In: Journal of Visualization and Computer Animation, 8(1):33-64, 1997

Object-Oriented 3D Modeling, Animation and Interaction

JÜRGEN DÖLLNER AND KLAUS HINRICHS

Institut für Informatik, FB 15, Westfälische Wilhelms-Universität
D-48149 Münster, Germany

SUMMARY

We present an object-oriented 3D graphics and animation framework which provides a new methodology for the symmetric modeling of geometry and behavior. The toolkit separates the specification of geometry and behavior by two types of directed acyclic graphs, the geometry graph and the behavior graph, which are linked together through constraint relations. All geometry objects and behavior objects are represented as DAG nodes. The geometry graph provides a renderer-independent hierarchical description of 3D scenes and rendering processes. The behavior graph specifies time- and event-dependent constraints applied to graphics objects. Behavior graphs simplify the specification of complex animations and 3D interactions by providing nodes for the management of the time and event flow (e.g. durations, time layouts, time repeaters, actions). Nodes contain, manipulate and share instances of constrainable graphical abstract data types. Geometry nodes and behavior nodes are used to configure high-level 3D widgets, i.e. high-level building blocks for constructing 3D applications. The fine-grained object structure of the system leads to an extensible reusable framework which can be implemented efficiently.

KEY WORDS Computer Animation Object-oriented Visualization 3D interaction Virtual Reality

1. INTRODUCTION

Interactive, animated 3D applications are difficult to develop. Many paradigms and metaphors for object-oriented 3D graphics have been proposed. However, the development of interactive and animated 3D applications is still difficult for several reasons.

Often geometric modeling is strongly related to a specific rendering system. The developer has to understand the underlying rendering library in order to work with the 3D graphics toolkit. Applications are difficult to adapt to rendering systems using a different rendering-technique (e.g. from an immediate mode library to a radiosity-based library) because the toolkit design is based on assumptions about the rendering pipeline. In order to overcome this problem, toolkits such as GRAMS [7] and GROOP [12] separate the graphics layer from the rendering layer. They provide a set of built-in shapes and properties which can be used to construct objects in the graphics layer. Main disadvantages of this concept are the loss of application semantics in the rendering layer and the communication overhead between the graphics and the rendering layer. Furthermore, current toolkits concentrate on an object-oriented representation of scene components but do not extend object orientation to the control and specification of the image synthesis process. If image synthesis processes are modeled as toolkit components, algorithms and rendering techniques (e.g. producing motion blur or magic lenses) can be encapsulated and provided to the developer.

Furthermore, current toolkits do not treat behavioral modeling at the same level of abstraction as geometric modeling. By *behavioral modeling* we mean the modeling of time-dependent and event-dependent actions and processes[1]. Behavioral modeling is as important as geometric modeling because the specification of time-varying and event-dependent properties is the key for animation and interaction control. Traditional toolkits do not provide an explicit object-oriented approach for time and event handling. Instead, they support behavioral modeling through procedural extensions leading to a break in the system architecture.

Finally, there exists no clear strategy how to integrate time- and event-dependent constraints for geometry components and animation components. Animated and interactive applications contain lots of constraints among the scene components. TBAG [26], based on a functional approach, appears to be one of the first systems integrating constraints as first class objects. No concept for the object-oriented integration of constraints is available for systems based on declarative scene descriptions.

Our contribution provides a framework for the uniform modeling of both geometry and behavior. The main goals of MAM/VRS, the “Modeling and Animation Machine” and the “Virtual Rendering System” are

- to specify geometry and behavior separately in *geometry nodes* and *behavior nodes* which are organized in two directed acyclic connected graphs, the *geometry graph* and the *behavior graph*,
- to provide renderer-independent graphics objects which can be visualized by geometry nodes and constrained by time- and event-related behavior nodes, and
- to construct high-level 3D widgets by combining geometry nodes and behavior nodes.

For the representation of class hierarchies and object relationships we use the notation of the Object Modeling Technique OMT [24].

2. OVERVIEW

We start with an overview of the system architecture, introduce graphical abstract data types used throughout the system, and define the basic node types of the toolkit.

2.1 System Architecture

The toolkit consists of two main parts, the rendering layer and the graphics layer (Fig. 1). The *rendering layer* provides *graphics objects* which are instances of graphical abstract data types and represent graphical entities, e.g. shapes, transformations, colors and textures. The rendering layer is based on low-level 3D rendering libraries (e.g. OpenGL and PEX), their functionality is encapsulated in *virtual rendering devices*. The uniform interface of the virtual rendering devices allows us to exchange them at run time without having to modify the application. Since the virtual rendering devices operate on renderer-independent graphics objects, it is easy to integrate new low-level 3D rendering libraries into the system. In general, a virtual rendering device is associated with a window in which it visualizes graphics objects.

