Geometric, Chronological, and Behavioral Modeling

Juergen Doellner and Klaus Hinrichs
FB 15, Informatik, Westfälische Wilhelms-Universität
Einsteinstr. 62, D - 48149 Münster, Germany
Phone & FAX: (++49) 251 / 83 - 3755; email: {dollner, khh}@math.uni-muenster.de

Abstract

Modeling and animating three-dimensional scenes involves inherent difficulties both in the specification of scenes and in the implementation of computer animation systems. We present an object-oriented methodology for the integrated modeling of geometry, time, and behavior. Models are defined by two directed acyclic graphs and a set of constraints: the *geometry graph* is used to specify hierarchically composed objects and their attributes, the *behavior graph* specifies time-dependent behaviors, and the *set of constraints* is applied to both the geometry graph and the behavior graph. Time, behavior, and constraints are represented as objects, i.e. similar to geometric primitives they are implemented as polymorphic basic building blocks. These blocks are lightweight, share a common communication protocol and can be composed in almost arbitrary manner leading to simple and elegant construction techniques as well as to efficient implementations. We have implemented our methodology in MAM, the Modeling and Animation Machine. MAM is an extensible and portable C++ toolkit which offers a rich set of modeling, animation and interaction classes. It separates rendering and modeling and therefore allows easy integration of new modeling, rendering, and interaction techniques.

Keywords: object-oriented animation, object-oriented 3D graphics, graphics software architecture, scene representation.

1 Introduction

Object orientation is a natural concept for computer graphics. It helps to manage the inherent complexity of three-dimensional modeling. First steps were made by Wisskirchen, who implemented in GEO++ [11] the functionality of PHIGS [7] and GKS in an object-oriented fashion using Smalltalk. For two-dimensional applications the object-oriented InterViews system ([3],[6]) proved to be successful. From the user's perspective, object orientation is a natural way to express the modeling of objects and tasks [4]. Computer animation adds additional complexity to graphics systems: Geometric modeling has to be combined with time-dependent behavior and constraints.

We propose a methodology with the following key goals:

- time, behavior, and constraints are first class notions, i.e. they are modeled as objects at the same level of abstraction like geometric primitives;
- integration of geometric and chronological modeling;
- lightweight and cooperative polymorphic basic building blocks; and
- independence from the underlying rendering package.

Models are separated in three parts: the *geometry graph*, the *behavior graph* and a *set of constraints*. A geometry graph is composed of basic building blocks which describe geometry and appearance. It defines shapes and hierarchically nested coordinate frames. A behavior graph is composed of basic building blocks which specify time-dependent actions, interactive behavior, and constraints. The geometry graph and the behavior graph are interrelated: The behavior graph specifies how nodes of the geometry graph are transformed in time, or how they respond to interactive manipulation. Actions are grouped in time by chronological container nodes, for instance in complex animations we must be able to group actions hierarchically in time. Constraints can be added to all basic building blocks. We

distinguish between constraints which can be resolved directly, and constraints which are resolved by an underlying constraint solver.

Large and complex models can be built by composing large numbers of lightweight and cooperative basic building blocks. Independence from the underlying rendering technique is achieved by separating rendering and modeling in two different system layers. The rendering layer is based on abstract graphical data types which are evaluated by virtual rendering devices. We have encapsulated different rendering packages, e.g. OpenGL [12], PEX [13], XGL[14], and Radiance[18], in this homogenous interface in such a way that their full functionality is accessible.

2 Related Work

Clockworks

One of the first object-oriented computer animation systems was Clockworks [1]. A Clockworks model is formed by a collection of objects which communicate through messages. Directing objects send messages to other objects in the scene at the times designated by a script. Clockworks scripts are collections of messages. Clockworks uses the object-oriented paradigm but does not apply it to the overall system structure. Building models by unordered collections of objects becomes increasingly difficult as the number of objects grows. Furthermore, there is no notion for chronological modeling, nor does Clockworks integrate constraint management.

GRAMS

The object-oriented Graphical Application Modeling Support System [4] provides a higher level of abstraction for 3D graphics as compared to low-level rendering packages. However, GRAMS does not provide constraint management nor chronological modeling. There is no explicit concept of time, therefore modeling of time-dependent and interactive behavior cannot be integrated in an object-oriented fashion.

GROOP

GROOP [5] constructs animated 3D graphic objects based on metaphors derived from movies: stage, actors, cameras. It consists of a scene construction and animation component, and a rendering component based on OpenGL, it does not support other rendering packages. GROOP does not integrate constraint management. It builds models by collections of geometrically nested objects; attributes are represented as separate objects. Also time is not a first class notion. Chronological and interactive modeling is not supported. The modeling of time-dependencies is based on “discrete” and “continuous” objects which are controlled at each time step.

OpenInventor

OpenInventor ([10],[19]), a toolkit for interactive 3D graphics, organizes scenes in directed acyclic graphs. There are three types of nodes: shapes, attributes, groups. Inventor graphs allow only certain node types to be inserted, it is therefore difficult to build new higher level objects using object-oriented inheritance [5]. Furthermore, objects depend on the order they were inserted into a node. Time-dependent behavior is implemented through “sensors” based on the callback mechanism. Time is not a first-class notion, nor exists an explicit notion for chronological modeling. Geometry and interaction are represented together in a single graph. OpenInventor provides only a simple constraint mechanism through attribute connections and engines. It is based on OpenGL and does not support other rendering packages.

