be included in all copies or substantial *
* portions of the Software. *
* *
* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY *
* OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT *
* LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS *
* FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO *
* EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE *
* FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER *
* IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING *
* FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR *
* THE USE OR OTHER DEALINGS IN THE SOFTWARE. *
*
*/

#ifndef FMCOMBINATORICS_H
#define FMCOMBINATORICS_H
/*
• NAME: FM combinatorics.h
• WRITTEN BY:
Patrick Moran
pmoran@nas.nasa.gov
*/
#include <vector>
#include "FMvector.h"

template <typename T>
void FM_swap(T* lhs, T* rhs) {
    T tmp;
    T tmp = *lhs;
    *lhs = *rhs;
    *rhs = tmp;
}

template <typename T>
T FM_fact(T n) {
    T res = 1;
    for (T i = 2; i < n; i++)
        res *= i;
    return res;
}

template <typename T>
T FM_choose(T b, T k) {
    assert(k <= b);
    return b < LUT_SIZE ? lut[b][k] : FM_fact(b) / (FM_fact(b - k) * FM_fact(k));
}

template <typename T, int B>
void FM_choose_choices(FM u32 k, std::vector<FMvector<B,bool>*> choices) {
    int i, ik = int(k);
    assert(i <= B);
    FM_u32 n = choices[FM_choose(FM_u32(B), k)];
    choices[FM_choose(FM_u32(B), k + 1)];
    int indices[B + 1];
    for (i = 0; i < ik; i++)
        indices[i] = i;
    indices[i] = B;
    FMvector<B,bool> all_false, choice;
    for (i = 0; i < B; i++)
        all_false[i] = false;
    for (FM u32 c = 0; c < n choices; c++) {
        choice = all_false;
        for (i = 0; i < ik; i++)
            choice[indices[i]] = true;
        for (i : ik i; i >= 0; i--)
            if (indices[i] < i)
                indices[i + i] = indices[i] + i;
            else
                break;
    }
}

template <typename T>
T FM_pow_2(T i) {
    return i << i;
}

template <typename T>
int FM_sign(T i) {
    return i < 0 ? -1 : 0;
}

template <typename T>
T FM_abs(T t) {
    return t < 0 ? -t : t;
}

template <typename T>
T FM_min(T lhs, T rhs) {
    return lhs < rhs ? lhs : rhs;
}

template <int N, typename T>
bool FM_odd_even(FMvector<N,T> v) {
    T sum = v[0];
    for (int i = 1; i < N; i++)
        sum += v[i];
    return sum & 1 ? true : false;
}
template <typename T>
class FM constant_field : public FM_field<B, M, T>
{
public:
    const T constant;

    FM constant_field(const FM ptr<FM mesh<B, M, T>>& m, const T& c, int na, FM properties_cache* pc = 0) :
        FM_field<B, M, T>(m, na, pc)_, constant(c) {
    }

    virtual std::ostream& str(std::ostream& o) const
    {
        return o << "FM constant field<" << typeid(T).name() << ">;";
    }

    virtual int at_cell(const FM cell* c, T* vals) const
    {
        std::vector<FM ptr<FM cell> > faces = mesh->faces(c, node association index);
        for (size_t i = 0; i < faces.size(); i++)
        { vals[i] = constant; }
        return FM_OK;
    }

};

/* Copyright (c) 2000 Advanced Management Technology, Incorporated */
/* Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: */
/* The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software. */
/* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. */
EMACS mode * c++ *

#ifndef FM_CONTEXT_H
#define FM_CONTEXT_H

/**
 * NAME: FM context.h
 * WRITTEN BY: Patrick Moran
 */

#include "FMcell.h"

pmoran@nas.nasa.gov

class FMcontext {
public:
FM context() :
  simplicial_decomposition(0),
  locate_verbosity(0),
  locate_effort(4) {
    FM u32 simplicial_decomposition;
    FMptr<FMcell> last_cell;
    FM u32 locate_verbosity;
    FM u32 locate_effort;
  }

};

Copyright (c) 2000
Advanced Management Technology, Incorporated

Permission is hereby granted, free of charge,
to any person obtaining a copy of this software
and associated documentation files (the "Software"),
to deal in the Software without restriction,
including without limitation the rights to use,
modify, merge, publish, distribute, sublicense,
and/or sell copies of the Software,
and to permit persons to whom the Software is furnished to do so,
subject to the following conditions:
The above copyright notice and this permission
notice shall be included in all copies or substantial
portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
THE USE OR OTHER DEALINGS IN THE SOFTWARE.

#endif

// Emacs mode * c++ *

#ifndef FM_CORE_FIELD_H
#define FM_CORE_FIELD_H

/**
 * NAME: FM core field.h
 * WRITTEN BY: Patrick Moran
 */

#include "FM field.h"
template <int B, int D, typename T>
class FM core field : public FM field<B,m,T> {
public:
FM core field(const FM ptr<FM mesh<B,m> >& mesh,
FM u32 na, FM properties cache* pc) :
  FM field<B,m,T>(mesh, na, pc) {
    virtual std::ostream& str(std::ostream& o) const
    return o << "FM core field";
  }

};

template <int B, int D, typename T>
class FMcore fieldT layout;

template <int B, int D, typename T>
std::pair<T,T>
FM get min max aux(const FMptr<FMcore fieldT layout<B, D,T> >& field,
const FMtime<FMu32>* t, const FM submesh id* sid,
const FMtruetype&);
© 2000, Advanced Management Technology, Incorporated

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

- The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

- The SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

The Software is provided "as is", without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claims, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the Software or any use or other dealings in the Software.

// Do not modify this file. //
#define FM_CURVILINEAR_MESH_H

/*
 * NAME: FM_curvilinear_mesh.h
 * WRITTEN BY: Patrick Moran
 * pmoran@nas.nasa.gov
 */

#include <algorithm>
#include <queue>
#include "FMorient.h"
#include "FMstructured_mesh.h"
#include "FMfunctional.h"
#include "FMstructured_mesh.h"

class FM curvilinear mesh : public FM structured mesh<B,D>
{
protected:
    FM classic core field T layout(const FM ptr<FM structured mesh<B,D>>& m, const T* d, bool ds, FM properties_cache* pc = 0) :
        FM structured_mesh_field_2_layout(m, d, ds, pc) {} 

public: {
    virtual int at cell(const FM cell* c, T* vals) const {
        FM u32 n_indices, _ndices(1 << B);
        FM vector<B,FU32> stride;
        FM coord stride size = FM coord(1);
        FM coord current distance = FM distance2(p, cv);
        for (d = 0; d < B; d++) {
            FM vector<D,FU32> cv;
            FM structured 0 cell<B> initial, final;
            FM structured 0 cell<B>* cm = cm->at cell(c, &cv);
            FM structured 0 cell<B>* final = cm->at cell(c, &cv);
            FM structured 0 cell<B> current = initial;
            FM structured 0 cell<B> post step;
            FM coord current distance, best post step distance = current distance;
            FM structured 0 cell<B> best post step;
            FM vector<B,FU32> best post step indices;
            FM coord best post step distance = current distance;
            for (d = 0; d < B; d++) {
                FM structed mesh vertex indices(mesh, &n indices, indices);
                FM u32 i = 0, l = n_indices, indices; 
                vals[l] = data[indices[i]];
                for (FM u32 i = 0; i < n indices; i++) {
                    T* dst = &(*vals)[previous size];
                    if (stride[d] == 0)
                        continue;
                    vals[l] = data[indices[i]];
                    FM coord current distance = best post step distance;
                    if (res != FM OK)
                        return res;
                    return FM_OK;
                } 
            } 
        } 
    } 
} 

/*
 * NAME: FM_curvilinear_mesh.cc
 * WRITTEN BY: Patrick Moran
 * pmoran@nas.nasa.gov
 */

#include "FMorient.h"
#include "FMstructured_mesh.h"
#include "FMfunctional.h"

class FM curvilinear mesh : public FM structured mesh<B,D>
{
protected: 
    Note: FM curvilinear mesh initialize() needs to be called by
    the derived classes that define at cell so this routine

};
// has the option of accessing vertex coordinates as part of the public:
void curvilinear_mesh_initialize()
{
    FM vector<3, FM u32> initial_location = dimensions;
    initial_location /= 2;
    initial_locations.push_back(initial_location);
}

public:
    FM curvilinear_mesh(const FM vector<3, FM u32>& dimensions,
        FM properties_cache* pc = 0;
        FM structured_mesh<3, 3, 3> initial,
        FM structured_cache<3> initial); const
    FM curvilinear_mesh(const FM vector<3, FM u32>& dimensions,
        FM properties_cache* pc = 0;
        FM structured_mesh<3, 3, 3> initial,
        FM structured_cache<3> initial); const
}

private:
    int hexahedral_walk_locate(const FM phys<3>& p,
        FM context* ctxt, FM ptr<FM cell>* c) const
    {
        FM orient(cv[FM hexahedron face[even odd][face][0]],
            cv[FM hexahedron face[even odd][face][2]],
            cv[FM hexahedron face[even odd][face][3]],
            cv[FM hexahedron face[even odd][face][1]]);
        if (res != FM OK) return res;

        *c = new FM structured<3, 3, 3, 3>(initial, initial, initial);
        return res << std::endl;
    }

    void curvilinear_meshInitialize();
    { /* code */ }
if (ctxt->locate verbosity > 0)
ctxt->last cell = *c;
else
]
]

next subtetrahedron face:

int res;
bool suppressed step off mesh = false;
FM u32 total faces tested threshold =
FM u32 total faces tested = 0;
FM u32 faces tested = 0;
bool new cell = true;
else
if (initial->is subsimplex())
FM u32 subid;
FM vector<3,FM u32> indices = initial->get indices();
assert(initial
res = FM POINT LOCATE WALKED OFF MESH;
res = FM OK;
}
private:
  std::vector<FM_vector<3, FM_u32>> > initial_locations;

