Extending The Scene Graph With A Dataflow Visualization System

Michael Kalkusch
kalkusch@icg.tu-graz.ac.at

Dieter Schmalstieg
schmalstieg@icg.tu-graz.ac.at

Institute for Computer Graphics and Vision
Graz University of Technology
Inffeldgasse 16, A-8010 Graz, Austria

ABSTRACT

Dataflow graphs are a very successful paradigm in scientific visualization, while scene graphs are a leading approach in interactive graphics and virtual reality. Both approaches have their distinct advantages, and both build on a common set of basic techniques based on graph data structures. However, despite these similarities, no unified implementation of the two paradigms exists. This paper presents an in-depth analysis of the architectural components of dataflow visualization and scene graphs, and derives a design that integrates both these approaches. The implementation of this design builds on a common software infrastructure based on a scene graph, and extends it with virtualized dataflow, which allows the use of the scene graph structure and traversal mechanism for dynamically building and evaluating dataflow.

Categories and Subject Descriptors

I.3.5 [**Computational Geometry and Object Modeling**]: Object hierarchies,scene graph, visualization, dataflow visualization system; I.3.4 [**Graphics Utilities**]: Application packages

Keywords

Object hierarchies, Scene graph, visualization, dataflow visualization system

1. INTRODUCTION

Bethel observed in a SIGGRAPH 1999 panel [2] that *"Scene graph models and scientific visualization systems can harmoniously coexist [...]. Taking a step back, we can reference the similarity between traditional dataflow visualization systems and scene graph models."* In a nutshell, this paper is an attempt to prove that this statement is indeed true.

VRST'06 November 1–3, 2006, Limassol, Cyprus.

Dataflow visualization systems are popular tools for implementing scientific visualizations, because they provide an intuitive approach to iterative development and data analysis. The user constructs a graph composed of nodes representing visualization functions, connected by edges representing the flow of data from one node to the next. This approach, related to the *pipes–and–filters* design pattern [4] is often complemented by a beginner friendly visual programming front–end: "Programming by plumbing." [8]

Scene graphs used for Virtual Reality (VR), such as Inventor [14], OpenSG [10], OpenSceneGraph [3] or Performer [11] are also based on a graph structure, but with a different objective. Nodes in a scene graph are graphical objects contributing to the scene, such as geometry, texture, transformations or cameras. Arcs in a scene graph are used to hierarchically arrange the scene graph, typically according to geometric or semantic constraints.

Some general–purpose scene graph libraries have been extended to address the needs of scientific visualization. For example, OpenSG [10] has integrated volume rendering, and TGS Open Inventor has an extension for scientific visualization, called DataVis [15]. However, these visualization extensions encapsulate visualization data and visualization algorithms tighly in a way that precludes general functional composition of different visualizations such as possible in for example VTK [12]. A typical design shortcoming of such approaches is the combinatorical explosion of classes in [15] resulting from the need to combine every type of input data with every visualization method.

The CashFlow architecture presented in this paper enriches a VR scene graph with a dataflow architecture for visualization. It has the following noteworthy features:

1. *Natural integration of dataflow into the scene graph* Object–oriented graphics systems have a number of common architectural features with dataflow systems, such as the organization of nodes into separate data objects and process objects (*Strategy* design pattern [6]); node reference counting; reflection; or the use of scripting languages for high level programming. CashFlow builds dataflow visualizations on top of a scene graph software architecture, reusing the common infrastructure and thereby tightly integrating both paradigms.

2. *Virtualization of dataflow* Rather than hardwiring data Rather than hardwiring data and flow in the scene graph, CashFlow uses virtualized data and virtualized

flow. Virtualized data separates raw data, access patterns and structure (geometry and topology) into separate entities that are dynamically composed as needed, eliminating expensive data copy operations. Virtualized flow between nodes is established dynamically by utilizing the scene graph traversal mechanism. The advantage of virtualized flow over a conventional definition of flow is the strong decoupling of the source and destination of the flow, following an *Event Channel* pattern [4]. Virtualized flow allows easy decomposition of complex visualization problems into smaller sub–problems with localized influence on the dataflow graph.

3. *VR interaction with components of the dataflow*
Scene graph architectures for VR typically allow a high degree of interaction with the scene. CashFlow inherits the full interaction capabilities from the underlying scene graph and allows to apply all existing powerful VR interaction techniques to the new elements of the dataflow visualization system, for enhanced control of visualization and computational steering parameters.

2. BACKGROUND

2.1 Dataflow visualization

The process of scientific visualization is usually explained as a visualization pipeline, composed of a sequence of stages (see Figure 1). This concept is closely related to the rendering pipeline and was introduced by [7][17][12].

This pipeline consists of filtering of data (selection from a larger or infinite pool), mapping of the data (transforming the raw data into a format applicable for visualization), and rendering of the data (generation of images) as seen in figure 1. The output of the mapping stage generates geometric primitives, which are ultimately composited and rendered. To manage this complexity, visualization systems use an object–oriented approach to embedding visualization algorithms in components with well–defined input and output ports [17][8][12]. Using a *pipes–and–filters* design pattern [4], these components can be assembled into graph structures.

Figure 1: Traditional visualization pipeline.

2.2 Graph evaluation

Unlike the edges denoting dataflow in dataflow visualization, edges in the scene graph denote parent-child relationships. In addition to the structural hierarchy made up from these parent-child edges, many scene graph systems like Open Inventor [14], Avango[16] or Maya introduce an additional category of functional dependencies through an *Observer* pattern [6] on nodes or part of nodes (fields). This *dependency graph* exists independently of structural graph of the scene, and represents a form of dataflow.