TBAG

TBAG [15] is based on graphical abstract data types and explicit functions of time. Parameterized geometric models are represented by mathematical formulas. A single class (“constrainable”) represents animation parameters of all types, user interaction and animations. TBAG provides a general

approach to interaction by directly encoding interactions into constrainables. Models are composed by values of graphical abstract data types for each time step. This is computationally expensive. TBAG does not offer chronological modeling. For each frame, all objects used must be generated due to the functional approach. TBAG does not support different rendering techniques. TBAG's functional approach to describe models is not suitable for real world scenes which in general can be described easier in a declarative and hierarchical rather than a functional manner. Furthermore, the same simplicity in constructing models can be achieved by an operational notation defined for building blocks.

Others

Mirage [16] is a high-level 3D object-oriented graphics system that supports a hierarchical temporal coordinate system, but it does not treat time-varying values as a first-class notion. UGA [17] appears to be the first 3D programming framework that supports direct expression of time-varying values as functions of time and input. UGA focusses on language mechanisms such as delegation hierarchies. More application specific animation systems are Pinocchio [22] and SOLAR [21]. Pinocchio is an animation system that controls human motion and provides sequencing facilities. The SOLAR language provides abstraction levels that enable an animation sequence to be defined in steps.

In all these systems behavior is not separated from geometry. Also, there is no explicit concept for chronological modeling, e.g. simultaneous or sequential actions, which is essential for building complex animation sequences. None of the systems treat constraints and time-dependencies as first class notions.

Furthermore, most toolkits rely on one specific low-level rendering package and do not support other rendering techniques. The systems differ in the degree of object-orientation; no system provides a fine-grained object-orientation, e.g. time, colors, points, constraints as objects. Without this, objects are not lightweight enough to be shared, subclassed, and used in large numbers [2].

3 Constructing Models

3D models are sets of interrelated objects which describe geometry, behavior and constraints. The geometry is represented by a directed acyclic graph of geometric transformations, shape objects, and viewing objects called *geometry graph*. The behavior is represented by a directed acyclic graph of time-structuring objects, constraint objects, and animation objects called *behavior graph*. All objects are constrainable and maintain an ordered list of constraints. Independent constraints are added directly to the constrained object by inserting them in its constraint list. Constraints which belong to a constraint net are part of the behavior graph and are maintained by an underlying constraint solver. The geometry graph, the behavior graph and the constraints are orthogonal to each other, i.e. they complement each other and describe a complex animated model completely.

Polymorphic building blocks are implemented by *glyphs*. Glyphs are objects which share a common communication protocol and therefore can be composed almost arbitrarily. Derived glyph types redefine existing protocols, delegate protocol requests, or add new protocols. Glyphs store information redundancy-free, i.e. the information is not provided by other glyphs. Glyphs do not store context information such as the current rendering device or the current modeling and transformation matrix. This information is passed to glyphs within the glyph protocol. The base glyph class defines no storage. Shape glyphs, for example, store suitable rendering primitives, whereas polyglyphs store a list of its children. Their lightweight has the following consequences:

- Glyphs can be shared since context information is passed to them through the protocol.
- Glyphs can be combined to a high degree because they represent only one type of modeling information, and can be used in large numbers due to the minimal overhead.

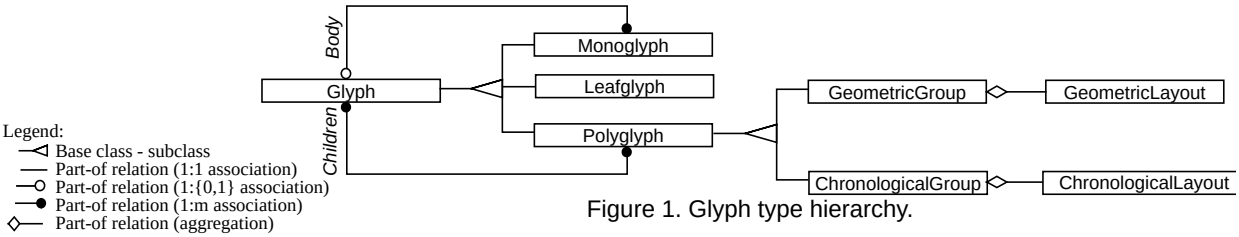

Figure 1. Glyph type hierarchy.

- The glyph class hierarchy is fine-grained and can be used efficiently for deriving new types.

We distinguish three basic types of glyphs: leafglyphs, monoglyphs and polyglyphs (s. Fig. 1)[1].

Leafglyphs have no children, their implementation of the glyph protocol is empty. The abstract base class for shapes, for example, is derived from the leafglyph class.

Monoglyphs have at most one child called *bodyglyph*. They are transparent glyphs, i. e. by default all operations are delegated to the bodyglyph. Monoglyphs are used to redefine communication protocols partially.

Polyglyphs have an arbitrary number of ordered children. The most important polyglyphs are geometric (resp. chronological) containers which determine the position of their children according to geometric (resp. chronological) layouts.

3.1 Geometry Graph

The geometry graph is composed of geometric transformations, shape glyphs and viewing glyphs. Transformation glyphs create hierarchically nested coordinate frames. Geometric groups organize their children according to geometric layouts which calculate spatial positions and geometric volumes for a group of glyphs based on their space requirements. Associating layouts with geometric groups instead of subclassing the geometric group class offers the advantage that layouts can be shared and can depend on each other. Examples:

- Unconstrained space: Position and size of children are not changed and not restricted.
- Grid: Children are aligned to three-dimensional grid points. The grid may be finite or infinite.
- Cells: Children occupy cells of a three-dimensional cell array.