The *graphics layer* is responsible for the management of *geometry graphs* and *behavior graphs*. *Node factories* support the construction and manipulation of nodes and graphs. *Geometry nodes* visualize associated graphics objects, and *behavior nodes* apply constraints to them. The graphics layer and the rendering layer are tightly coupled because geometry nodes and behavior nodes manipulate and operate on shared graphics objects.

The toolkit is implemented in C++. We provide a C++ and a [incr Tcl] [16] application programming interface. Motif and [incr Tk] [17] user interface bindings are available. Currently, we have integrated the following low-level 3D rendering libraries: OpenGL, PEX, XGL, RenderMan [29] and Radiance [30]. Portability is guaranteed due to different application pro-

1.We use the term ‘behavior’ in a technical sense because it captures both animated and interactive aspects of the model description. However, the term has a different meaning in the context of behavioral animation and the modeling of artificial life.

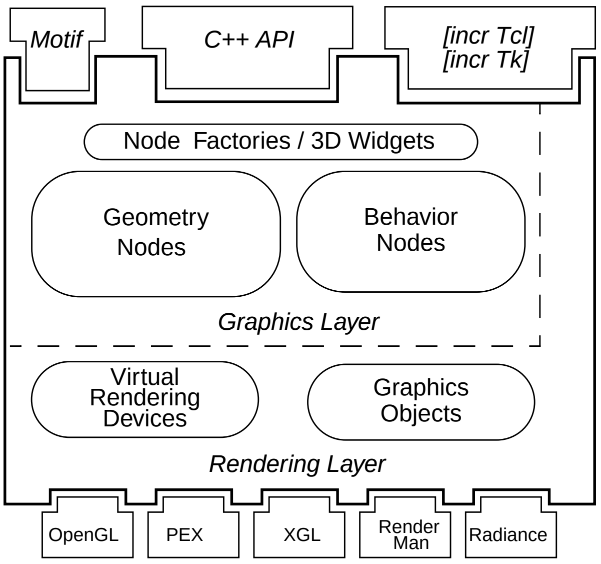


Figure 1. The System Architecture.

gramming interfaces and independence from window systems and low-level 3D rendering libraries.

2.2 Graphical Abstract Data Types

Graphical abstract data types support the tight coupling of the graphics layer, the rendering layer, and the application layer. Consider an application managing 3D arrows. An arrow object is an application object which has to be represented in the graphics layer in terms of nodes. To visualize an arrow, we could combine a cone node and a cylinder node. However, two problems arise: 1. The application stores redundant information, i.e. for each arrow a cone node and a cylinder node. 2. The arrow semantics is lost in the graphics layer and the rendering layer. The semantics could be useful to optimize rendering and intersection algorithms.

In our approach, applications can define application-specific types as new graphical abstract data types and integrate them in the rendering layer and the graphics layer. For example, an arrow class can be registered in a virtual rendering device together with an arrow rendering algorithm. The application creates its arrow objects and embeds them in generic shape nodes. If these shape nodes are requested to render themselves, they pass the arrow objects to the virtual rendering device which in turn uses the arrow rendering algorithm. This approach maintains the application-semantics in all three layers and avoids data duplication and data conversion.

The design of graphical abstract data types can be characterized by their flyweight and elementary nature. The flyweight design [4] ensures that they are as minimal and as small as possible. They do not include any context information and make no assumptions about how they are visualized. A sphere graphics object, for example, stores its radius and midpoint, but does not include a transformation matrix, a color, a virtual rendering device or a surface approximation. Graphical abstract data types are elementary because shapes and attributes contain only the information implied by their types, but no context information. Only a fine-grained hierarchy of elementary types can be used without redundancy by all kinds of applications. These

design considerations allow us to use graphics objects in large numbers and to implement them efficiently [14].
Furthermore, the instances of graphical abstract data types are shareable objects which can be multiply referenced and are automatically removed when they become completely dereferenced. Shareability provides an efficient management of dynamically allocated objects.
We use the term *graphics object* as a synonym for an instance of a graphical abstract data type. A 3D application is specified by a collection of geometry nodes and behavior nodes which are associated with graphics objects. The animation and interaction is specified by nodes which apply constraints to these graphics objects.