// FM_curvilinear_mesh_T_layout<B,D> is a curvilinear mesh where the
// coordinates are contained in a single array of FM_vector<D, FM_coord>.
// i.e., an array where the coordinates for each vertex are contiguous.
// template <int B, int D>
  class FM_curvilinear_mesh_T_layout : public FM_curvilinear_mesh<B,D> {

public:
  const FM_vector<B, FM_coord>* const coordinates;
  const bool delete suppression;

  FM_curvilinear_mesh_T_layout(const FM_vector<B, FM_coord>& dimensions, const FM_vector<D, FM_coord>* const coordinates);
  FM_curvilinear_mesh_T_layout(const FM_vector<B, FM_coord>& dimensions, const FM_vector<D, FM_coord>* const coordinates);
  FM_curvilinear_mesh_T_layout(const FM_vector<B, FM_coord>& dimensions);
  FM_curvilinear_mesh_T_layout(const FM_vector<B, FM_coord>& dimensions, const FM_vector<D, FM_coord>* const coordinates);

private:
  std::vector<FM_vector<3, FM_u32>> initial_locations;

  if (!bounding box valid) {
    return bounding box;
  }

  virtual std::pair<FM_vector<3, FM_coord>, FM_vector<3, FM_coord>> get_bound_box(const FM_time<FM_u32>*, const FM_submesh_id*) const { return FM_OK; }

  virtual int cell(const FM_cell* c, FM_vector<m, FM_coord>* vals) const {
    auto* pc = &(*vals)[previous size];
    FM operator minimize_EQUALS(bb, *cp++);
    for (i = 0; i < n_indices; i++) {
      FM u64 indices[l << B];
      FM u32 n_indices;
      FM curvilinear mesh initialize();
    }
  }

  virtual FM operator minimize_EQUALS(bb, *cp);

  FM curvilinear mesh initialize();

  bounding_box_valid = true;
  return bounding_box;

  // FM_curvilinear_mesh_T_layout<B,D>: get_bound_box works with
  // a pointer directly into the coordinates buffer, testing every vertex.
// template <int B, int D>
  std::pair<FM_vector<B, FM_coord>, FM_vector<B, FM_coord>> > get_bound_box(const FM_time<FM_u32>* = 0,
  const FM_submesh_id* = 0) const {
    return bounding_box;
  }

  // FM_curvilinear_mesh_T_layout<B,D>: get_bound_box works with
  // a pointer directly into the coordinates buffer, testing vertices
  // on the boundary of the mesh.
  FM operator minimize_EQUALS(bb, *cp++);
  // edge of k = 1 to dimensions[2] - 1 slices
  for (i = 0; i < dimensions[2] - 1; i++) {
    FM operator minimize_EQUALS(bb, *cp);
  } // edge of k = 1 to dimensions[2] - 2 slices
  for (i = 0; i < dimensions[2] - 2; i++) {
    FM operator minimize_EQUALS(bb, *cp++);
  }
  // // edge of k = 1 to dimensions[2] - 3 slices
  // // for (i = 0; i < dimensions[2] - 3; i++) {
  // //   FM operator minimize_EQUALS(bb, *cp);
  // //   cp += dimensions[2] - 1;
  // // } // edge of k = 1 to dimensions[2] - 4 slices
  // // for (i = 0; i < dimensions[2] - 4; i++) {
  // //   FM operator minimize_EQUALS(bb, *cp++);
  // // }

  // bounding box = bb;
  // bounding_box_valid = true;
  // return bounding_box;

*/

/* Copyright (c) 2000 * Advanced Management Technology, Incorporated *
 * Permission is hereby granted, free of charge, to any person obtaining a copy of this software * and associated documentation files (the "Software"), * to deal in the Software without restriction, including without limitation the rights to use, * copy, modify, merge, publish, distribute, sublicense, * and/or sell copies of the Software, and to permit * persons to whom the Software is furnished to do so, * subject to the following conditions: *
 * * The above copyright notice and this permission notice * shall be included in all copies or substantial * portions of the Software. *
 * * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY *
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS *
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO *
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE *
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER *
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING *
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR *
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE. *
 */

/*=
#define */
template <int B, int D> class FM_binary derived field : public FM field<B,D,T> 

public:
    // Constructor
    FM_binary derived field(const FM ptr<FM field<B,D,R> >& lhs, const F function; const FM field interface<B,D,S>* rhs field; const FM field interface<B,D,R>* lhs field; const FM ptr<FM shared object> lhs so; const FM ptr<FM shared object> rhs so; const FM field<B,D,T>(lhs >mesh, lhs >node association index, pc), const F& fun, FM properties cache* pc) : 

    // Virtual methods
    virtual int at cell(const FM cell* c, std::vector<T>* vals) const 
    
    // Other methods

private:
    void init() const 

    // friendship


}
std::vector<T> vals[0];
for ( ; i != e; ++i) {
  ++i;
  min max.first vals[0];
  min max.first vals[0];
} else if (vals[0] > min max.second)
  if (vals[0] < min max.first)
    if (res != FM_OK) continue;
  i = *field >at cell (*i, &vals);
  vals. clear();
} throw std::logic_error(err.str());

if (i e) {
  for ( ; i != e; ++i) {
    std::vector<T> vals (i);
    min max.first vals[0];
    min max.first vals[0];
  }
}
template<typename T> class FM_negate_fun : public std::unary_function<T,T>

    public:
    int operator()(const T& t, T* res) const
    {
        *res = -t;
        return FM_OK;
    }

    template<typename T>
    class FM_plus_fun : public std::binary_function<T,T,T>

    public:
    int operator()(const T& lhs, const T& rhs, T* res) const
    {
        *res = lhs + rhs;
        return FM_OK;
    }

    template<typename T>
    class FM_minus_fun : public std::binary_function<T,T,T>

    public:
    int operator()(const T& lhs, const T& rhs, T* res) const
    {
        *res = rhs - lhs;
        return FM_OK;
    }

    template<typename T>
    class FM_multiplies_fun : public std::binary_function<T,T,T>

    public:
    int operator()(const T& lhs, const T& rhs, T* res) const
    {
        *res = lhs * rhs;
        return FM_OK;
    }

    template<typename T>
    class FM_divides_fun : public std::binary_function<T,T,T>

    public:
    int operator()(const T& lhs, const T& rhs, T* res) const
    {
        *res = lhs / rhs;
        return FM_OK;
    }

    template<typename S, typename T>
    class FM_add_bet : public std::binary_function<S,T,T>

    public:
    S operator()(const S& s, const T& t) const
    {
        return S(s) + T(t);
    }

    template<typename S, typename T>
    class FM_sub_bet : public std::binary_function<S,T,T>

    public:
    S operator()(const S& s, const T& t) const
    {
        return S(s) - T(t);
    }

    template<typename S, typename T>
    class FM_mul_bet : public std::binary_function<S,T,T>

    public:
    S operator()(const S& s, const T& t) const
    {
        return S(s) * T(t);
    }

    template<typename S, typename T>
    class FM_div_bet : public std::binary_function<S,T,T>

    public:
    S operator()(const S& s, const T& t) const
    {
        return S(s) / T(t);
    }

    template<typename S, typename T>
    class FM_dot_bet : public std::binary_function<S,T,T>

    public:
    S operator()(const S& s, const T& t) const
    {
        return S(s) * T(t);
    }

    template<typename S, typename T>
    class FM_cross_bet : public std::binary_function<S,T,T>

    public:
    S operator()(const S& s, const T& t) const
    {
        return S(s) * T(t);
    }

    template<typename T>
    class FM_abs_bet : public std::unary_function<T,T>

    public:
    T operator()(const T& t) const
    {
        return std::abs(t);
    }

    template<typename T>
    class FM_min_bet : public std::binary_function<T,T,T>

    public:
    T operator()(const T& lhs, const T& rhs) const
    {
        return (lhs < rhs) ? lhs : rhs;
    }

    template<typename T>
    class FM_max_bet : public std::binary_function<T,T,T>

    public:
    T operator()(const T& lhs, const T& rhs) const
    {
        return (lhs > rhs) ? lhs : rhs;
    }

    template<int N, typename T>
    class FM_mag_fun : public std::unary_function<FM_vector<N,T>,T>

    public:
    T operator()(const FM_vector<N,T>& v) const
    {
        return FM_mag(v);
    }

    template<int N, typename T>
    class FM_brackets_fun : public std::unary_function<FM_vector<N,T>,T>

    public:
    T operator()(const FM_vector<N,T>& v) const
    {
        return v[0];
    }

    template<int M, int N, typename T>
    class FM_slice_brackets_fun : public std::unary_function<FM_vector<M,T>,FM_vector<N,T>>

    public:
    FM_vector<N,T> operator()(const FM_vector<M,T>& v) const
    {
        return v.slice(0, N);
    }

    template<int M, int N, typename T>
    class FM_brackets_fun : public std::unary_function<FM_vector<M,T>,T>

    public:
    T operator()(const FM_vector<M,T>& v) const
    {
        return v[0];
    }
int operator()(const FM_vector<M, T>& v, FM_vector<N, T>* res) const {
    *res = FM_vector<N, T>(static_cast<const T*>(v) + index);
    return FM_OK;
}

private:
    const int index;
};

template<typename T>
class FM_swap_endian_fun : public std::unary_function<T,T>
{
public:
    int operator() (const T& t, T* res) const {
        union {
            T t;
            char chars[8];
        } u;
        u.t = t;
        switch(sizeof(T)) {
        case 1:
            break;
        case 2:
            c = u.chars[0];
            u.chars[0] = u.chars[1];
            u.chars[1] = c;
            break;
        case 4:
            c = u.chars[0];
            u.chars[0] = u.chars[3];
            u.chars[3] = c;
            c = u.chars[1];
            u.chars[1] = u.chars[2];
            u.chars[2] = c;
            break;
        case 8:
            c = u.chars[0];
            u.chars[0] = u.chars[7];
            u.chars[7] = c;
            c = u.chars[1];
            u.chars[1] = u.chars[6];
            u.chars[6] = c;
            c = u.chars[2];
            u.chars[2] = u.chars[5];
            u.chars[5] = c;
            c = u.chars[3];
            u.chars[3] = u.chars[4];
            u.chars[4] = c;
            break;
        default:
            abort();
            return FM_OK;
        }
        *res = u.t;
        return FM_OK;
    }
};