Geometric objects are generalized to *shapes*. Based on their dimensionality they are subdivided into point-based, line-based and facet-based shapes. Shapes which possess physical properties such as volume, center of mass etc., are generalized to analytic shapes. Shapes typically represent their form in terms of abstract graphical data types provided by the rendering layer. For instance, physically based superquadrics represent their form as quadrilateral mesh or directly as formula according to the capabilities of the rendering device. Building shapes by aggregating abstract rendering data types leads to elegant implementations because we can delegate management operations to these types in many cases, and can pass them without further conversion to the rendering device.

In Fig. 2 the geometry graph of a virtual hall is illustrated. The scene consists of one geometric group. Columns and plinths build their own geometric group, prefixed by a color glyph. The logo on the table is built by a turtle path glyph. This glyph is a specialized shape glyph which interprets 3D turtle commands and sweeps a disk along its path. In our example, we build the string “MAM”. The turtle-path glyph is prefixed by a rotation, a scaling, and a color glyph. The animation described in the next section refers to these glyphs.

1. Class relations and object relations are given in the Object Modeling Technique (OMT) notation [23].

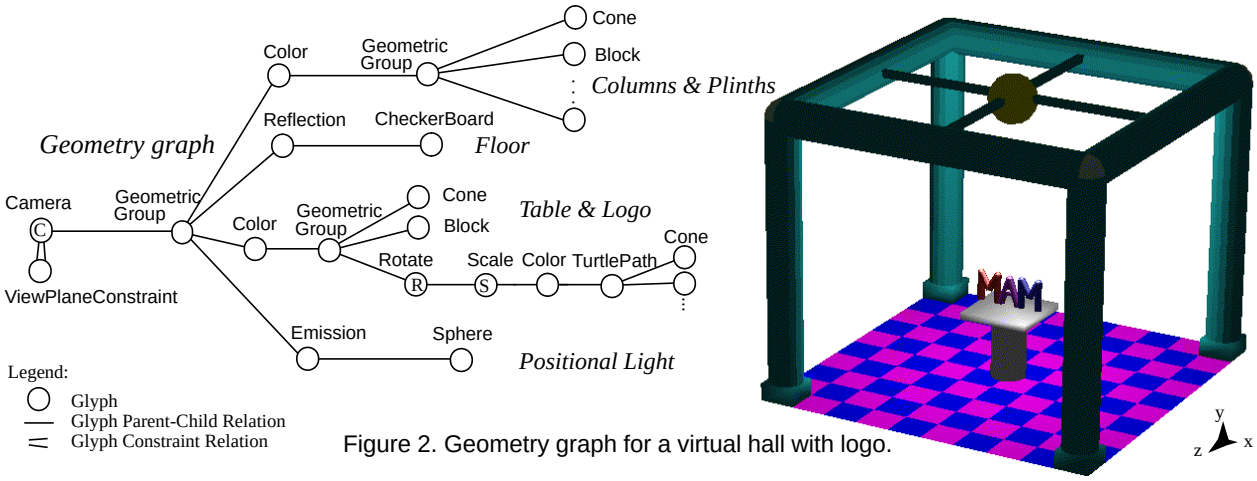

Figure 2. Geometry graph for a virtual hall with logo.

The appearance of shapes is modified by *attribute glyphs*. Attributes are monoglyphs which modify the appearance of its bodyglyph provided that there is no attribute with higher priority. If we want to fix a value of an attribute for a given subgraph temporarily we prefix the subgraph with an attribute glyph having higher priority, e.g. a wire-frame drawing style can be imposed on the whole model during user interaction. Prioritized attributes allow to overwrite attributes in subgraphs without having to modify these subgraphs.

Examples for attributes are surface color, reflection properties, emission and transmission properties, fog, and drawing styles. The attribute hierarchy is open because new attribute classes can be derived from existing attribute classes in order to integrate functionality of the specific rendering device types; e.g. there is a specialized XGL edge style glyph which defines edge antialiasing for XGL renderers. A static attribute hierarchy would not permit the integration of specific rendering capabilities.

Viewing objects include camera glyphs, camera view specifications, lightsources and prebuilt virtual environments (e.g. rooms, grounds and terrains). Camera glyphs are connected to rendering devices and typically located at the root of the geometry graph. Instead of specializing several camera types, we opt for delegating the view specification to projection objects and to provide several projection constraints, e.g. the view plane constraint which sets and restricts the view plane normal, or the view window constraint which sets and restricts the field of view. For instance, in Fig. 2 a view plane constraint glyph is added to the camera glyph.

Lightsources are modeled by shapes with light emission. The illumination result depends on the shape form. For instance, if a cone emitting light is oriented, the light spot follows. In particular, the lighting model of MAM can be mapped to radiosity-based renderers. Good mappings exist also for immediate-mode renderers such as PEX: positional lightsources correspond to spheres with light emission, spotlights correspond to cones with light emission. The sphere at the ceiling in Fig. 2 embeds a positional lightsource.

3.2 Behavior Graph

The behavior graph defines the behavior of the model. Behavior graphs specify story books, interactive behavior, and relations with glyphs of the geometry graph. *Actions* are glyphs which control behavior.