The runtime system's main responsibility is the appropriate scheduling of the evaluation of the graph. For efficiency, most toolkits use either caching or lazy evaluation.

- *Caching* After an initial full evaluation, the results are cached. Only nodes that have been modified or depend on a modified node need to be reevaluated.
- *Lazy evaluation* Only if the results are requested, the evaluation is triggered. Thus the data flow graph is executed in reverse order while requesting the data from sink to source [12].

A node that has been modified is marked for reevaluation. Dependent nodes are recursively notified and marked for reevaluation. In case a system has both a structural graph and a dependency graph, both structurally and functionally dependent nodes need to be notified. This propagation of notification can either happen immediately after the modification [14], or can be deferred to an explicit notification phase [1][12][10]. If the notification reaches a final output, typically a root or sink node connected to a rendering view, reevaluation is scheduled.

Reevaluation, for example for rendering a new image, is then simply done with a *Visitor* design pattern [6] traversing all marked nodes starting from the sink (see figure 2). While the actual traversal strategy may be specific to each system (e.g., depth first and left–before–right [14], in–order, breadth–first, priority queue), the mechanisms of all systems are conceptually similar.

2.3 State propagation

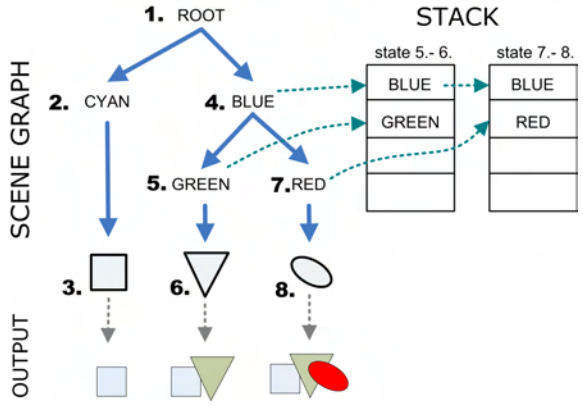

Figure 2: Scene graph traversal with stack.

The dependency graph is a secondary structure in scene graph systems. Each node may have several *fields*, which can be connected to corresponding fields of another node. It allows modelling of functional dependencies between objects in the scene graph that are disconnected in the geometric and semantic hierarchy. However, the direct connection of dependent nodes, while convenient to model simple connections, can have disadvantages when naively used for dataflow modeling involving computationally and memory intensive operations:

- *Problem of strong coupling*
Field connections in scene graphs are direct connections. This may not always be desirable given that the

nodes involved come from different parts of a scene graph, as it prevents a divide and conquer strategy from being used to model the overall scene graph. Otherwise independent sub–graphs constituting useful local solutions to a specific design problem can no longer be treated independently. Direct connections also imply that the number of senders and/or receivers is known exactly in advance, and changing connections at runtime may be undesirable since it means a constantly changing scene graph.

- *Problem of copying*
  The evaluation of functional dependencies computes and stores the result, for nodes further down the dependency chain. The strict type system of fields requires that many typical filter operations (selection from and interpretation of raw data) are done explicitly as part of the evaluation, resulting in the copying of potentially large quantities of data unless special adaptor objects are used. Many systems [8][15] allow multiple references to large data items, but this only makes the data available in the original form, rather than allowing at least some degree of filtering (such as stripping parts of the data) on the fly.

CashFlow solves these problems by embedding a dataflow graph inside a scene graph using *Elements* (see Figure 5). Scene graph traversal presents an alternative to direct connections via a dependency graph. During the traversal, general information is accumulated in the state associated with the traversal. The state contains a set of working variables (elements), for example the current drawing color or transformation matrix. Nodes modify the state as needed, and read it to obtain configuration information needed to carry out their own responsibilities. The state can be used to communicate arbitrary information in an anonymous way between nodes. Source and destination node communicate indirectly and need not be aware of each other explicitly.

Elements of the state are pushed on a stack as the traversal progresses deeper down the structural graph, and restored when the traversal is backtracking. Hence, the stack preserves intermediate state, and by default state manipulations have local scope. The stack management is done implicitly by the non–leaf nodes of the graph, conveniently decoupling the effects of state modification happening in independent sub–graphs.

Using elements addresses the problem of strong coupling of sender and receiver. The second problem − the copying of data − can also be overcome by elements passing references to data via the state rather than the data itself. These ideas will become important in our software design, which uses the elements as virtual edges in a virtualized dataflow graph.

3. SOFTWARE ARCHITECTURE

Our approach of embedding a dataflow visualization system into a scene graph is based on Coin[13], a free reimplementation of the popular Open Inventor API. One of the overall design goals was to create a very slim, yet powerful library, by maximizing the reuse of existing runtime infrastructure of the Coin framework.

First of all, nodes in the dataflow graph can be directly mapped to nodes in the scene graph. This approach is trivial, but immediately provides the following infrastructure features which are essential for a dataflow visualization system:

- Object–oriented class hierarchy enabling reflection
- Straightforward extensibility through subclassing
- Interpreted scripting language via Coin's ".iv" file format (the precursor to the VRML standard)
- Parameterization of nodes by their data members which are called "fields".

While nodes of the dataflow graph are implemented as nodes in the scene graph, edges of the dataflow graph can obviously not be implemented as edges of the structural graph. Coin also offers a dependency graph, but using it for dataflow incurs the drawbacks mentioned in section 2.3. It was therefore decided to implement virtualized dataflow on top of stateful traversal.

3.1 Virtualized dataflow

Virtualized dataflow removes the conventional tight coupling of raw data to its topological, geometrical and visual interpretation. The data itself (DataNode), the instructions on filtering the relevant subset of the data to be used (SelectionNode), the geometric/topological interpretation (mapping) of the data (GridNode), and finally the visualization techniques (ConsumerNode) are all separate entities that can be placed anywhere in the scene graph. Visualization techniques can be divided into rendering (RenderNode) and generating intermediate data (GeneratorNode).