template<>
class FM_swap_endian_fun<int> : public std::unary_function<int,int>
{
public:
    int operator() (const int& i, int* res) const {
        union {
            int i;
            char chars[4];
        } u;
        u.i = i;
        switch(sizeof(int)) {
        case 1:
            break;
        case 2:
            c = u.chars[0];
            u.chars[0] = u.chars[1];
            u.chars[1] = c;
            break;
        case 4:
            c = u.chars[0];
            u.chars[0] = u.chars[3];
            u.chars[3] = c;
            c = u.chars[1];
            u.chars[1] = u.chars[2];
            u.chars[2] = c;
            break;
        case 8:
            c = u.chars[0];
            u.chars[0] = u.chars[7];
            u.chars[7] = c;
            c = u.chars[1];
            u.chars[1] = u.chars[6];
            u.chars[6] = c;
            c = u.chars[2];
            u.chars[2] = u.chars[5];
            u.chars[5] = c;
            c = u.chars[3];
            u.chars[3] = u.chars[4];
            u.chars[4] = c;
            break;
        default:
            abort();
            return FM_OK;
        }
        *res = u.i;
        return FM_OK;
    }
};

template<>
class FM_swap_endian_fun<signed> : public std::unary_function<signed,signed>
{
public:
    int operator() (const signed& i, signed* res) const {
        union {
            signed i;
            char chars[4];
        } u;
        u.i = i;
        switch(sizeof(signed)) {
        case 1:
            break;
        case 2:
            c = u.chars[0];
            u.chars[0] = u.chars[1];
            u.chars[1] = c;
            break;
        case 4:
            c = u.chars[0];
            u.chars[0] = u.chars[3];
            u.chars[3] = c;
            c = u.chars[1];
            u.chars[1] = u.chars[2];
            u.chars[2] = c;
            break;
        case 8:
            c = u.chars[0];
            u.chars[0] = u.chars[7];
            u.chars[7] = c;
            c = u.chars[1];
            u.chars[1] = u.chars[6];
            u.chars[6] = c;
            c = u.chars[2];
            u.chars[2] = u.chars[5];
            u.chars[5] = c;
            c = u.chars[3];
            u.chars[3] = u.chars[4];
            u.chars[4] = c;
            break;
        default:
            abort();
            return FM_OK;
        }
        *res = u.i;
        return FM_OK;
    }
};
template <typename T>
int FMfread(FILE* fp, const T* dat, size_t n_items, bool swap_endian, bool fortran)
{
    size_t n_remaining = n_items;
    size_t n_to_read = n_remaining;
    while (n_remaining > 0) {
        size_t n_to_read = n_remaining;
        if (fortran) {
            size_t n_bytes_fortran;
            FM fwrite(fp, &n_bytes_fortran, 1, swap_endian);
            if (res != FM OK)
                return res;
            n_bytes_fortran = n_to_read * sizeof(T);
        }
        size_t n_read = FM fread(fp, dat, n_to_read);
        if (res != FM OK)
            return res;
        n_remaining -= n_to_read;
    }
    return FM OK;
}

int FMfwrite(FILE* fp, const T* dat, size_t n_items, bool swap_endian, bool fortran)
{
    size_t n_remaining = n_items;
    size_t n_to_read = n_remaining;
    while (n_remaining > 0) {
        size_t n_to_read = n_remaining;
        if (fortran) {
            size_t n_bytes_fortran;
            FM fwrite(fp, &n_bytes_fortran, 1, swap_endian);
            if (res != FM OK)
                return res;
            n_bytes_fortran = n_to_read * sizeof(T);
        }
        size_t n_read = FM fwrite(fp, dat, n_to_read);
        if (res != FM OK)
            return res;
        n_remaining -= n_to_read;
    }
    return FM OK;
}

template <typename T>
int FMftime(FILE* fp, const T* dat, size_t n_items, bool swap_endian, bool fortran)
{
    size_t n_remaining = n_items;
    size_t n_to_read = n_remaining;
    while (n_remaining > 0) {
        size_t n_to_read = n_remaining;
        if (fortran) {
            size_t n_bytes_fortran;
            FM fwrite(fp, &n_bytes_fortran, 1, swap_endian);
            if (res != FM OK)
                return res;
            n_bytes_fortran = n_to_read * sizeof(T);
        }
        size_t n_read = FM fwrite(fp, dat, n_to_read);
        if (res != FM OK)
            return res;
        n_remaining -= n_to_read;
    }
    return FM OK;
}

template <typename T>
int FMftime(FILE* fp, const T* dat, size_t n_items, bool swap_endian, bool fortran)
{
    size_t n_remaining = n_items;
    size_t n_to_read = n_remaining;
    while (n_remaining > 0) {
        size_t n_to_read = n_remaining;
        if (fortran) {
            size_t n_bytes_fortran;
            FM fwrite(fp, &n_bytes_fortran, 1, swap_endian);
            if (res != FM OK)
                return res;
            n_bytes_fortran = n_to_read * sizeof(T);
        }
        size_t n_read = FM fwrite(fp, dat, n_to_read);
        if (res != FM OK)
            return res;
        n_remaining -= n_to_read;
    }
    return FM OK;
}

/* Copyright (c) 2000
   Advanced Management Technology, Incorporated
   Permission is hereby granted, free of charge, to any person obtaining a copy of this software
   and associated documentation files (the "Software"), to deal in the Software without restriction,
   including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense,
   and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so,
   subject to the following conditions:
   The above copyright notice and this permission notice shall be included in all copies or substantial
   portions of the Software.
   THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
   INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR
   PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
   CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
   FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. */
and/or sell copies of the Software, and to permit
persons to whom the Software is furnished to do so,
subject to the following conditions:

* The above copyright notice and this permission
* notice shall be included in all copies or substantial
* portions of the Software.

* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
* OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
* LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
* FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
* EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
* FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
* IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
* FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
* THE USE OR OTHER DEALINGS IN THE SOFTWARE.

*/

#ifndef FM_IRREGULAR_INTERVAL_H
#define FM_IRREGULAR_INTERVAL_H

/*
 * NAME: FM irregular interval.h
 * WRITTEN BY:
 * Patrick Moran pmoran@nas.nasa.gov
 */
#include "FMstructured mesh.h"

class FM irregular interval : public FM structured mesh<l,l>
{
public:
    const FM coord* const coordinates;
    const bool delete suppression;
    FM irregular interval(FM u32 d, const FM coord* c,
        bool ds : false,
        FM properties cache* pc : 0) :
        FM structured mesh<l,l>(d, pc), coordinates(c),
        delete suppression(ds)
    for (FM u32 i : 0; i < d i; i++)
    if (!(c[i] < c[i + i])) {
        FM ostringstre_a err;
        err << "FM irregular interval::FM irregular interval: ";
        err << "coordinates must be strictly
        ascending";
        throw std::logic error(err.str());
    }
    virtual "FM irregular interval()"
    {
        if (!delete suppression)
            delete [] coordinates;
    }
    virtual std::ostream& str(std::ostream& o) const
        return o << "FM irregular interval";
    virtual int at cell(const FM cell* c, std::vector<FM vector<l,FM coord> >* vals) const
        FM u32 n indices;
        FM u64 indices[2];
        c->structured mesh vertex indices(this, &n indices, indices);
        for (FM u32 i : 0; i < n _ndices; i++)
            vals >> push back(coordinates[indices[i]]);
        return FM OK;
    virtual int at cell(const FM cell* c, FM vector<l,FM coord>* vals) const
        FM u32 n indices;
        FM u64 indices[2];
        c->structured mesh vertex indices(this, &n indices, indices);
        for (FM u32 i : 0; i < n _ndices; i++)
            vals-> coordinates[indices[i]];
        return FM OK;
    virtual int phys to base(const FM phys<l>& p, FM context*, FM base<l>* b,
        FM ptr<FM structured B cell<l> >* sc : 0) const
    if (p[0] < coordinates[0] II p[0] > coordinates[dimensions[0] i])
        return FM OUT OF BOUNDS;
    FM u32 lo = 0, hi = dimensions[0] i;
    while (hi - lo > 1) {
        // assert(coordinates[lo] <= p[0] && p[0] <= coordinates[hi])
        int mid = (lo + hi) / 2;
        if (p[0] > coordinates[mid])
            lo = mid;
        else
            hi = mid;
    }
    FM u32 index = (lo < dimensions[0] i) ? lo : lo i;
    return std::pair<FM vector<l,FM coord>,FM vector<l,FM coord> >
        (coordinates[0], coordinates[dimensions[0] i]);
}

};

*/

// Copyright (c) 2000
* Advanced Management Technology, Incorporated
*
* Permission is hereby granted, free of charge,
* to any person obtaining a copy of this software
* and associated documentation files (the "Software"),
* to deal in the Software without restriction,
* including without limitation the rights to use,
/* copy, modify, merge, publish, distribute, sublicense, */
/* and/or sell copies of the Software, and to permit */
/* persons to whom the Software is furnished to do so, */
/* subject to the following conditions: */
/* */
/* The above copyright notice and this permission */
/* notice shall be included in all copies or substantial */
/* portions of the Software. */
/* */
/* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY */
/* OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT */
/* LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS */
/* FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO */
/* EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE */
/* FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER */
/* IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING */
/* FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR */
/* THE USE OR OTHER DEALINGS IN THE SOFTWARE. */