Chronological Modeling

An important problem in the production of animation sequences is the chronological modeling. A large amount of information is necessary to control and specify complex animations. Grouping actions in time reduces its complexity [20]. Chronological modeling is supported by *chronological*

groups with chronological layout. Basic layout types are:

- Sequence: Glyphs become non-intersecting sequential spans of life.
- Simultaneity: Glyphs become simultaneous spans of life.
- Fade-in and fade-out: Fade-in layouts assign spans of life which start in cascading order and end at the same point in time. Fade-out layouts are reversed fade-in layouts.

Layouts calculate the span of life for children of chronological groups. A span of life is defined by a starting point and an end point in time. A *moment* is a point in time within a span of life. The time is measured in virtual milliseconds. Instead of sending a global system time to glyphs, we send moments since moments include additional information, e.g. actions know how long they will last.

Fig. 3 contains the behavior graph of a simple logo animation for the example in Fig. 2. The script is defined as follows: 1. Scale the logo from 0 to 0.2 along the y-axis, i.e. make it visible. 2. Rotate it around the center of the table twice; at the same time, scale it along the y-axis from .2 to .5, and then reverse this scaling. 3. Scale the logo down to make it invisible. Note that step 2 lasts 20 seconds; the inner sequence does not define any time requirements; they are calculated from the simultaneity group and assigned in equal parts to the subactions. The total animation time is 5+20+5=30 seconds. If we would place a duration of 300 seconds at top of the behavior graph, the time requirements of the children would be scaled proportionally.

Glyphs specify time requirements consisting of desired, minimal, and maximal durations. Time requirements are specified by time-placement monoglyphs which redefine the corresponding protocols. Time-placements include stretchable and shrinkable durations, fixed durations, natural durations, and time requirement copies from other glyphs. Not all glyphs of the behavior graph set their time requirements explicitly; frequently they are determined automatically by time layouts. Time layouts may stretch the duration if more time is available and the glyph's time requirement is stretchable. Analogously, they shorten the duration if less time is available and the glyph's time requirement is shrinkable.

Time reversal monoglyphs invert the direction of the time progress. They are useful to model retrograde motions. Modeling of repeating actions is done with *time repeat* monoglyphs which map a span of time modulo another span of time, e.g. to model a never-ending rotation at a given speed, say 10 degrees per second, we compose a time repeat glyph of 10 seconds duration and a rotation glyph. We add a mapping to the rotation glyph. For a given moment, a mapping object computes and returns a numerical value. The linear mapping, for example, interpolates two given values during the span of life provided by the moment.

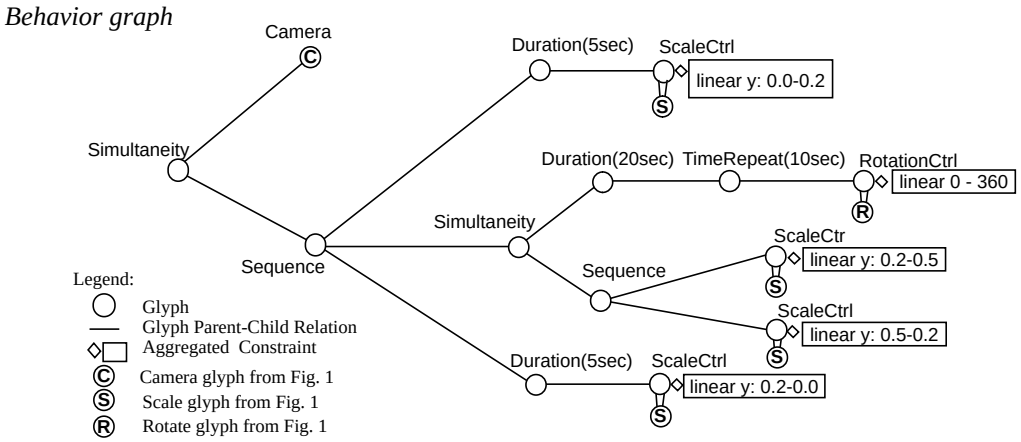

Figure 3. Behavior graph for animating the logo in Fig. 2.

Interactive Behavior

The behavior graph defines also the interactive capabilities of models. *Interaction controllers* are monoglyphs which consume and interpret events. For instance, we rotate a wheel interactively by connecting the wheel with a select-and-drag interaction controller. Events are objects which describe an incident of interest. Typically events are generated by the underlying user interface or the application (synthetic events). The behavior graph root evaluates and delegates events to its children.

3.3 Constraints

Constraints describe and maintain a set of relations on a finite set of objects. All glyphs are constrainable objects; constraints can be added to and removed from them. Constraints do not access object data directly, they only use public methods of objects. We opt for not accessing object data directly in order to be independent from the implementation of the glyph class, to allow objects to be distributed over the network, and to maintain encapsulation. Current constraint solvers (e.g. [24]) are based on atomic variables; using object data directly as these variables would violate the encapsulation principle. Instead, we define several controller classes which install a communication between protocol constraints and constrainable glyphs. Typically constraints are contained in the behavior graph.

One of the most frequently used constraint types is the controller monoglyph. It associates an access method of a target glyph with a time-dependent value. In general, controllers can be evaluated directly, i.e. they do not depend on other constraints. A time-dependent rotation, for instance, consists of two parts: a rotation glyph in the geometry graph and a rotation controller in the behavior graph. The rotation controller is a constraint which associates a mapping with a rotation angle. Whenever a new point in time is reached, the controller determines the new rotation angle according to the time-to-angle mapping and assigns the new angle. Controller glyphs remember the original value of the attribute they control. If the controller glyph dies, it restores this original value.