Figure 3: Rendering pipeline in CashFlow. Corresponding UML diagram is shown in figure 5.

In isolation, only the RenderNode has visible effects. Visualizations are assembled dynamically during the scene graph traversal by dynamically associating data, selection and grid nodes with render nodes based on string keys stored in elements. Since the keys are managed as elements, binding of data, selection, grid, and rendering occurs at the latest possible moment. Once a RenderNode (see section 4.4) or a GeneratorNode (see section 4.5) are touched during scene graph traversal, the references to the data, selection and grid nodes are received by querying the appropriate elements (see figure 5).

3.2 An example for virtualized dataflow

A dataset of the flow field surrounding the Space Shuttle organized in a regular radial grid. The goal is to produce a visualization showing the body of the space shuttle and selected planar slices of the surrounding flow, both along the main axis and perpendicular to the main axis. For the sake of simplicity, color coded polylines connecting the sample points are used in rendering.

The scene graph used to generate the image from figure 4a is shown in figure 4c, together with the corresponding virtual dataflow in figure 4d, while figure 4b show the same dataset in bird's eye view. Four instances of dataflow

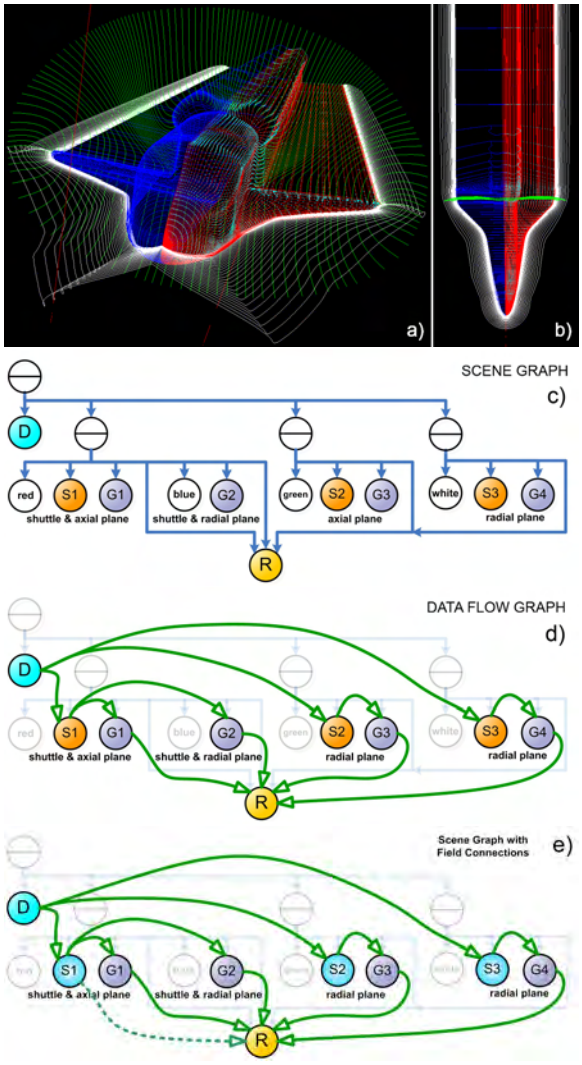

Figure 4: Space shuttle: axial shuttle grid (G1) rendered in red and radial shuttle grid (G2) rendered in blue are using the same SelectionNode (S1). The green grid (G3) and the perpendicular white grid (G4) use their own Selections (S2) and (S3). The shuttle data (D) is reused in figure 4a-d. Figure 4e shows the same Scene Graph with field connections were data need to be copied to (S1)-(S3) and to (G1)-(G4). Dashed line from (S1) to (R) indicates an optimization for field connections.

(shown as sequences of green arrows in figure 4d) from the single data node D to the single rendering node R of the scene graph along various combinations of selection and grid nodes work together to produce the result. The dataflow is established dynamically as a result of the traversal, and is not evident when the scene graph is inspected as a static entity in figure 4c. In particular, note that not only D and R are used multiple times, but also selection $S1$ is used for both the white and green slices. Figure 4e) shows the same scene graph using field connections. The SelectionNode would create a copy of the data representing the current selection. One optimization to avoid copying of data from the SelectionNode to the grid node would be a direct linking of selection and rendering node as indicated between $S1$ and R.

The visualization is established by combining the four parts in the final framebuffer through conventional OpenGL rendering. No data from D is copied explicitly in the generation of the visualization, however real–time performance of the scene graph rendering is ensured through display list caching of the rendered data, which is managed implicitly by the Coin runtime system.

3.3 Elements as virtual edges

In essence, a virtual dataflow is established by using elements as virtual edges. Since elements are organized in stacks and affected by the traversal order, the currently active list of virtual edges changes during traversal, allowing extremely flexible composition of the dataflow graph embedded inside the scene graph.

When a DataNode is traversed, the pair *<pointer to DataNode, key of DataNode>* is placed in the data element. Another data node, which uses the same key but is traversed later, can overwrite the entry in the element with a pointer to itself. Similarly, the SelectionNode places a key/pointer pair in the selection element, and the GridNode places a key/pointer pair in the grid element. Multiple assignments to the same key in a particular element's set of key/pointer pairs will override the previous assignment, but backtracking will restore the previous state when progressing to another part of the scene graph. It is also possible to query the elements for the last n–nodes added to the elements. This is very useful once ConsumerNodes require multiple data sources. Changing this order inside the element is also possible.