// Emacs mode * c++ *

#ifndef FM_ITER_H
#define FM_ITER_H

/*
• NAME: FM iter.h
• WRITTEN BY:
Patrick Moran
pmoran@nas.nasa.gov
*/

#include "FMsharedobject.h"
#include "FMcell.h"

enum FM iter attr enum
{
    FM_ITER_ATTR_CELL_DIMENSION,
    FM_ITER_ATTR_CELL_TYPE,
    FM_ITER_ATTR_TIME,
    FM_ITER_ATTR_SUBMESH_ID,
    FM_ITER_ATTR_AXIS_BEGIN,
    FM_ITER_ATTR_AXIS_END,
    FM_ITER_ATTR_AXIS_STIDE,
    FM_ITER_ATTR_SIMPLIFIED_DECOMPOSITION
};

class FM iter attr : public FM shared object
{
    public:
        const FM iter attr enum attr;

        static virtual FM iter attr();

    private:
        static FM iter attr enum attr;

    typedef std::vector<FM ptr<FM iter attr>> FM iter attr s;

    std::ostream& operator<<(std::ostream& o, const FM iter attr s& ia) const
    {
        o << "FM iter attr(";
        for(FM u32 i = 0; i < ia.size(); i++)
        {
            if(i > 0)
                o << ", ";
            o << *ia[i];
        }
        return o << ")";
    }

    class FM cell dimension iter attr : public FM iter attr
    {
    public:
        const FM u32 cell dimension;

        static virtual FM cell dimension iter attr();

    private:
        static FM cell dimension iter attr s;

    typedef std::vector<FM iter attr> FM iter attr s;

    std::ostream& operator<<(std::ostream& o, const FM iter attr s& ia) const
    {
        o << "FM cell dimension iter attr(";
        for(FM u32 i = 0; i < ia.size(); i++)
        {
            if(i > 0)
                o << ", ";
            o << *ia[i];
        }
        return o << ")";
    }

    class FM cell type iter attr : public FM iter attr
    {
    public:
        const FM cell type enum cell type;

        static virtual FM cell type iter attr();

    private:
        static FM cell type iter attr s;

    typedef std::vector<FM iter attr> FM iter attr s;

    std::ostream& operator<<(std::ostream& o, const FM iter attr s& ia) const
    {
        o << "FM cell type iter attr(";
        for(FM u32 i = 0; i < ia.size(); i++)
        {
            if(i > 0)
                o << ", ";
            o << *ia[i];
        }
        return o << ")";
    }

    class FM time iter attr : public FM iter attr
    {
    public:
        const FM time<FM u32> time;

        static virtual FM time iter attr();

    private:
        static FM time iter attr s;

    typedef std::vector<FM iter attr> FM iter attr s;

    std::ostream& operator<<(std::ostream& o, const FM iter attr s& ia) const
    {
        o << "FM time iter attr(";
        for(FM u32 i = 0; i < ia.size(); i++)
        {
            if(i > 0)
                o << ", ";
            o << *ia[i];
        }
        return o << ")";
    }

    class FM submesh id iter attr : public FM iter attr
    {
    public:
        const FM submesh id submesh id;

        static virtual FM submesh id iter attr();

    private:
        static FM submesh id iter attr s;

    typedef std::vector<FM iter attr> FM iter attr s;

    std::ostream& operator<<(std::ostream& o, const FM iter attr s& ia) const
    {
        o << "FM submesh id iter attr(";
        for(FM u32 i = 0; i < ia.size(); i++)
        {
            if(i > 0)
                o << ", ";
            o << *ia[i];
        }
        return o << ")";
    }

    class FM axis begin iter attr : public FM iter attr
    {
    public:
        const FM u32 axis, index;

        static virtual FM axis begin iter attr();

    private:
        static FM axis begin iter attr s;

    typedef std::vector<FM iter attr> FM iter attr s;

    std::ostream& operator<<(std::ostream& o, const FM iter attr s& ia) const
    {
        o << "FM axis begin iter attr(";
        for(FM u32 i = 0; i < ia.size(); i++)
        {
            if(i > 0)
                o << ", ";
            o << *ia[i];
        }
        return o << ")";
    }

    class FM axis end iter attr : public FM iter attr
    {
    public:
        const FM u32 axis, index;

        static virtual FM axis end iter attr();

    private:
        static FM axis end iter attr s;

    typedef std::vector<FM iter attr> FM iter attr s;

    std::ostream& operator<<(std::ostream& o, const FM iter attr s& ia) const
    {
        o << "FM axis end iter attr(";
        for(FM u32 i = 0; i < ia.size(); i++)
        {
            if(i > 0)
                o << ", ";
            o << *ia[i];
        }
        return o << ")";
    }

    class FM iter attr enum
    { /* define FM iter attr enum */
    }

    class FM iter attr
    { /* define FM iter attr */
    }

*/

#endif /* FM ITER_H */
public:
  const FM u32 axis, index;
  FM axis
  end iter attr(FM u32 a, FM u32 i) :
  FM iter attr(FM ITER ATTR AXIS END),
  axis(a),
  index(i) {

  virtual std::ostream& str(std::ostream& o) const
  return o << "FM axis
  end iter attr("
  << axis << ", " << index "\n  ");
  
  class FM_axis_stride_iter_attr : public FM_iter_attr
  public:
  const FM u32 axis, stride;
  FM_axis_stride_iter_attr(FM u32 a, FM u32 s) :
  FM_iter_attr(FM ITER ATTR_AXIS_STRIDE),
  axis(a),
  stride(s) {

  virtual std::ostream& str(std::ostream& o) const
  return o << "FM_axis_stride_iter_attr("
  << axis << ", " << stride << ");
  
  class FM simplicial_decomposition_iter_attr : public FM_iter_attr
  public:
  const FM u32 simplicial_decomposition;
  FM simplicial_decomposition_iter_attr(FM u32 sd) :
  FM_iter_attr(FM ITER ATTR_SIMPLICIAL_DECOMPOSITION),
  simplicial_decomposition(sd) {

  virtual std::ostream& str(std::ostream& o) const
  return o << "FM simplicial_decomposition_iter_attr("
  << simplicial_decomposition << ");
  
  class FM_iter_impl
  public:
  virtual "FM_iter_impl() {
  virtual FM_iter_impl* copy() const 0;
  virtual const FM cell* advance() 0;
  virtual const FM cell* dereference() const 0;

  virtual std::ostream& str(std::ostream& o) const
  return o << "FM iter impl";
  
  class FM_iter
  public:
  FM_iter() : impl(0), cell(0) {
  FM_iter(FM_iter_impl* i) : impl(i), cell(impl->dereference()) {
  FM_iter(const FM_iter& iter)
  impl=impl->copy(); cell=impl->dereference();

  FM_iter operator=(const FM_iter& rhs)
  impl=rhs.impl->copy();
  cell=rhs.impl->dereference();
  return this;
  
  "FM_iter()
  if (impl) delete impl;

  inline const FM cell* operator++()
  return cell = impl->advance();

  void operator=(const (void) operator++());

  inline const FM cell* operator*() const
  return cell;

  inline bool done() const
  return cell == 0;

  friend bool operator==(const FM_iter& lha, const FM_iter& rha)
  { return lha.cell == rha.cell; }

  friend bool operator!=(const FM_iter& lha, const FM_iter& rha)
  { return !operator==(lha, rha); }

  private:
  FM_iter_impl* impl;
  const FM cell* cell;
  };

// Copyright (c) 2000
// Advanced Management Technology, Incorporated
// Permission is hereby granted, free of charge,
// to any person obtaining a copy of this software
// and associated documentation files (the "Software"),
// to deal in the Software without restriction,
```c++
// Written by: Patrick Moran

#include "FM_vector.h"

// WRITTEN BY: Patrick Moran
// NAME: FM matrix.h

template <int M, int N, int P, typename T>
operator*(const FM vector<M, FM vector<N, T> >& lhs,
          const FM vector<P, T>& rhs)
{
    T tmp[N];
    for (int n = 0; n < N; n++) {
        T sum = (T) 0;
        for (int m = 0; m < M; m++) {
            sum += lhs[m][n] * rhs[n][m];
        }
        tmp[n] = sum;
    }
    return FM vector<P, T>(tmp);
}

template <typename T>
operator*(const FM vector<4, FM vector<4, T> >& lhs,
          FM vector<4, T>
          )
{
    return FM vector<4, T>(FM dot(lhs[0], rhs),
                           FM dot(lhs[1], rhs),
                           FM dot(lhs[2], rhs),
                           FM dot(lhs[3], rhs));
}

template <typename T>
operator*(const FM vector<3, FM vector<3, T> >& lhs,
          FM vector<3, T>
          )
{
    return FM vector<3, T>(FM dot(lhs[0], rhs),
                           FM dot(lhs[1], rhs),
                           FM dot(lhs[2], rhs));
}

template <typename T>
operator*(const FM vector<2, FM vector<2, T> >& lhs,
          FM vector<2, T>
          )
{
    return FM vector<2, T>(FM dot(lhs[0], rhs),
                           FM dot(lhs[1], rhs));
}

template <typename T>
operator*(const FM vector<M, FM vector<N, T> >& lhs,
          FM vector<N, T>
          )
{
    T sum;
    for (int n = 0; n < N; n++) {
        sum += FM det(lhs[n] * rhs[n]);
    }
    return FM vector<M, T>(sum);
}

template <typename T>
operator*(const FM vector<4, FM vector<4, T> >& lhs,
          FM vector<4, T>
          )
{
    return FM vector<4, T>(FM dot(lhs[0], rhs),
                           FM dot(lhs[1], rhs),
                           FM dot(lhs[2], rhs),
                           FM dot(lhs[3], rhs));
}
```
```cpp
int FM inv(const FM vector<3, FM vector<3, T>> & in, T det = (T) 0);
```

```cpp
int FM inv(const FM vector<N, FM vector<N, T>> & in, T det = (T) 0);
```

```cpp
int FM inv(const FM vector<N, FM vector<N, T>> & in, T det = (T) 0);
```

```cpp
T det = FM det(in);
if (det == (T) 0)
  return 0;

T minorl = in[0][l] * in[2][l] * in[3][l] + in[3][0] * in[2][0] * in[3][0];
T minor2 = in[0][2] * in[1][0] * in[3][1] + in[3][0] * in[1][0] * in[3][0];
T minor3 = in[0][3] * in[1][0] * in[2][1] + in[3][0] * in[1][0] * in[2][1];

T rlr3 = in[l][3] in[l][2] in[l][1];
T rlr2 = in[l][2] in[l][1] in[l][0];
T r0rl = in[0][l] in[l][1] in[l][0];
```

```cpp
T inv det = Tinv det (T) 1
```

```cpp
T inv det = Tinv det (T) 1
```

```cpp
T inv det = Tinv det (T) 1
```

```cpp
T inv det = Tinv det (T) 1
```

```cpp
T inv det = Tinv det (T) 1
```

```cpp
for (int row = 0; row < N; row++) {
  FM vector<N, FM vector<N, T>> res;
  for (int col = 0; col < N; col++) {
    res[col][row] = row == col ? one : zero;
  }
  return res;
}
```

```cpp
void FM identity (FM vector<N, FM vector<N, T>>* out)
```

```cpp
typedef FM vector<4, FM vector<4, float>> FM matrix44f;
typedef FM vector<3, FM vector<3, double>> FM matrix33d;
typedef FM vector<3, FM vector<3, float>> FM matrix33f;
typedef FM vector<3, FM vector<2, float>> FM matrix32f;
typedef FM vector<2, FM vector<3, float>> FM matrix23f;
```
virtual FM_ptr<FM_shared_object>
get_aux(const std::string& key, FM_u32 pass,
const FM_time<FM_u32>* t, const FM_submesh_id* sid) const
{
if (key == "bounding box") {
std::pair<FM_vector<D, FMcoord>, FM_vector<D, FMcoord> > min_max =
get_bounding_box(t, sid);
std::vector<FM_ptr<FM_shared_object> > lo_values(D);
std::vector<FM_ptr<FM_shared_object> > hi_values(D);
for (int j = 0; j < D; j++) {
lo_values[j] = new FM_simple_value<FMcoord>(min_max.first[j]);
hi_values[j] = new FM_simple_value<FMcoord>(min_max.second[j]);
}
return new FM_tuple_value(new FM_tuple_value(lo_values),
new FM_tuple_value(hi_values));
}
return FM_mesh::get_aux(key, pass, t, sid);
}
virtual std::set<std::string>
get_property_names(const std::set<std::string>& property_names,
const FM_time<FM_u32>* t,
const FM_submesh_id* sid) const
{
property_names.insert("bounding box");
return FM_mesh::get_property_names(property_names, t, sid);
}
}
const char* FM_orientation_names[4] = {"outside", "orientation 0", "inside", "orientation i"};