2.3 Nodes

Nodes are the basic building blocks used in the construction of animated, interactive 3D applications. They are realized as instances of node classes. Six base classes are derived by multiple inheritance from the node structure classes *MLeaf*, *MMono* and *MPoly*, and the node semantics classes *MGeometry* and *MBehavior* (Fig. 2).

2.3.1 Structural Properties of Nodes

The structural properties define how a node can be linked to other nodes and restrict therefore the position of a node in a DAG. A *leafnode* terminates a subgraph and does not propagate messages. A *mononode* has at most one child node, called *body*; all messages received by a mononode are passed to its body if it exists. Mono nodes are mainly used to partially redefine protocols passed through them. A *polynode* manages an ordered sequence of arbitrarily many child nodes, it passes messages to all or to a subset of its child nodes.
We introduce these types to ensure the following properties of DAGs: All traversals starting from an arbitrary node end in a leafnode, and the traversal of a mononode produces no side-effects, i.e. the mononode does not affect its parent or siblings, but only its body.

2.3.2 Semantic Properties of Nodes

We distinguish between *geometry nodes* and *behavior nodes*. Nodes must be able to understand a set of semantic-specific communication protocols. For example, geometry nodes define protocols for intersection tests and rendering, behavior nodes for inquiring time requirements, for synchronization and event handling. Each geometry node and each behavior node is either a leafnode, mononode or polynode (Fig. 2).

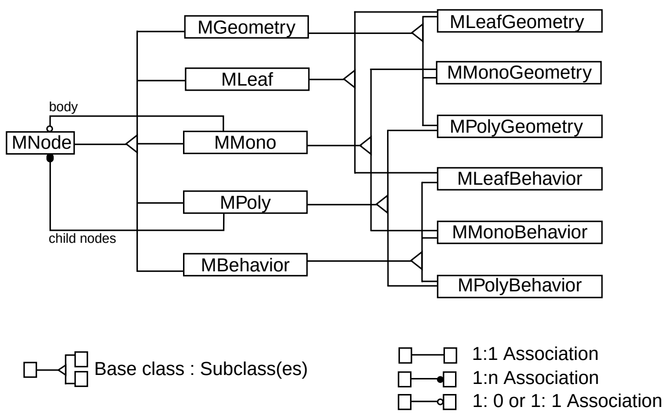


Figure 2. Base Node Classes.

To construct a 3D application, we combine geometry nodes to a geometry graph and behavior nodes to a behavior graph. A geometry graph specifies 3D scene components, their appearance and modeling coordinate systems. A behavior graph specifies how these components vary in time and how they react to events. The next two sections discuss both graphs in detail.

3. GEOMETRY GRAPHS

Geometry graphs are declarative and rendering-library independent descriptions of 3D scenes. In contrast to traditional systems, geometry node classes are oriented towards the user's view of a 3D scene instead towards a specific rendering library. Developers can use geometry graphs without detailed knowledge of the underlying rendering library.

Four categories of geometry nodes are provided to compose 3D scenes: shapes, attributes, geometry groups, and image controllers. *Shapes* are leafnodes which embed shape graphics objects. *Attributes* are mononodes which embed attribute graphics objects. *Geometry groups* are polynodes which link several subgraphs. The scene objects described by the subgraphs are arranged based on the group's layout strategy. *Image controllers* manage the image synthesis process, e.g. each camera initializes a virtual rendering device and initiates the rendering of its assigned scenes. Scene nodes are the *root nodes* of geometry graphs.

3.1 A Sample Geometry Graph

To illustrate the MAM/VRS concepts we develop an interactive application for animating an algorithm which visually constructs the Voronoi diagram [21] of a given set of points. The points are represented by small spheres. During the animation the points are lifted from the floor onto a paraboloid (Fig. 3a), planes that are tangential to the lifted points on the paraboloid are visualized (Fig. 3b), and finally the Voronoi diagram is exposed by looking into the open paraboloid (Fig. 3c).

The geometry graph of this application (Fig. 4) is discussed in this section, the behavior graph is explained in section 4.