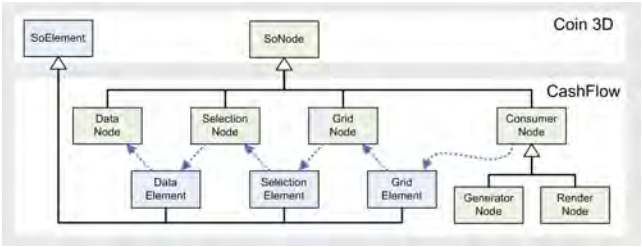

Figure 5: Class diagram showing the main components (nodes & elements) participating in the virtualized dataflow. Dashed lines show access to data.

At evaluation, the virtual edges established through elements are traversed backwards: The ConsumerNode uses its specific keys to query the elements for the intended GridNode. The GridNode refers back to the SelectionNode, and the SelectionNode refers back to the DataNode (see figure 5). Depending on the current state, i.e., the content of the elements, the same ConsumerNode will yield different output, because its input is different. The virtual dataflow is dynamically composed as an outcome of the scene graph traversal. By modifying the scene graph traversal order, for example by rearranging the scene graph or using switch nodes for selective traversal, the virtual dataflow can be interactively altered.

The intended consequence of this design is that DataNode, SelectionNode and GridNode can be arbitrarily combined to produce visualizations. Specifically, a pool of data composed of one or more DataNodes can be interpreted in different ways through multiple combinations of SelectionNodes, GridNodes and ConsumerNodes in the same traversal. SelectionNode and GridNode together provide the ConsumerNodes with appropriate iterators to access the data stored in the DataNode.

4. IMPLEMENTATION OF CASHFLOW

CashFlow extends Coin and is implemented as a set of nodes, actions and elements. Figure 5 shows the fundamental nodes and elements used.

4.1 DataNode

Objective: Store raw data under a given key.

The DataNode's purpose is simply to assign a *dataKey* (stored in the data element) to a block of raw data. This defines the first stage of the visualization pipeline. The user can specify the actual data either directly inline in the .iv file defining the data node, or by using a separate loader node capable of reading specific third–party formats from a file. Data is stored in the form of arrays (multi–valued field types, starting with SoMF...) of any of Coin's basic data types. Rather than implementing individual subclasses for each primitive data type, the DataNode offers all primitive types simultaneously. This allows the use of multiple fields, which is an arrays of fields, for heterogeneous data sources. It often removes the need to use two data nodes, because different types of data can be stored inside one DataNode. The SelectionNode, that will be introduced in the next section, will also allow storing several blocks of data inside one multiple field of a DataNode.

4.2 SelectionNode

Objective: Define access pattern(s) to raw data.

The purpose of the SelectionNode is to define an access pattern to the raw data. Consequently, the only interface of the SelectionNode relevant to the client is an access function that maps an index to a data value. This is a form of filtering in the sense of [17][1], but the SelectionNode performs this filtering on the fly rather than it being precomputed, and is therefore limited to shuffling and skipping data. The SelectionNode is motivated by the observation that many filtering operations do not actually synthesize new data (this is reserved for the GeneratorNode explained in section 4.5), but only require the data in a different order and composition from its native form due to strict typing requirements.

Copying such data more than strictly necessary just to provide type compatibility with RenderNodes is wasteful. Copying the data for optimal rendering efficiency is useful, but this is internally handled by the Coin graphics runtime system which automatically builds display lists to accelerate the performance of RenderNodes.

- In the simplest single–block mode the SelectionNode returns a section of the data. The data is taken from the DataNode designated by *dataKey*. The value of *dataKey* is compared to all keys in the data element, and the matching DataNode is used as the source of the data. The type field determines the type of data. The returned data array starts at the given *offset* and has the given *length*. The result is made available to the next stage in the visualization pipeline by placing a *selectionKey* associated with the SelectionNode in the selection element (see figure 5).

- The multi–block mode allows certain data values to be skipped. Again, the relevant data is identified by *dataKey* and type. The returned data starts at *offset* and divides the array of given *length* into blocks of *increment* data items. The first *repeat* data items of every block are returned (see figure 6).

- The random–block mode is intended to implement random access to the DataNode identified by *dataKey* and type. The returned data is retrieved according to the index specified in a lookup table.

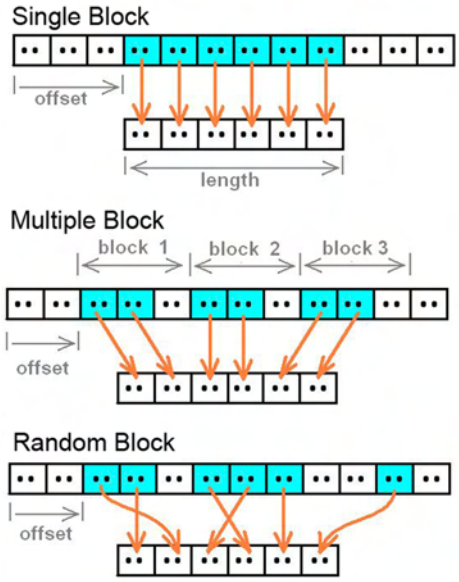

Figure 6: Virtual arrays: Single, multiple and random block virtual arrays.

All basic configuration fields in the SelectionNode are arrays and can store multiple values. This mechanism allows multiple independent selections of the data to be configured in a single SelectionNode. Every selection corresponds to a specific index into the arrays *selectionKey*, *dataKey*, *offset*, *length*. Two restrictions apply: obviously, all the mentioned configuration fields in one SelectionNode must have the same number of items; furthermore storing multiple selections is only possible in single–block and multi–block mode.

4.3 GridNode

Objective: Define topological and geometrical interpretations of raw data.