return FM_sign(results[largest result]);
}

if (verbosity > 0) {
    int res = FM_sign(results[largest_result]);
    std::cerr << "FM orient(" << a << ", " << b << ", " << c << ", " << d << ", " << e << ") returning " << res << std::endl;
}

// Compute orientation of a with respect to quadrilateral abcd
// by treating abcd as two triangles: abc and acd.

Compute orientation of a with respect to quadrilateral abcd
// by treating abcd as two triangles: abc and acd.

FM vector<3,double> obi = b - i;
FM vector<3,double> odi = d - i;
FM vector<3,double> abc = a - b;
FM vector<3,double> acd = a - c;
FM vector<3,double> ede = d - e;

double result[4];
result[0] = FM dot(obi, FM cross(oca, obc));
result[1] = FM dot(odi, FM cross(oda, odc));
result[2] = FM dot(abi, FM cross(abd, acd));
result[3] = FM dot(abi, FM cross(abd, acd));

int i, largest_result = 0;
for (i = i; i < 4; i++) {
    if (FM abs(result[i]) > FM abs(result[largest_result]))
        largest_result = i;
}

std::cerr << "largest magnitude result " << 
    "(triangle " << triangle_names[largest_result] << ": ");

std::cerr << 
    "returning " << res << std::endl;
return res;

}
// Copyright (c) 2000
// Advanced Management Technology, Incorporated

// Permission is hereby granted, free of charge,
// to any person obtaining a copy of this software
// and associated documentation files (the "Software"),
// to deal in the Software without restriction,
// including without limitation the rights to use,
// copy, modify, merge, publish, distribute, sublicense,
// and/or sell copies of the Software, and to permit
// persons to whom the Software is furnished to do so,
// subject to the following conditions:

// The above copyright notice and this permission
// notice shall be included in all copies or substantial
// portions of the Software.

// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
// OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
// LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
// FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
// EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
// FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
// IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
THE USE OR OTHER DEALINGS IN THE SOFTWARE.

// Copyright (c) 2000
// Advanced Management Technology, Incorporated

// Permission is hereby granted, free of charge,
// to any person obtaining a copy of this software
// and associated documentation files (the "Software"),
// to deal in the Software without restriction,
// including without limitation the rights to use,
// copy, modify, merge, publish, distribute, sublicense,
// and/or sell copies of the Software, and to permit
// persons to whom the Software is furnished to do so,
// subject to the following conditions:

// The above copyright notice and this permission
// notice shall be included in all copies or substantial
// portions of the Software.

// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
// OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
// LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
// FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
// EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
// FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
// IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
THE USE OR OTHER DEALINGS IN THE SOFTWARE.

// Copyright (c) 2000
// Advanced Management Technology, Incorporated

// Permission is hereby granted, free of charge,
// to any person obtaining a copy of this software
// and associated documentation files (the "Software"),
// to deal in the Software without restriction,
// including without limitation the rights to use,
// copy, modify, merge, publish, distribute, sublicense,
// and/or sell copies of the Software, and to permit
// persons to whom the Software is furnished to do so,
// subject to the following conditions:

// The above copyright notice and this permission
// notice shall be included in all copies or substantial
// portions of the Software.

// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
// OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
// LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
// FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
// EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
// FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
// IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
THE USE OR OTHER DEALINGS IN THE SOFTWARE.

}
```cpp
#include "FM_structured_mesh.h"

class FM_product_mesh : public FM_structured_mesh<0,0>
{
public:
  const std::vector<FM_ptr<FM_structured_mesh<1,1> > > & axes;

  FM_product_mesh(const std::vector<FM_ptr<FM_structured_mesh<1,1> > >& a, FM_properties_cache* pc = 0) :
    FM_structured_mesh<0,0>(pc),
    axes(a)
  {
    if (axes.size() != size_t(B)) {
      FM ostringstream err;
      err << "FM_product_mesh<" << B << "," << m << ">::FM_product_mesh: expecti" << B << " axes,
           got " << axes.size();
      throw std::logic_error(err.str());
    };
    FM vector<l,FM u32> dimension;
    for (int i = 0; i < B; i++) {
      dimension = axes[i] >get_base_dimensions();
      const cast<FM u32&>(dimensions[i]) = dimension[0];
    }
    init(dimensions);
  }

private:
  FM_product_mesh(const FM vector<B,FM u32>& d, FM_properties_cache* pc = 0) :
    FM_structured_mesh<B,m>(d, pc),
    axes(B)
  {
    // axes filled in by derived class constructor
  }

public:
  virtual std::ostream& str(std::ostream& o) const
  {
    return o << "FM_product_mesh<" << B << "," << m << ">;"
  }

  virtual int at_cell(const FM_cell* c, std::vector<FM vector<m,FM coord> >* vals) const
  { const FM_structured_cell<B>* sc =
    dynamic_cast<const FM_structured_cell<B>*>(c);
    if (sc == 0)
      FM throw bad_cell_argument(this, "at_cell", c);
    FM base<l,FM u32> b(c >get_time(), c >get_submesh_id());
    FM u32 d = sc >get_dimension();
    FM u32 n = c >is_subsimplex() ? d + 1 : FM_pow 2(d);
    vals->resize(previous_size + n);
    return FM_product_mesh::at_cell(c, &(*vals)[previous_size]);
  }

  virtual int at_cell(const FM_cell* c, FM vector<m,FM coord>* vals) const
  { const FM_structured_cell<B>* sc =
    dynamic_cast<const FM_structured_cell<B>*>(c);
    if (sc == 0)
      FM throw bad_cell_argument(this, "at_cell", c);
    FM base<l,FM u32> b(c >get_time(), c >get_submesh_id());
    FM u32 d = sc >get_dimension();
    FM vector<l,FM coord> coordinate;
    if (sc >is_subsimplex()) {
      //assert(B == 2 || B == 3);
      FM u32 subid = sc >get_submesh_id();
      FM u32 n = d + 1;
      for (FM u32 i = 0; i < n; i++)
        if (FM_structured_hexahedron_subfaces[d][subid][i] & mask)
          b[0]++;
      int res = axes[0] >at_base(b, coordinate);
      if (res != FM_OK) return res;
      vals[i][0] = coordinate[0];
      mask <<= 1;
    };
    for ( ; j < D; j++)
      vals[i][j] = FM coord(0);
    if (res != FM_OK) return res;
  }

  virtual std::ostream& operator<<(std::ostream& o) const
  {
    return o << "FM_product_mesh<" << B << "," << m << ">;"
  }
};
```
Portions of the Software.

Copyright (c) 2000
Advanced Management Technology, Incorporated

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

1. The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

2. The Software is provided "AS IS", without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the Software or the use or other dealings in the Software.