Frequently time-dependent behaviors are specified by numerical constraints. MAM defines a generic *numerical constraint.* It determines a numerical value for a given moment. Frequently used numerical constraints are linear interpolation mappings or B-spline mappings. Time sensors are numerical constraints which return the current system time. numerical constraints can be composed to form networks of constraints by connecting them with numerical operations, such as addition, multiplication, etc., and are managed by an external constraint solver. Controller constraints communicate the result of such a network to an object by sending the resulting value with the appropriate method to the object.

To turn individual glyphs on or off, the alive state of glyphs can be constrained by *life constraint glyphs*, e. g. an actor defined in the geometry graph. Per default, the geometry graph is alive during the whole animation. To make an object disappear in some parts of the animation, we install a life time constraint glyph in the behavior graph attached to this object. As long as this constraint is alive, the associated object is turned off. Individual objects of the geometry graph can also be restricted in their life time by *life duration constraint glyphs.* These constraints are added to the object and keep the object alive during the specified time interval, and turn it off otherwise.

3.4 Editing and Persistency of Graphs

Animation and simulation applications can be based on building geometry graphs and behavior graphs. The graph structure and the glyph concept lead to simple editing techniques for animations: editing consists of creating and combining glyphs.

Geometry graphs and behavior graphs are persistent. Both are written in depth-first order to a C++ stream, respectively are read from the stream. The task of writing and reading is delegated to the glyphs. Glyphs specify which of its data values are persistent, and can textually represent these val-

ues. Persistency of glyph associations is supported by an association manager. Persistency is available to developers of new glyph classes through inheritance.

Persistent graphs allow to transfer complex animated scenes in a compact form. In comparison to MPEG [9], this approach offers the following advantages:

- Models retain their geometric and behavioral building blocks.
- Models are significantly smaller for large animation sequences.
- The realization can take full advantage of locally available hardware.
- Models retain interactive behavior.

Of course, complex and rendering time consuming models must be prebuilt and compressed into frame sequences as with MPEG. However, in the near future improved hardware graphics capabilities allow to play and view interactive synthetic films in real time.

4 Dynamic Graph Modification

Sensors are glyphs which describe interactive behavior. Based on callbacks (modeled as objects), they modify the subgraphs of the behavior graph. Imagine the following interaction behavior: A wheel represented by a thin disc is rotated by a select-and-drag interaction. While dragging the mouse, the wheel rotates. If the dragging acceleration exceeds a certain threshold, the wheel shall continue to rotate. This complex interactive behavior can be achieved by a dynamic modification of the behavior graph as follows (s. Fig. 4):

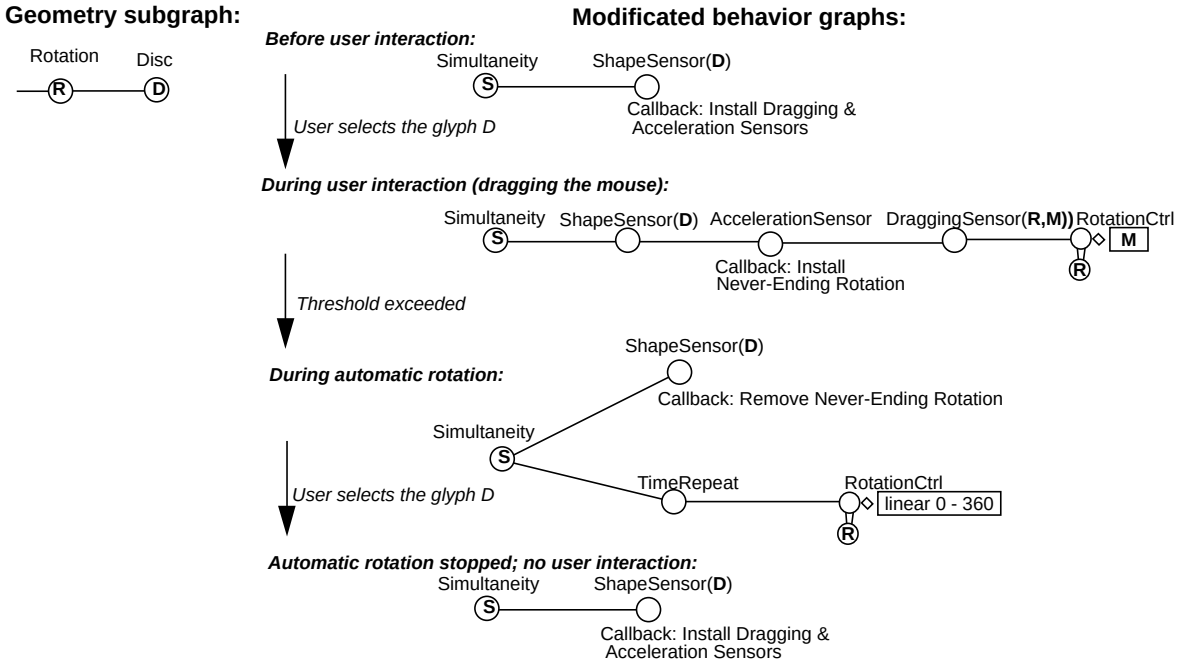

Figure 4. Dynamic modification of graphs during interaction.