The GridNode's purpose is to add topological and geometrical interpretation to raw data. For example, a set of raw geometric coordinates can be interpreted as a 2D or 3D mesh, or a set of measurements can be interpreted as density values at the vertices of a tetrahedral mesh. The data

[1]AVS Express http://www.avs.com

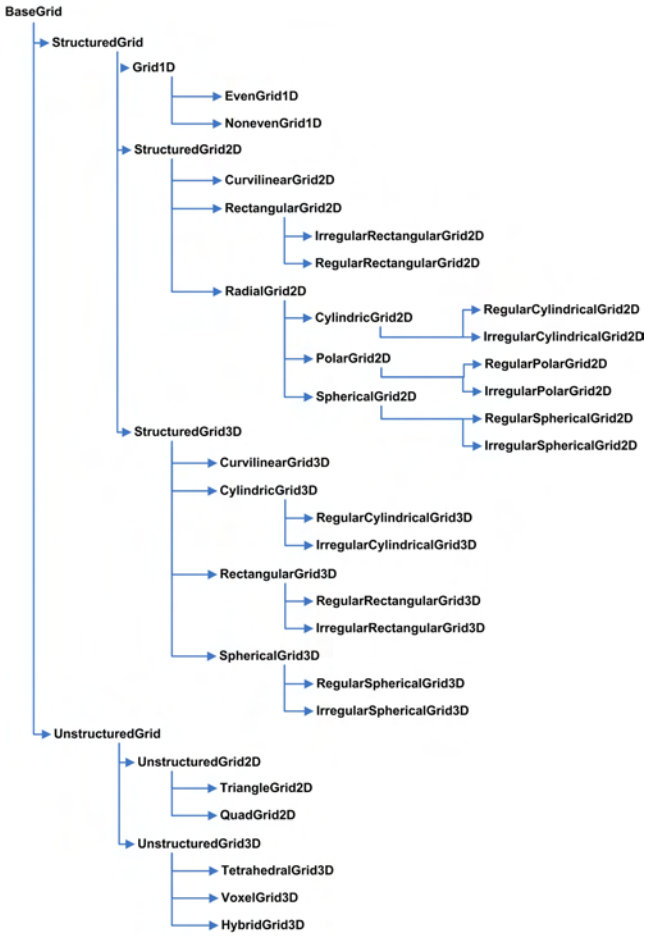

Figure 7: BaseGrid class hierarchy of CashFlow.

is read via a SelectionNode indicated by the selection element corresponding to *selectionKey*, and is made available by writing the *gridKey* to the grid element.

The main interface of the GridNode is the grid iterator. It allows a RenderNode to systematically iterate through all items of a chosen geometric entity (either point, edge, face or cell) defined by the grid. The grid iterators relieves the RenderNode from the need to know the actual topological or geometrical structure as long as drawing requires information about only a local geometric entity. Consequently, the same RenderNode can work with a large variety of grids.

CashFlow supports a large variety of grids organized as a class hierarchy loosely inspired by the Field Model library [5] [9]. The abstract base class defining the iterators interface is called BaseGrid. The main characteristics of the grids used for organizing the class hierarchy are:

- Structure: structured grids fit into one array and information on adjacency of topological entities is implicit. In contrast, unstructured grid are usually not repetitive, and require explicit adjacency information for topological entities.

- Dimension: grids can define line sets (1D), surface meshes (2D) or volumetric meshes (3D).

- Regularity: a structured grid can be subdivided in a regular way into evenly sized entities by just specifying the number of subdivisions applied to each dimension. Alternatively, a grid can be subdivided in an irregular way into non–evenly sized entities by using an explicitly given data set of sizes to be used.

- Shape: there are a number of basic shapes available for each structure, dimension and regularity. These shapes include perpendicular shapes (rectangle, cuboid), radial shapes (disc, cylinder, sphere) and various irregular primitives such as triangle or tetrahedron.

4.4 RenderNode

Objective: Create OpenGL based drawings.

The class hierarchy of render nodes implement rendering algorithms. A RenderNode requests data from a GridNode by accessing the grid element given by the *gridKey*. There are a large number of possible RenderNodes depending on the application domain, and only a basic set, mostly from the domain of flow visualization, have been implemented to showcase the capabilities of CashFlow. Examples for RenderNodes are shown in figures 4,8–10.

- Point Cloud Renderer: The most basic Render node draws a color–coded point at each vertex position. This visualization can be useful for example to investigate the density of a grid.

- Glyph Renderer : This is the complement of the point renderer in vector fields. At each selected point a glyph shows the direction of the flow. Glyphs can be also used to visualize multidimensional data.

- Cell Renderer: The grid is divided into cells and each cell is rendered.

- Wireframe Renderer: The grid or parts of the grid are visualized using a wire–frame.

- Surface Renderer: This renderer is similar to the wireframe renderer. The grid is rendered using surfaces.

- Streamline Renderer: Rendering of streamlines in different ways. In extension to the default streamline renderer an up–vector per each interpolation point can be taken into account to visualize vorticity.

- Polygonal Renderer: renders polygonal data.

4.5 GeneratorNode

Objective: Synthesize output data from input data.

The CashFlow approach up to now can be described as a fixed function visualization pipeline. While its components have a high amount of flexibility, the basic stages of the pipeline are fixed as show in figure 3. None of these stages truly synthesizes new data from data already existing in the scene graph. For example, a marching cubes algorithm creates a polygonal iso–surface from a set of voxels. To address the need for such data synthesis, we introduce an additional node, the GeneratorNode. Like the RenderNode, it is derived from ConsumerNode, because it consumes information from a DataNode. The mapper node differs from the previously introduced nodes in one important aspect: it actively modifies another node, namely the destination DataNode.