* * *

/**
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software")
 * to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OF OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
#endif
template <int B, int D> class FM regular interval : public FM structured mesh<l,l> {

public:

const FM coord origin;
const FM coord spacing;

FM regular_interval(FM u32 d, const FM coord spacing; const FM coord origin; 
FM structured mesh<l,l>(d, pc), origin(o), spacing(s)) {
}

virtual std::ostream& str(std::ostream& o) const

return o << "FM regular interval"; ]

virtual int

at(const FM cell* c, std::vector<FM vector<l,FM coord> >* vals) const

for (FM u32 i = 0; i < n _ndices; i++)
c _structured mesh vertex indices(this, &n indices, indices);

FM u64 indices[2];
FM u32 n indices;

for (FM u32 i = 0; i < n _ndices; i++)
c _structured mesh vertex indices(this, &n indices, indices);

FM u64 indices[2];
FM u32 n indices;

return FM OK;

virtual std::pair<FM vector<l,FM coord>,FM vector<l,FM coord> >

get_bounding_box(const FM time<FM u32>* = O, const FM _ubmesh id* = 0) const

return std::pair<FM vector<l,FM coord>,FM vector<l,FM coord> >;

if (sc)

if (_es != FM OK) return res;

for (FM u32 i = 0; i < n _ndices; i++)
c _structured mesh vertex indices(this, &n indices, indices);

FM u64 indices[2];
FM u32 n indices;

vals >push back(origin + FM coord(indices[i]) * spacing);

return FM OK;

FM properties cache* pc = 0) :

FM coord s = FM coord(1),
FM coord o = FM coord(0),
FM regular mesh (const FM vector<B,FM u32>& d, FMproperties cache* pc 0) :

FM product mesh<B,m>(d, pc)

return o << "FM regular mesh";

for (int i = 0; i < B; i++)
cast<FM ptr<FM structured mesh<l,l> >&)(axes[i]) =

new FMregularinterval(d[i]);

return FM OK;

new FM regular interval(1,1);

return o << "FM regular interval";

return res;

return res;

for (int i = 0; i < B; i++)
cast<FM ptr<FM structured mesh<l,l> >&)(axes[i]) =

new FMregularinterval(d[i]);

return FM OK;

FM regular interval(FM u32 d, const FM coord spacing; const FM coord origin; 
FM structured mesh<l,l>(d, pc), origin(o), spacing(s)) {
}
const int FM_OK = 0;
const int FM_OUT_OF_BOUNDS = 1;
const int FM_BOUNDED_DATA = 2;
const int FM_POINTLOCATION_FAILED = 3;
const int FM_INTERPOLATION_ERROR = 4;
const int FM_NOTDEFINED = 6;
const int FM_POINTLOCATE WALKED OFFMESH = 7;
const int FMPOINTLOCATE STUCK = 8;
const int FMPOINTOUTSIDEBOUNDINGBOX = 9;

FMptr is a “smart pointer” for pointing at FMsharedobject’s. 

class FM_ptr
{
    // public:
    FM_ptr() : ptr(0) {
    }
    FM_ptr(const T* p) : ptr(p) {
        if (ptr) ptr->increment_reference();
    }
    template <typename S>
    FM_ptr(const FM_ptr<S>& p) :
    ptr(dynamic cast<const T*>(static cast<const S*>(p)))
    {
        if (ptr) {
            ptr->increment_reference();
        } else {
            if (static cast<const S*>(p)) {
                FM ostri_gstre_n err;
                err << "FMptr<" << typeid(T).name() << ">
                << "::operator (const FMptr<" << typeid(S).name() << ">
                << "&): bad dynamic cast";
                throw std::logic error(err.str());
            }
        }
    }
    FM_ptr(FM_ptr<T>& p) : ptr(static cast<const T*>(p))
    {
        if (ptr) ptr->increment_reference();
    }
    template <typename S>
    const FM_ptr<T>& operator (const FM_ptr<S>& rhs)
    {
        const T* tmp dynamic cast<const T*>(static cast<const S*>(rhs));
        if (static cast<const S*>(rhs) && !tmp) {
            FMostringstream err;
            err << "const FM ptr<" << typeid(T).name() << "&
            operator (const FMptr<" << typeid(S).name() << "&): bad dynamic cast";
            throw std::logic_error(err.str());
        }
        set (trap);
        return *this;
    }
    const FM_ptr<T>& operator (const FM_ptr<T>& rhs)
    {
        set(static cast<const T*>(rhs));
        return *this;
    }
    FM_ptr()
    if (ptr) {
        if (ptr->decrement_reference() == 0) {
            delete ptr;
        }
    }
    void set(const T* t)
    {
        if (t) t->increment_reference();
        if (ptr) {
            if (ptr->decrement_reference() == 0) {
                delete ptr;
            }
        }
        ptr = t;
    }
    // public:
    FM_ptr() : ptr(0) {
    }
    FM_ptr(const T* p) : ptr(p) {
        if (ptr) ptr->increment_reference();
    }
    template <typename S>
    FM_ptr(const FM_ptr<S>& p) :
    ptr(dynamic cast<const T*>(static cast<const S*>(p)))
    {
        if (ptr) {
            ptr->increment_reference();
        } else {
            if (static cast<const S*>(p)) {
                FM ostri_gstre_n err;
                err << "FMptr<" << typeid(T).name() << "::operator (const FMptr<" << typeid(S).name() << ">
                << "&): bad dynamic cast";
                throw std::logic error(err.str());
            }
        }
    }
    FM_ptr(FM_ptr<T>& p) : ptr(static cast<const T*>(p))
    {
        if (ptr) ptr->increment_reference();
    }
    template <typename S>
    const FM_ptr<T>& operator (const FM_ptr<S>& rhs)
    {
        const T* tmp dynamic cast<const T*>(static cast<const S*>(rhs));
        if (static cast<const S*>(rhs) && !tmp) {
            FMostringstream err;
            err << "const FM ptr<" << typeid(T).name() << "&
            operator (const FMptr<" << typeid(S).name() << "&): bad dynamic cast";
            throw std::logic_error(err.str());
        }
        set (trap);
        return *this;
    }
    const FM_ptr<T>& operator (const FM_ptr<T>& rhs)
    {
        set(static cast<const T*>(rhs));
        return *this;
    }
    FM_ptr()
    if (ptr) {
        if (ptr->decrement_reference() == 0) {
            delete ptr;
        }
    }
    void set(const T* t)
    {
        if (t) t->increment_reference();
        if (ptr) {
            if (ptr->decrement_reference() == 0) {
                delete ptr;
            }
        }
        ptr = t;
    }
#ifndef FM_SHAREDOBJECTWITHPROPERTIES_CACHE_H
#define FM_SHAREDOBJECTWITHPROPERTIES_CACHE_H

/*
 * NAME: FM shared object with properties cache.h
 * WRITTEN BY:
 * Patrick Moran pmoran@nas.nasa.gov
 */

#include "FMsharedobject.h"
#include "FMpropertiescache.h"

class FMsharedobject_with_properties_cache : public FMsharedobject {
public:
  FMsharedobject_with_properties_cache()
  FMsharedobject_with_properties_cache(FMpropertiescache* pc) :
    properties_cache(pc), pc(new FMpropertiescache()) {
    virtual FM_ptr<FMsharedobject> get(const std::string& key,
      const FMtime<FMu32>* t0, const FMsubmeshid* sid0) const
      // i. check cache
      FM_ptr<FMsharedobject> property = properties_cache->get(key, t0, sid0);
      if (property) return property;
      // 2. first pass: bottom up, through class lineage
      property_getaux(key, 0, t0, sid0);
      // 3. second pass: opportunity to query composed classes
      property = getpropertynamesaux(property_names, t0, sid0);
    if (!(((t0) && t0->defined()) II ((sid0) && sid0->defined()))) {
      const cast<FMpropertiescache*>(static cast<const FMpropertiescache*>(properties_cache)) >
        set(key, property, t0, sid0);
    }
    return property;
  }

  virtual void set(const std::string& key, FMsharedobject* property,
    const FMtime<FMu32>* t0, const FMsubmeshid* sid0) const
  {
    const_cast<FMpropertiescache*>(properties_cache)->
      set(key, property, t0, sid0);
  }

  virtual std::set<std::string>
    getpropertynamesaux(std::set<std::string>& property_names,
      const FMtime<FMu32>* t, const FMsubmeshid* sid) const
  {
    property_names =
      properties_cache->getpropertynamesaux(property_names, t, sid);
    return FMsharedobject::getpropertynamesaux(property_names, t, sid);
  }

protected:
  FM_ptr<FMpropertiescache> properties_cache;
};

#endif
The canonical vertex numbering for a structured hexahedron.

The vertex indices in the tables below are in terms of this hexahedron numbering.

```c
const FM u32 FM structuring_hexahedron_face[] = {
  FM u32 FM hsf3 5, FM u32 FM hsf3 6, FM u32 FM hsf3 7, FM u32 FM hsf3 8, FM u32 FM hsf3 9,
  FM u32 FM hsf3 0, FM u32 FM hsf3 i, FM u32 FM hsf3 2, FM u32 FM hsf3 3, FM u32 FM hsf3 4,
};
```

```c
const FM u32 FM structuring_hexahedron face2[] = {
  FM u32 FM hsf2 0, FM u32 FM hsf2 i, FM u32 FM hsf2 2, FM u32 FM hsf2 3, FM u32 FM hsf2 4,
};
```

```c
const FM u32 FM structuring_hexahedron face0[] = {
  FM u32 FM hsf0 0,
};
```

```c
#include "FM orient, h"
#include "FM combinatorics.h"
#include "FM mesh. h"
```

```c
#define FM STRUCTURED_MESH_H
```

```c
• WRITTEN BY: Patrick Moran
• NAME: FM structuring_mesh.h
```

```c
FM hsf3 5, FM hsf3 6, FM hsf3 7, FM hsf3 8, FM hsf3 9
FM hsf3 0, FM hsf3 i, FM hsf3 2, FM hsf3 3, FM hsf3 4,
FM hsfl 0, FM hsfl i, FM hsfl 2, FM hsfl 3, FM hsfl 4, FM hsfl 5
FM hfl 0, FM hfl i, FM hfl 2
```

```c
// Helper routines for FM structured_mesh.h: Most routines
```
template <int B>
std::vector<FM<point> & FM_cell>
FM_structured_mesh_adjsimplex(const FM_vector<FM, u32>& dimensions,
const FM_structured_cell<B>& sc) {
  assert(sc->get dimension() == B);
  assert(!sc->is subsimplex());

  std::vector<FM<point> & FM_cell> adjacent_cells;
  for (int i = 0; i < B; i++) {
    if (sc->get_index(i) < 0) {
      FM_vector<FM, u32> indices = sc->get_indices();
      --indices[i];
    }
  }

  return adjacent_cells;
}

protected:
const FM_vector<FM, u32> begin, end, stride;

}
template <int B>
class FM_structured_mesh_multi_alignment_iter_impl : public FM_iter_impl{
public:

FM_structured_mesh_multi_alignment_iter_impl(const FM_time<FM u32>& time, const FM_submesh_id& submesh_id,
FM u32 d, const FM_vector<B,FM u32>& s, const FM_vector<B,FM u32>& e):
structured_k_cell = new FM_structured_k_cell<B>(time, submesh_id, d, s, e),
FM structured mesh multi alignment iter impl