A shape selection sensor waits for an event which selects glyphs. Whether a glyph has been selected may be tested with the picking facility of many rendering packages. If the shape is selected, it adds an acceleration sensor, a dragging sensor, and a rotation controller for **R**. The dragging sensor informs the mapping **M** about the mouse movements. **M** is used by the rotation controller to calculate the rotation angle . The acceleration sensor waits until the acceleration exceeds a critical value. In this case, the acceleration sensor removes itself and installs a never-ending rotation for the **R** and a new

callback for the shape sensor. If the user selects the disk, the never-ending rotation subgraph is removed.

5 Implementation

We implemented the proposed methodology in MAM, a C++ object-oriented 3D modeling and animation toolkit. MAM applications can be divided logically in three layers:

- *Application layer*: Manages application objects.
- *Modeling layer*: Manages the geometry and behavior graph which visualize the application data.
- *Rendering layer*: Communicates with the underlying rendering package.

Application designers normally interact with the modeling layer, whereas developers of new rendering features interact with the rendering layer. The system structure is illustrated in Fig. 5.

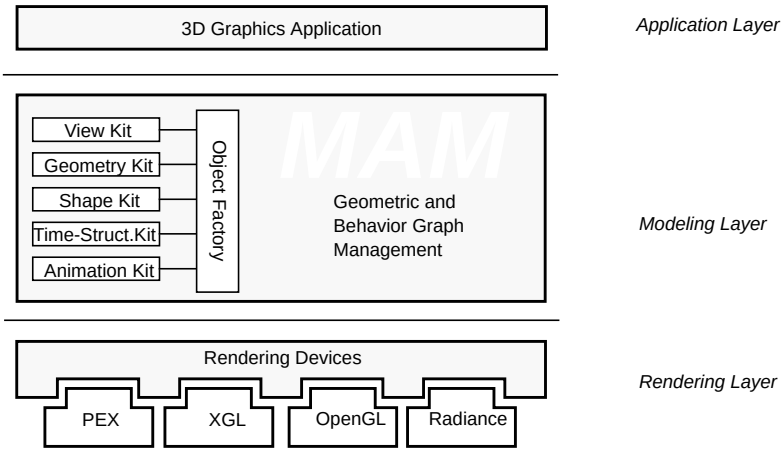

Figure 5. MAM application structure.

5.1 Modeling Layer and Rendering Layer

The modeling layer mediates between the application and the rendering layer. The application does not have to care about the rendering, since glyphs transform themselves into objects suitable for the rendering layer.

Virtual rendering devices are objects which understand a set of rendering operations called the *rendering protocol*. This rendering protocol is based on a set of abstract graphical data types. These include *rendering primitives* (e.g. lines, facets), *rendering attributes* (e.g. textures, surface properties), and numerical data types (projection matrices). For specific rendering devices we can derive specialized rendering attributes to access rendering package specific features. For instance, the base edge style defines edge color and edge width, the XGL edge style defines additionally edge antialiasing.

The rendering protocol standardizes the access to the underlying capabilities of low-level rendering packages. Therefore MAM applications can exchange rendering devices even at run time. Furthermore, this abstraction provides an easy interface to complex and hard-to-learn low-level graphic packages.

New renderers can be added to the rendering layer if they implement the rendering protocol. These operations include the capability to render a minimal set of rendering primitives and rendering attributes. Adding a new rendering device type does not affect the modeling layer or the application layer.

To enable rendering of more complex primitives as those guaranteed by the rendering protocol the modeling layer and the rendering layer can communicate. For example, glyphs can inquire from the

renderer the types of primitives supported and their respective rendering costs. Therefore glyphs can improve rendering by using the most efficient rendering primitives of a renderer. If the set of rendering primitives were fixed the capabilities of high-performance renderers would be restricted.

5.2 Kits and Classes

MAM is implemented as a library of approximately 150 C++ classes. We distinguish between modeling classes and rendering classes. The former define glyph types, the latter define abstract graphical data types and rendering devices. To provide an easy access to classes, glyphs are produced by kits. *Kits* are object factories which are user-oriented interfaces giving easy access to frequently used objects and object configurations. They hide much of the complexity of class hierarchies and class constructors. For instance, the block glyph has only one general constructor requesting the origin and three spanning vectors. The shapekit produces with this class general parallelepipeds, rectangular blocks, and cubes. Thus, eliminating the need for subclasses leads to slimmer class hierarchies.

```
class ShapeKit : public virtual Kit {
public:
   ...
   virtual GShape& block(MVector centre, Vector dir_x, MVector dir_y, MVector dir_z) const;
   virtual GShape& block(MVector centre, float size_x, float size_y, float size_z) const;
   virtual GShape& cube(MVector centre, float size) const;
   ...
};
```

Kits also manage the persistency of glyphs, i.e. they handle the storage and the retrieval of glyphs from streams. MAM defines the following kits: geometry kit, shape kit, viewing kit, time-structuring kit, and animation kit. Additionally, the MAM kit can link application-defined kits at run time.