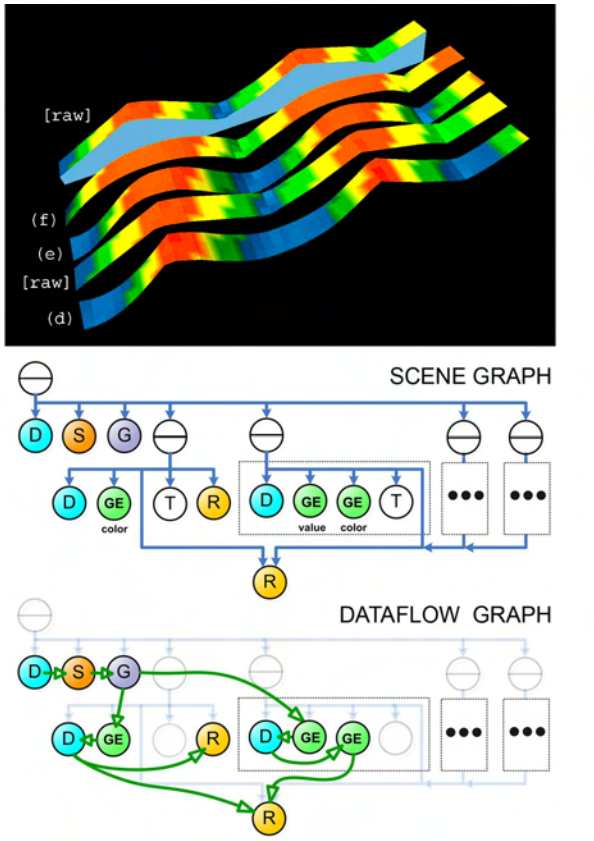

Figure 8: Parametric mapping of scalar values. Linear input data [raw] shown as height–field with color–coding. Input data is altered in a *GeneratorNode* using cubic splines (d)-(f) and a second *GeneratorNode* creates the corresponding color–coding. The scene graph shows a *DataNode* (D), *SelectionNode* (S) and a *GridNode* (G) storing raw data, which is rendered (R) using a colormap created by a *GeneratorNode* (GE) and shifted by a *TransformerNode* (T). Inside the dashed box data is altered (GE), buffered (D) and a colormap (GE) is created and rendered (R).

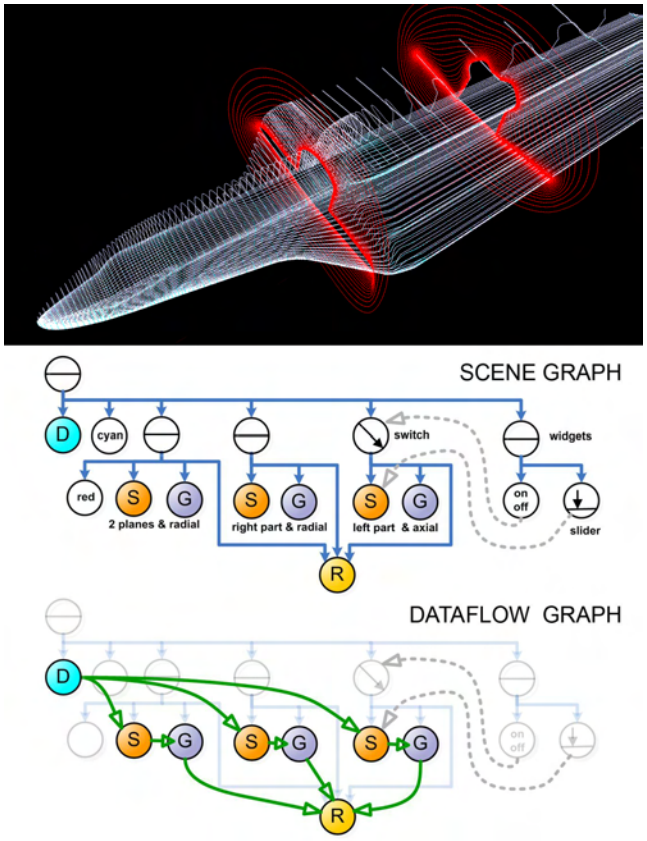

Figure 9: Left part of space shuttle grid rendered in axial direction, right part of grid rendered in radial direction. Two red radial layers in red show uneven distibution of the grid. All 3 rendering styles access the same data in memory. The cyan radial grid may be disabled via the SwitchNode or shifted by the SliderNode.

One caveat with using GeneratorNodes is that they can be computationally intensive. In such a case the execution of the computation during the traversal is not appropriate, since the traversal may stall. The usual solution for this kind of problem is decoupling the computation in a separate thread and deferring the update of the rendering data until the result of the computation becomes available. In Coin, the spawning of a separate thread can easily be encapsulated in the GeneratorNode itself, while the Coin runtime system automatically takes care of caching the render data in OpenGL display lists. One limitation of the current implementation is that there is no automatic scheduling of such computation threads, but the developer of each type of GeneratorNode needs to schedule a separate computation thread explicitly in the implementation of the GeneratorNode.

5. RESULTS

Figure 8 shows a combination of two GeneratorNodes, a CubicDataGeneratorNode (labeled GE value) and a ColorGeneratorNode (labeled GE color). The CubicDataGeneratorNode interpolates the raw data using cubic splines and stores the intermediate data in a DataNode, which is used to create a color map based on that intermediate data. Since a GeneratorNode should not insert new nodes inside the scene graph, because it would limit the scripting capabilities, a DataNode for intermediate data is placed in the scene graph as shown in figure 8 inside the dashed box. Using 3 different cubic functions creates the height fields in figure 8 (d) –(f).