FM struct.

```cpp
//bool cell_alignment_given = false;
FM_u32 cell_alignment = 0;
FM_vector<B,FU_u32> begin_indices;
for (i = 0; i < B; i++)
begin_indices[i] = 0;
FM_vector<B,FU_u32> end_indices;
for (i = 0; i < B; i++)
end_indices[i] = 1;
FM_vector<B,FU_u32> strides;
if (cell type given) {
    //
    for (i = 0; i < B; i++)
    end_indices_given[i] = false;
    cell type trumps cell dimension
    simplicial decomposition = 0;
}
else if (cell dimension == B) {
    ii = new FM structured mesh begin cell impl<B>(
        time, submesh_id, cell_dimension, begin_indices, end_indices, strides);
    return ii.get_iter<B>();
}
else if (cell dimension == 2) {
    ii = new FM structured mesh 0 cell impl<B>(
        time, submesh_id, cell_dimension, begin_indices, end_indices, strides);
    return ii.get_iter<B>();
}
```
virtual int base_to_cell(const FH mesh *, const FH structured subcell <3> * sm) const{
    return FH structured mesh vertex indices(const FH mesh *, FH u32 nind, FH u64 ind[]) const;
}

virtual int phys_to_sub simplex(const FH mesh *, const FH structured subcell <3> * sm) const{
    return FH structured mesh vertex indices(const FH mesh *, FH u32 nind, FH u64 ind[]) const;
}

virtual int phys_to_sub simplex(const FH mesh *, const FH structured subcell <2> * sm) const{
    return FH structured mesh vertex indices(const FH mesh *, FH u32 nind, FH u64 ind[]) const;
}

virtual int phys_to_sub simplex(const FH mesh *, const FH structured subcell <1> * sm) const{
    return FH structured mesh vertex indices(const FH mesh *, FH u32 nind, FH u64 ind[]) const;
}

virtual int phys_to_sub simplex(const FH mesh *, const FH structured subcell <0> * sm) const{
    return FH structured mesh vertex indices(const FH mesh *, FH u32 nind, FH u64 ind[]) const;
}

const FH structured mesh<3,3> * sm = reinterpret_cast<const FH structured mesh<3,3>*>(m);

for (FH u32 i = 0; i < *nind; i++)
    ind[0] = index;
    ind[0] *= sm->dimensions[0];
    ind[0] += indices[0];

for (int i = B - 2; i >= 0; i--) {
    ind[i] = index * sm->dimensions[i];
    index *= indices[i];
}

for (int i = B - 2; i >= 0; i--) {
    ind[i] = index + sm->cube_offsets[i];
}

}
template <
    void FMstructured_subsimplex<3>::
    structured_mesh vertex indices(const FM mesh * m, FMu32* nind,
    FMu64 ind[]) const
    {
        const FMu32* vi = FMstructured_hexahedron_subfaces(dimension)[subid];
        nind[dimension] = 1;
        const FMstructured mesh[3,3]* sm =
            reinterpret_cast<const FMstructured mesh[3,3]*>(m);
        FMu64 index indices[2] * sm >dimensions[0] * sm >dimensions[1] +
            indices[2] * sm >dimensions[0] + indices[1];
        for (FMu32 i = 0; i < *nind; i++)
            ind[i] = index + sm >cubeoffsets[vi[i]];
    }

    #endif
    // Emacs mode * c++ *
    #ifndef FM SUBMESH IDH
    #define FM SUBMESH IDH
    /*
    • NAME: FM submesh id.h
    • WRITTEN BY:
    Patrick Moran
gpmoran@nas.nasa.gov
    */
    #include <iostream>
    #include <stdexcept>
    #include "FMtypes.h"
    class FMsubmesh id
    {
        public:
            FMsubmesh id() : id(1) {
            FMsubmesh id(int i) : id(i) {
            FMsubmesh id(FMu32 i) : id(int(i)) {
            void set(FMu32 i) { id = int(i); }
            inline bool defined() const { return id != 1; }
            friend std::ostream& operator<<(std::ostream& o, const FMsubmesh id& s)
                { if (s.defined())
                    o << s.id;
                else
                    o << "submesh id undefined";
            return o;
            }
            friend bool operator==(const FM submesh id& lhs, const FMsubmesh id& rhs)
                { return lhs.id == rhs.id; }
            friend bool operator<(const FMsubmesh id& lhs, const FM submesh id& rhs)
                { return lhs.id < rhs.id; }
            FMu32 index () const
                { if (!defined())
                    throw std::logic_error("attempting to access undefined submesh id");
                return FMu32(id);
            }
        }
        private:
            int id;
        
        /*
        Copyright (c) 2000
        Advanced Management Technology, Incorporated
        */
        /*
        Permission is hereby granted, free of charge,
        to any person obtaining a copy of this software
        and associated documentation files (the "Software"),
        to deal in the Software without restriction,
        including without limitation the rights to use,
        copy, modify, merge, publish, distribute, sublicense,
        and/or sell copies of the Software, and to permit
        persons to whom the Software is furnished to do so,
        subject to the following conditions:
        */
        /*
        The above copyright notice and this permission
        notice shall be included in all copies or substantial
        portions of the Software.
        */
        /*
        THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
        OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
        LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
        FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
        EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
        FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
        IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
        FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
        THE USE OR OTHER DEALINGS IN THE SOFTWARE.
        */
    #endif
    */
    #undef
#ifndef FMTIMEH
#define FMTIMEH

/*
 * NAME: FMtime.h
 * WRITTEN BY:
 * Patrick Moran
 * pmoran@nas.nasa.gov
 */
#include <iostream>
#include "FM types.h"

template <typename T>
class FMtime
{
public:
    FMtime () : value (undefined_value) {
    FMtime (T t) : value (t) {
        T& operator (const T& t) { value = t; }
        T get() const
        {
            if (!defined())
                throw std::logic_error("attempting to access undefined time");
            return value;
        }
        void set(T t) { value = t; }
        void set_undefined() { value = undefined_value; }
        inline bool defined() const { return value != undefined_value; }
    friend bool operator==(const FMtime lhs, const FMtime rhs);
    friend bool operator!=(const FMtime lhs, const FMtime rhs);
    friend bool operator< (const FMtime lhs, const FMtime rhs);
    T value;
    static const T undefined_value;
};

template <typename T>
const T FMtime<T>::undefined_value = T(-1L30);

template <>
const FM_u32 FMtime<FM_u32>::undefined_value = 0xFFFFFFFF;

template <typename T>
std::ostream& operator<<(std::ostream& rhs, const FMtime<T>& rhs)
{
    if (rhs.defined())
        return rhs << rhs.get();
    else
        return rhs << "<time undefined>";
}

/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 */
#endif

#endif
// Emacs mode * c++ *
ifndef FM_TYPES_H
#define FM_TYPES_H
/*
• NAME: FMtypes.h
• WRITTEN BY: Patrick Moran pmoran@nas.nasa.gov
*/
ifndef FMCOORD
#define FMCOORD
typedef float FM_coord;
endif
typedef unsigned short FM_u16;
typedef unsigned FM_u32;
typedef unsigned long FM_u64;
struct FMtruetype {]
struct FMfalse_type {]
template <int N, typename T> class FM_vector;
template <typename T>
struct FMtraits
{
    typedef T element_type;
    typedef FMfalse_type is scalar;
};
template <>
struct FMtraits<char>
{
    typedef char element_type;
    typedef FMfalse_type is scalar;
};
template <>
struct FMtraits<unsigned char>
{
    typedef unsigned char element_type;
    typedef FMtrue_type is scalar;
};
template <>
struct FMtraits<short>
{
    typedef short element_type;
    typedef FMtrue_type is scalar;
};
template <>
struct FMtraits<unsigned short>
{
    typedef unsigned short element_type;
    typedef FMtrue_type is scalar;
};
template <>
struct FMtraits<int>
{
    typedef int element_type;
    typedef FMtrue_type is scalar;
};
template <>
struct FMtraits<unsigned int>
{
    typedef unsigned int element_type;
    typedef FMtrue_type is scalar;
};
template <>
struct FMtraits<long>
{
    typedef long element_type;
    typedef FMtrue_type is scalar;
};
template <>
struct FMtraits<unsigned long>
{
    typedef unsigned long element_type;
    typedef FMtrue_type is scalar;
};
template <>
struct FMtraits<long long>
{
    typedef long long element_type;
    typedef FMtrue_type is scalar;
};
template <>
struct FMtraits<unsigned long long>
{
    typedef unsigned long long element_type;
    typedef FMtrue_type is scalar;
};
template <>
struct FMtraits<float>
{
    typedef float element_type;
    typedef FMtrue_type is scalar;
};
template <>
struct FMtraits<double>
{
    typedef double element_type;
    typedef FMtrue_type is scalar;
};
template <>
struct FMtraits<long double>
{
    typedef long double element_type;
    typedef FMtrue_type is scalar;
};
template <int N, typename T>
struct FM_traits<FMvector<N,T> >
{
    typedef typename FM_traits<T>::element_type element_type;
    typedef FMfalse_type is scalar;
};
/*
Copyright (c) 2000
Advanced Management Technology, Incorporated
*/
* Reclassification is hereby granted, free of charge,
* to any person obtaining a copy of this software
* and associated documentation files (the "Software"),
* to deal in the Software without restriction,
* including without limitation the rights to use,
* copy, modify, merge, publish, distribute, sublicense,
* and/or sell copies of the Software, and to permit
* persons to whom the Software is furnished to do so,
* subject to the following conditions:
* The above copyright notice and this permission
* notice shall be included in all copies or substantial
* portions of the Software.
* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
* OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
* LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
* FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
* EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
* FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
* IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
* FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
* THE USE OR OTHER DEALINGS IN THE SOFTWARE.
* */
#endif
51
#ifndef FMVECTORH
#define FMVECTORH

/*
• NAME: FM vector.h
• WRITTEN BY:
Patrick Moran
pmoran@nas.nasa.gov
*/

#include <iostream>
#include <utility>
#if defined( sgi) && !defined( GNUC )
#include <math.h>
#else
#include <cmath>
#endif
#include "FM types.h"

template <int N, typename T>
T FMdot (const FMvector<N, T>&, const FMvector<N, T>&) ;

template <typename T>
FM vector<3, T> FMcross (const FMvector<3, T>&,
const FM vector<3, T>&) ;

template <int N, typename T>
class FM vector
{
public :
FM vector();
FM vector(const T dat[] ) ;
template <typename T>
FM vector(const FM vector<N, T>& dat);