Example

The listing below shows the source code to construct the virtual hall geometry graph shown in Fig. 2.

```
01 MAM mam;                                                  // object factory
02 GGlyph& geometry_graph = mam.simple_view() *                // constrained camera glyph
03 ( mam.geometricgroup()                                      // default: unit cube
04 +  mam.color(.7,.7,.7) * ( mam.geometricgroup()             // 70% gray hall
05    + mam.cylinder(MVector(.05,0,.05), MVector(.05,.9,.05), .1) // first column
06    + mam.block(MVector(0,0,0), .1, .03, .1)                   // and its plinth
07    + ... // more columns
08    )
09 +  mam.reflection(.8,.8)*mam.checkerboard(                  // specular reflection floor
10      MVector(0,0,0), MVector(1,0,0), MVector(0,0,1), 10, 10, 0,0,1, 1,0,0
11    )                                                        // tesselation: 10x10
12 +  mam.color(0,1,0) * (mam.geometricgroup() // green table
13    + mam.cylinder(MVector(.5,0,.5), MVector(.5,.18,.5), 0.05)) // table-leg
14    + mam.block(MVector(.4,.18,.4), .2, .02, .2)               // table-top: thin block
15    + mam.translate(MVector(.4,.2,.4))*mam.scale(MVector(.5,.5,.5))*
16      mam.color(0,0,1)*mam.turtlepath(...)                     // logo as turtle path
17    )
18 +  mam.positional_light(MVector(.5,.9,.5), .07)            // lightsource at the ceiling
19 );
```

Figure 6. Code for the virtual hall geometry graph of Fig. 2.

First, we construct a MAM object factory. Line 2 creates a camera glyph together with a view plane constraint; the function '`MAM::simple_view`' returns a reference to the camera glyph. Next, the bodyglyph of the camera, a geometric group, is constructed. The first child of this group is another geometric group modified by a color glyph. To connect glyphs, the '`*`' and '`+`' operators are overloaded. '`*`' adds bodyglyphs to monoglyphs; '`+`' adds children to polyglyphs. In line 4, all parts of the hall (i.e. columns and plinths) are added. '`MAM::cylinder`' returns a cone glyph. As parameters we pass both endpoints and the radius. 3D points are given as `MVector` objects. Line 6 adds a block glyph given by its corner and its width, height, and length. The whole geometric group is modified by the color glyph added in line 4. Colors can be passed as `MColor` objects or (for convenience) directly by its RGB values. The checkerboard is built in lines 9-10; we pass the origin, two spanning vectors, the tessellation (10x10 quads), and the colors. Lines 12-13 construct the table. The table consists of a cyl-

inder and a thin block. The logo built by a turtle path glyph is scaled and translated by the corresponding monoglyphs (lines 15-16). The logo takes as arguments 3D turtle commands given as a string. Note that shape glyphs do not store explicitly transformation matrices; they are composed and passed as rendering arguments during the traversal of the geometry graph. Line 18 adds a small light emitting sphere which is positioned at the ceiling.

The behavior graph is built analogously (s. listing below): '`MAM::simultaneity`' and

```
01 MAM mam;
02 GGlyph& behavior_graph = mam.simultaneity()
03 + geometric_graph
04 + ( mam.sequence()
05     + mam.duration(5)*mam.do_scale(S, mam.linear(MVector(0,0,0), MVector(.2,.2,.2)))
06     + ( mam.sequence()
07         + mam.duration(20)*mam.time_repeat(10)*mam.do_rotate(R, mam.linear(0,360))
08         + ( mam.sequence()
09             + mam.do_scale(S, mam.linear(MVector(.2,.2,.2), MVector(.2,.5,.2)))
10             + mam.do_scale(S, mam.linear(MVector(.2,.5,.2), MVector(.2,.2,.2)))
11           )
12       )
13     + mam.duration(5)*mam.do_scale(S, mam.linear(MVector(.2,.2,.2), MVector(0,0,0)))
14   );
```

Figure 7. Code for the virtual hall behavior graph of Fig. 2.

'`MAM::sequence`' construct chronological groups with corresponding chronological layout. As first child, we add the camera which will be alive during the whole animation. This is done explicitly, since there may be animations which switch between different cameras. '`MAM::duration`' in line 5 returns a monoglyph which defines the time requirement of its bodyglyph. '`MAM::do_scale`' returns a constraint which sets the scale vector of the scale glyph **S** according to a given constraint. In this example, the scale vector is determined by a linear time-to-vector mapping constructed by '`MAM::linear(MVector, MVector)`'. This numerical constraint interpolates the given vectors during the span of time of the constraint glyph. In line 7, we stack together three monoglyphs: duration, time repeater, and rotation constraint associated with the rotation glyph **R** of the geometry graph and a linear numerical constraint. In this case, '`MAM::linear(float,float)`' returns a numerical time-to-float constraint. It interpolates during the assigned life time of the constraint (10 sec.) the interval [0..360]. The duration of the sequence added in line 8 is determined implicitly through the duration in line 7: Since the sequence does not define own time requirements, the time requirements of its parent will be assigned to it.

6 Conclusions

We have presented an object-oriented methodology for geometric, chronological, and behavioral modeling. Models are built by geometry graphs, behavior graphs and constraints. Graph nodes and constraints are represented as glyphs, i.e. as lightweight basic building blocks which share a common communication protocol and which can be combined in almost arbitrary manner.

Shape glyphs, viewing glyphs, and prioritized attribute glyphs cover the traditional three-dimensional modeling, whereas time-structuring glyphs like chronological groups, time repeater, time reversals, and duration placements offer chronological modeling techniques. All glyphs are constrainable and maintain a list of constraints which apply to them. Directly attached constraints allow the integration of a wide set of simple constraints. Constraints which cannot be resolved directly are integrated as behavior graph glyphs and managed by a constraint solver.