In figure 9 two widgets alter the scene graph and its dataflow graph interactively. An on/off node, which is coupled to a switch, allows the red slicing plane to be alternately shown and hidden. The position of the slicing plane can be modified by a slider node. Both on/off and slider node use the standard dependency graph to update the switch and selection node, respectively. More examples are shown in figure 11–10 including shaded and textured surfaces (figure 11), rendering three objects as point–clouds and gener-

ating a colormap mapped to pressure (see figure 12). The third example in figure 10 shows an iso–surface created from a CT dataset rendered as an indexed–faceset.

6. CONCLUSIONS AND FUTURE WORK

We have presented CashFlow, an architecture that integrates concepts from scene graphs and dataflow visualization systems. CashFlow is based on the scene graph library Coin, and systematically extends it towards dataflow visualization. Its design allows the advantages of both approaches to be simultaneously exploited. All attributes used to parametrize the data flow and the visualization pipeline are part of the scene graph as nodes and fields. Thus these attributes can now be manipulated interactive inside the scene graph, influencing the data flow graph.

A key achievement is the virtualization of dataflow which is enabled by introducing elements which are dynamically managed by the scene graph traversal as virtual edges, and by decomposing visualization objects into separate objects for data, topology, filtering and mapping. CashFlow is integrated into the Studierstube framework and can be used with an ART tracking system in combination with a back–projection screen (see figure 13). Several visualization parameters as well as selection properties can be mapped to tracked objects allowing interactive exploration of the datasets.

Besides the obvious extension of the feature set, such as new rendering classes, future work will focus on enhanced scheduling of the flow, in particular multi–threaded and parallel scheduling of the traversal that is the foundation of CashFlow. In addition, a visual programming front end is on the list of desired extensions. Such a visual programming tool could extend *CoinDesigner*[2], a prototype scene graph editing tool for Coin.

Acknowledgments. This research was sponsored in part by the Austrian Science Fund *FWF* under contract no. Y193. Many thanks to Gerhard Reitmayr for useful discussions. The space shuttle dataset is courtesy of *NASA NAS Systems Division Office*.

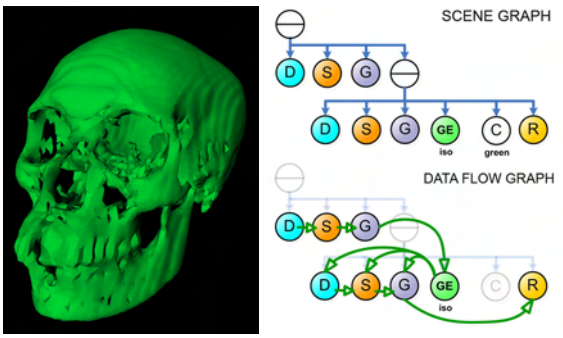

Figure 10: Iso–surface created from a CT dataset by Marching Cubes (left) with its scene graph and data flow graph (right).

[2]http://coindesigner.sourceforge.net

7. REFERENCES

[1] G. Abram and L. Treinish. An extended data-flow architecture for data analysis and visualization. In *proc. IEEE Visualization '95*, pages 263–270, 1995.

[2] W. e. Bethel. Scene graph APIs: Wired or tired? In *proc. ACM SigGraph'99 Conference abstracts and applications*, pages 136–138, 1999.

[3] D. Burns and R. Osfield. Open scene graph, 2002. http://www.openscenegraph.org/ visited at 01.02.2006.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Vol.1*. Wiley, 1996.

[5] J. M. Favre and J. Hahn. An object oriented design for the visualization of multi–variable data objects. In *proc. IEEE Visualization'94*, pages 318–325, 1994.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[7] D. Ingalls, S. Wallace, Y.-Y. Chow, F. Ludolph, and K. Doyle. Fabrik: a visual programming environment. In *proc. ACM OOPSLA '88*, pages 176–190, 1988.

[8] B. Lucas, G. Abram, D. E. N. Collins, D. Gresh, and K. McAuliffe. An architecture for a scientific visualization system. In *proc. IEEE Visualization '92.*, pages 107–114, 1992.

[9] P. Moran. Field model: An object–oriented data model for fields. Technical report, NASA, 2001. open source C++ Template library.

[10] D. Reiners. A flexible and extensible traversal framework for scenegraph systems. In *proc. 1st OpenSG Symposium*, 2002.

[11] J. Rohlf and J. Helman. Iris performer: A high performance multiprocessing toolkit for real-time 3d graphics. In *Computer Graphics (SigGraph'94 proc.)*, pages 381–394, July 1994.

[12] W. Schroeder, K. Martin, and W. Lorensen. The design and implementation of an object–oriented toolkit for 3d graphics and visualization. In *proc. IEEE Visualization'96*, pages 93–100, 1996.

[13] Coin. Systems In Motion, released 2000, http://www.coin3d.org/lib/about_coin3d/ vitited at 01.02.2006.

[14] P. Strauss and R. Carey. An object-oriented 3d graphics toolkit. In *proc. ACM SigGraph'92*, pages 341–349, 1992.

[15] TGS© Open Inventor, MeshViz (formerly DataViz and 3DDataMaster). http://www.tgs.com/support/datasheet/MCS_oiv_MeshViz.pdf visited at 01.02.2006.

[16] H. Tramberend. Avango: A distributed virtual reality framework. In *proc. IEEE Virtual Reality Conference '99*, pages 14–21, 1999.

[17] C. Upson, T. J. Faulhaber, D. Kamins, D. H. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam. The application visualization system: A computational environment for scientific visualization. In *IEEE Computer Graphics and Applications*, volume 9 of *4*, pages 30–42, July 1989.