T& operator[] (int i) { return d[i]; }
const T& operator[] (int i) const { return d[i]; }

friend FM vector<N,T> operator*(const FMvector<N,T>& u) ;
friend FM vector<N,T> operator*(typename FM traits<T>::element_type lhs,
const FM vector<N,T>& rhs) ;
friend FM vector<N,T> operator*(const FM vector<N,T>& lhs,
typename FM traits<T>::element_type rhs) ;
friend FM vector<N,T> operator+(const FMvector<N,T>& lhs,
const FM vector<N,T>& rhs) ;
friend FM vector<N,T> operator-(const FMvector<N,T>& lhs,
const FM vector<N,T>& rhs) ;

private :
T d[N];
};

template <typename T>
class FM vector<1, T>
{
public:
FM vector();
FM vector(const T dat[] ) ;

T& operator[] (int i) { return d[i]; }
const T& operator[] (int i) const { return d[i]; }

friend FM vector<1,T> operator*(const FMvector<1,T>& u) ;
friend FM vector<1,T> operator*(typename FM traits<T>::element_type lhs,
const FM vector<1,T>& rhs) ;
friend FM vector<1,T> operator*(const FM vector<1,T>& lhs,
typename FM traits<T>::element_type rhs) ;
friend FM vector<1,T> operator+(const FMvector<1,T>& lhs,
const FM vector<1,T>& rhs) ;
friend FM vector<1,T> operator-(const FMvector<1,T>& lhs,
const FM vector<1,T>& rhs) ;

private :
T d[0];
};

// Emacs mode * c++ *
friend FM vector<3, T>;
operator*(typename FM_traits<T>::element_type lhs, const FM_vector<3, T>& rhs) {
    return FM_vector<3, T>(lhs * rhs.d[0], lhs * rhs.d[1], lhs * rhs.d[2]);
}
friend FM vector<3, T> operator*(const FM_vector<3, T>& lhs, typename FM_traits<T>::element_type rhs) {
}
friend FM vector<3, T>& operator/=(typename FM traits<T>::element_type s) {
    return *this;
}

FM vector<3, T>& operator*=(typename FM traits<T>::element_type s) {
    return *this;
}

FM vector<3, T>& operator+=(const FM_vector<3, T>& v) {
    return *this;
}

FM vector<3, T>& operator-=(const FM_vector<3, T>& v) {
    return *this;
}

FM traits<T>::element_type* v() const
{
    return reinterpret_cast<const typename FM traits<T>::element_type*>(d);
}

const FM traits<T>::element_type* v() const
{
    return reinterpret_cast<typename FM traits<T>::element_type*>(d);
}

T& operator[](int i) { return d[i]; }

const T& operator[](int i) const { return d[i]; }

FM vector(const T& a0, const FM_vector<2, S>& dat) {
    d[0] = static_cast<T>(dat[0]);
    d[1] = static_cast<T>(dat[1]);
    d[2] = static_cast<T>(dat[2]);
}

explicit FM vector(const FM_vector<3, S>& dat) {
    d[0] = dat[0];
    d[1] = dat[1];
    d[2] = dat[2];
}

FM_vector(const T dat[]) {
    T d[3];
    for (int i = 0; i < 3; ++i) {
        d[i] = dat[i];
    }
    return FM_vector<3, T>(d[0], d[1], d[2]);
}

template<typename S>
FM_vector<2, T> operator*(const FM_vector<2, T>& lhs, const FM_vector<2, S>& rhs) {
    return FM_vector<2, T>(lhs.d[0] * rhs.d[0], lhs.d[1] * rhs.d[1]);
}

template<typename T>
class FM vector<2, T> public:

FM vector():{}
FM_vector(const T dat[]) {
    d[0] = dat[0];
    d[1] = dat[1];
}

template<typename S>
explicit FM_vector(const FM_vector<2, S>& dat) {
    d[0] = static_cast<T>(dat[0]);
    d[1] = static_cast<T>(dat[1]);
}

FM_vector(const T a0, const FM_vector<2, S>& dat) {
    d[0] = a0;
    d[1] = dat[1];
}

FM_vector(const T dat[]) {
    T d[2];
    for (int i = 0; i < 2; ++i) {
        d[i] = dat[i];
    }
    return FM_vector<2, T>(d[0], d[1]);
}

T operator[](int i) { return d[i]; }

const T& operator[](int i) const { return d[i]; }

FM traits<T>::element_type a0;

typedef typename FM_traits<T>::element_type* v;

friend T FMdot<T>(const FM_vector<2,T>&_, const FM_vector<2,T>&)
{
    return FM_vector<2,T>(lhs.d[0] * rhs.d[0], lhs.d[1] * rhs.d[1]);
}

friend FM_vector<2,T> operator*(typename FM_traits<T>::element_type lhs, const FM_vector<2,T>& rhs) {
    return FM_vector<2,T>(lhs * rhs.d[0], lhs * rhs.d[1]);
}

friend FM vector<2, T> operator*(const FM_vector<2, T>& lhs, typename FM_traits<T>::element_type rhs) {
    return FM_vector<2, T>(lhs.d[0] * rhs, lhs.d[1] * rhs);
}

friend const FM_vector<2, T>& operator-(const FM_vector<2, T>& u){
    return FM_vector<2, T>(0, 0);
}

friend FM_vector<2, T>& operator/=(typename FM traits<T>::element_type s) {
    return *this;
}

FM traits<T>::element_type* v() const
{
    return reinterpret_cast<typename FM traits<T>::element_type*>(d);
}

const FM traits<T>::element_type* v() const
{
    return reinterpret_cast<const typename FM traits<T>::element_type*>(d);
}

T& operator[](int i) { return d[i]; }

const T& operator[](int i) const { return d[i]; }

friend FM_vector<2,T> operator*(const FM_vector<2, T>& lhs, typename FM_traits<T>::element_type rhs) {
    return FM_vector<2,T>(lhs.d[0] * rhs, lhs.d[1] * rhs);
}

friend FM_vector<2, T> operator*(const FM_vector<2, T>& lhs, typename FM_traits<T>::element_type rhs) {
    return FM_vector<2, T>(lhs.d[0] * rhs, lhs.d[1] * rhs);
}
public:

class FM vector<4,T>
{...

private:

T d[4];

};

template <typename T>
class FM vector<4,T>
{...

template <typename T>
FM vector<4,T>&
{...

FM vector<4,T>&
{...

friend bool operator==(const FM vector<4,T>& lhs, const FM vector<4,T>& rhs)
{...

FM vector<4,T>&
{...

friend FM vector<3,T> FMcross<T>(const FM vector<3,T>& lhs, const FM vector<3,T>& rhs)
{...

}
template <typename T>
T FMdot (const FMvector<3,T>& lhs, const FMvector<3,T>& rhs) {
    return 
        lhs.d[0] * rhs.d[0] + 
        lhs.d[1] * rhs.d[1];
}

template <typename T>
T FMdot (const FMvector<4,T>& lhs, const FMvector<4,T>& rhs) {
    return 
        lhs.d[0] * rhs.d[0] + 
}

template <typename T>
FMvector<3,T> FMcross(const FMvector<3,T>& lhs, const FMvector<3,T>& rhs) {
    return FMvector<3,T> ( 
}

template <int N, typename T>
T FMmag(const FMvector<N,T>& v) {
    return (T) sqrt(FMdot(v, v));
}

template <int N, typename T>
T FMdistance2(const FMvector<N,T>& lhs, const FMvector<N,T>& rhs) {
    FMvector<N,T> d; 
    d = rhs - lhs;
    return FMdot(d, d);
}

template <int N, typename T>
std::pair<FMvector<N,T>,FMvector<N,T> >
FMoperator rain_max (std::pair<FMvector<N,T>,FMvector<N,T> > r, 
                     const FMvector<N,T>& v) {
    for (int i = 0; i < N; i++) {
        if (v[i] < r.first[i]) 
            r.first[i] = v[i];
        else if (v[i] > r.second[i])
            r.second[i] = v[i];
    }
    return r;
}

template <int N>
FMvector<N,bool> operator! (const FMvector<N,bool>& u) {
    bool tmp[N];
    for (int i = 0; i < N; ++i) 
        tmp[i] = !u[i];
    return FMvector<N,bool>(tmp);
}

template <int N>
FMvector<N,bool> operator&& (const FMvector<N,bool>& lhs, 
                           const FMvector<N,bool>& rhs) {
    bool tmp[N];
    for (int i = 0; i < N; ++i) 
        tmp[i] = lhs[i] && rhs[i];
    return FMvector<N,bool>(tmp);
}

template <int N>
FMvector<N,bool> operator|| (const FMvector<N,bool>& lhs, 
                           const FMvector<N,bool>& rhs) {
    bool tmp[N];
    for (int i = 0; i < N; ++i) 
        tmp[i] = lhs[i] || rhs[i];
    return FMvector<N,bool>(tmp);
}

template <int N>
bool operator< (const FMvector<N,bool>& lhs, 
                const FMvector<N,bool>& rhs) {
    bool res = true;
    for (int i = 0; i < N; ++i) {
        if (lhs[i] && !rhs[i]) {
            res = false;
            break;
        }
    }
    return res;
}

typedef FMvector<2,int> FMvector2i;
typedef FMvector<2,float> FMvector2f;
typedef FMvector<2,double> FMvector2d;
typedef FMvector<3,int> FMvector3i;
typedef FMvector<3,float> FMvector3f;
typedef FMvector<3,double> FMvector3d;
typedef FMvector<4,int> FMvector4i;
typedef FMvector<4,float> FMvector4f;
typedef FMvector<4,double> FMvector4d;

/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 * * The above copyright notice and this permission
 *   notice shall be included in all copies or substantial
 *   portions of the Software.
 * * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 *   OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 *   LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 *   FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 *   EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 *   FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 *   IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 *   FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 *   THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 * */
#endif