Building models with highly reusable small building blocks leads to comprehensible modeling techniques and to faster and simpler implementations. Particularly, the graph notation enables geometric and animation editing. Dynamic behavior changes can be modeled by dynamic modifications of the behavior graph.

The methodology is implemented in MAM as a strongly fine-grained object-oriented system which is broken up in several object factories to provide easy access to glyphs. The system is implemented as C++ class library and portable across different rendering techniques and rendering packages. MAM works with several commonly used rendering packages and takes full advantage of their capabilities by establishing a communication between the modeling and the rendering layer.

Currently we are integrating more rendering packages, more modeling techniques, e.g. fractal geometry, and constraint-based modeling glyphs. Furthermore, we are integrating more three-dimensional interaction behaviors for virtual environments.

References

[1] David E. Breen et al. The Clockworks: An Object-Oriented Computer Animation System. In Eurographics ‘87. G. Maréchal (Ed.), pp. 275-282.

[2] Paul Calder and Mark Linton. Glyphs: Flyweight objects for user interfaces. Proceedings of the *ACM SIGGRAPH Symposium on User Interface Software and Technology*, Snowbird, Utah, pp. 92 - 10, October 1990.

[3] Paul Calder and Mark Linton. The Object-Oriented Implementation of a Document Editor. *Proceedings of OOPSLA ‘88* (ACM SIGPLAN Notices 27 (10)), pp. 154 - 165, October.

[4] Parris K. Egbert and William J. Kubitz. Application Graphics Modeling Support Through Object Orientation.In *COMPUTER*, pp. 84 - 91, October 1992.

[5] Larry Koved and Wayne L. Wooten. GROOP: An object-oriented toolkit for animated 3D graphics. In *ACM SIGPLAN NOTICES OOPSLA ‘93*, 28(10):309-325, October 1993.

[6] Mark A. Linton, Paul R. Calder, and John M. Vlissides. The Design and Implementation of InterViews. *Proceedings of the USENIX C++ Workshop*, Santa Fe, New Mexico, November 1987.

[7] PHIGS+ Committee, Andries van Dam, chair. PHIGS+ Functional Description, Revision 3.0. *Computer Graphics*, 22(3), pp. 125 - 218 July 1988.

[8] Steven Upstill. The RenderMan Companion. A Programmer’s Guide to Realistic Computer Graphics. Addison-Wesley, Reading, MA, 1990.

[9] Didier Legall. MPEG - A Video Compression Standard for Multimedia Applications. Communications of the ACM, 34(4), pp. 47-58, April 1991.

[10] Paul S. Strauss. IRIS Inventor, A 3D Graphics Toolkit. In *ACM SIGPLAN NOTICES OOPSLA ‘93*, 28(10), pp. 192 - 200, Oct. 1993.

[11] Peter Wisskirchen. *Object-Oriented Graphics: From GKS and PHIGS to Object-Oriented Systems*. Springer-Verlag, Berlin, 1990

[12] Graphics Library Programming Guide, Silicon Graphics Computer Systems, Mountain View, Calif., 1991.

[13] Randi J. Rost, Jeffrey D. Friedberg, and Peter L, Nishimoto. PEX: A Network-Transparent 3D Graphics System. In *IEEE Computer Graphics & Applications*, pp. 14 - 26, July 1989.

[14] Solaris XGL 3.0.1 Reference Manual, SunSoft, Sun Microsystems, Inc., Mountain View, CA 94043, 1993.

[15] Conat Elliot, Greg Schechter, Richy Yeung, and Salim Abi-Ezzi SunSoft, Inc. TBAG: A High Level Framework for Interactive, Animated 3D Graphics Applications. In *Proceedings of SIGGRAPH ‘94*, pages 421-434.

[16] Mark A. Tarlton and P. Nong Tarlton. A framework for dynamic visual applications. In *1992 Symposium on Interactive 3D Graphics*, pages 161-164, 1992

[17] Robert C. Zeleznik, et al. An Object-Oriented Framework for the Integration of Interactive Animation Techniques. In *Proceedings of SIGGRAPH ‘91*. In Computer Graphics 25, 4, (July, 1991), 105-111.

[18] Gregory J. Ward. The READIANCE Lighting Simulation and Rendering System. In Proceedings of SIGGRAPH ‘94, pp. 459-472.

[19] Josie Wenecke. *The Inventor Mentor. Programming Object-Oriented 3D Graphics with OpenInventor*, Release 2. Addison-Wesley, 1994.

[20] Bilge Erkan and Bülent Özgüç. Object-Oriented Motion Abstraction. In *The Journal of Visualization and Computer Animation*, 6(1):49-65.

[21] T. S. Chua, W. H. Wong and K. C. Chu. Design and implementation of the animation language SOLAR. In *New Trends in Computer Graphics, Proceedings of CG International*, N. Magnenat-Thalmann and D. Thalmann (Eds.), Springer Verlag, 1988, pp. 15-26.

[22] R. Maiocchi and B. Pernici. Directing an animated scene with autonomous actors. *The Visual Computer*, 6, pp. 359-371, 1990.

[23] James Rumbaugh. et al. *Object-Oriented Modeling and Design*. Prentice-Hall, Inc., 1991.

[24] Michael Sannella. SkyBlue: A Multi-Way Local Propagation Constraint Solver for User Interface Construction. Dept. of Computer Science and Engineering, FR-35, Univ. of Washington, 1994.

Appendix: Class Hierarchy (Core Classes Only)