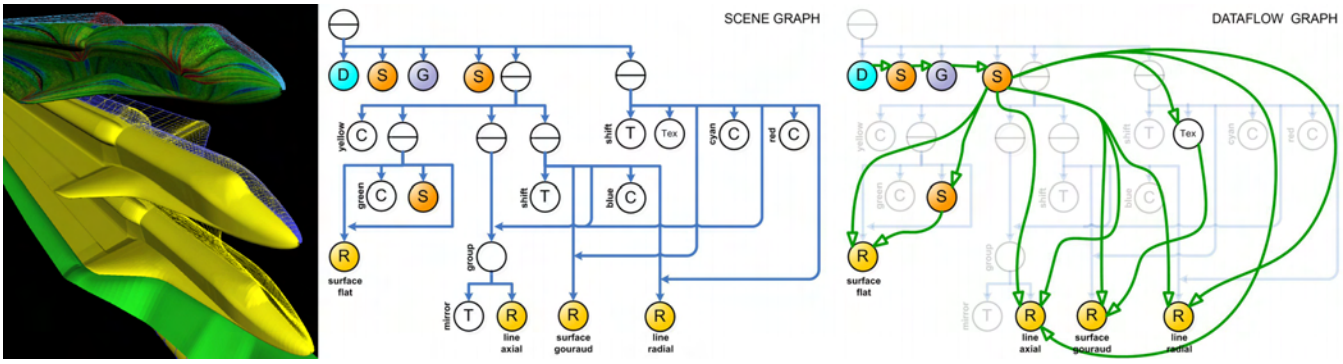

Figure 11: Space shuttle dataset with textured and shaded surfaces (left). Corresponding scene graph and data flow graph (right).

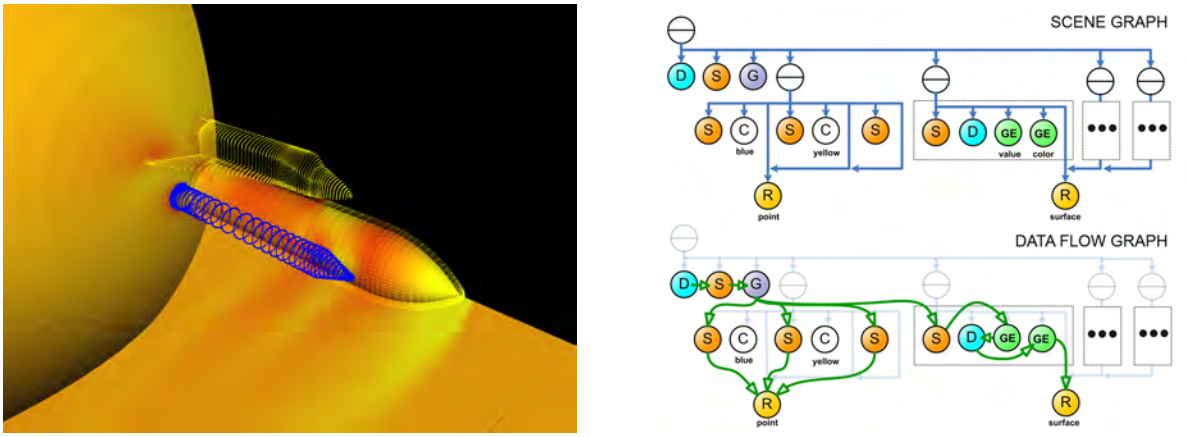

Figure 12: Space shuttle dataset rendered as point–cloud and a colormap mapped to pressure (left). Corresponding scene graph and data flow graph (right).

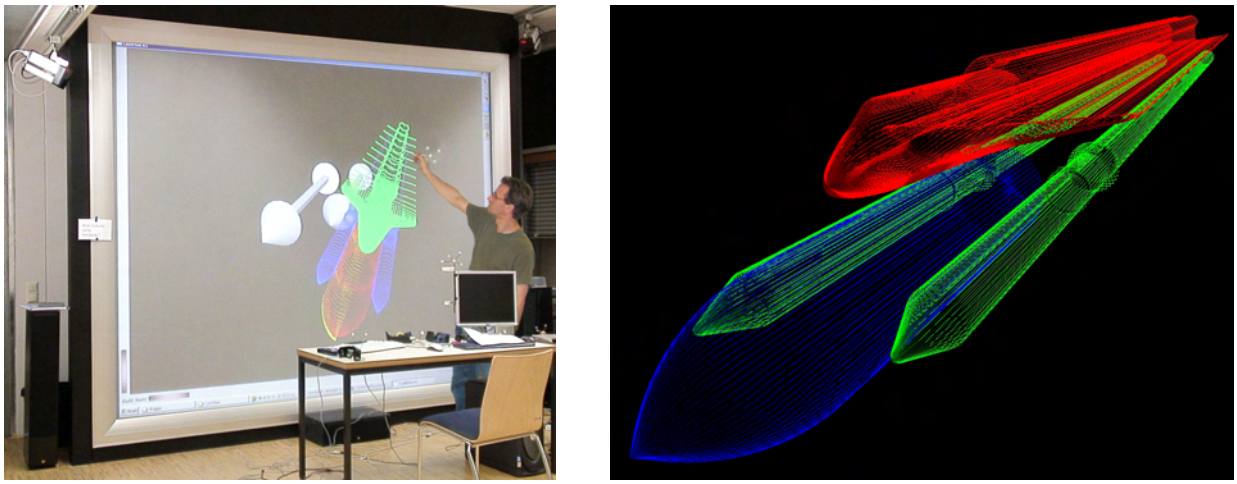

Figure 13: Point cloud visualization on large back–projection screen (left). Scene rendered using illuminated streamlines (right).