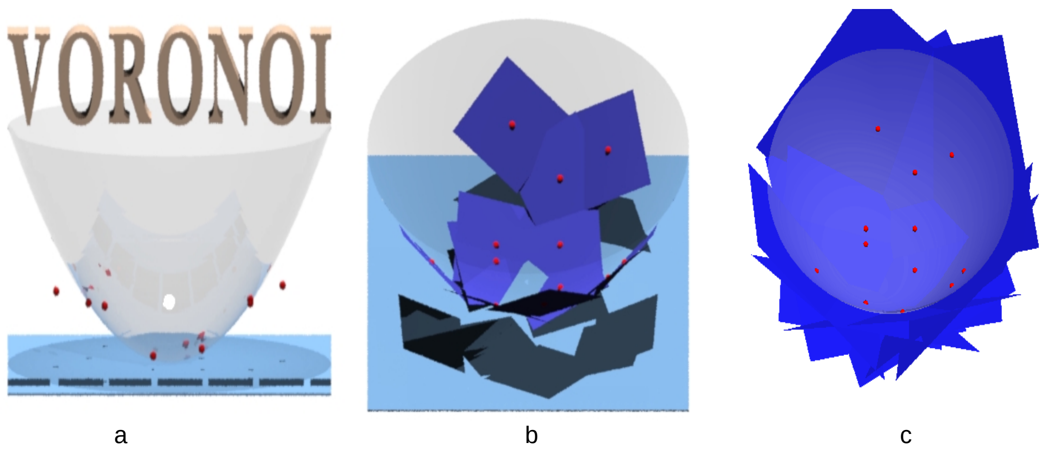


Figure 3. Animation of the Construction of a Voronoi diagram. Lifting points onto the paraboloid (a). Fading in the tangential planes (b). Exposing the Voronoi diagram inside the paraboloid (c).

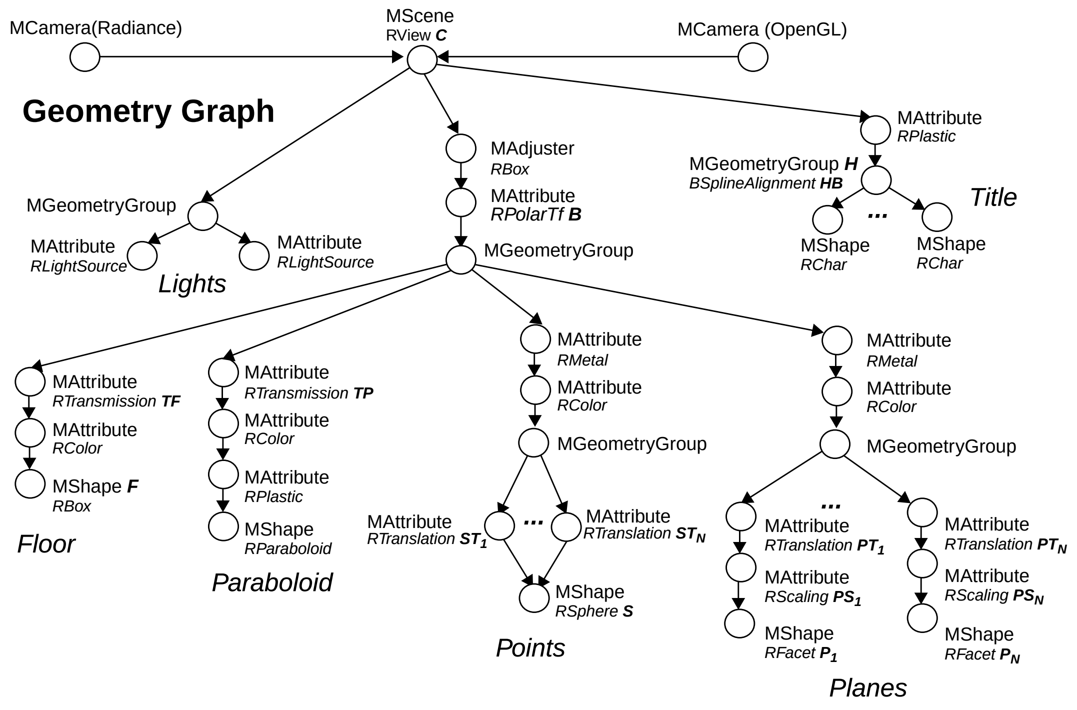


Figure 4. Geometry Graph for the Algorithm Animation.

3.2 Shapes

Shape nodes are leafnodes of the geometry graphs. A shape node specifies a visual component of a 3D scene and is associated with one or more graphics objects. In the example (Fig. 4), the *RFacet* objects $\boldsymbol{P_1}$, ..., $\boldsymbol{P_N}$ are referenced by shape nodes. Graphics objects and nodes can be shared since they are shareable objects, e.g. all points are represented by only one shape node and an associated *RSphere* object ***S*** (Fig. 4).
Each *MShape* node is associated with a *RShape*$^{2)}$ object (Fig. 2). A *MConvexHull* node is associated with a vertex-based graphics object (*RVertexBased*) and visualizes its convex hull through a triangle set (*RTriangleSet*). The *MNormalViewer* node visualizes the facet normals of a polyhedron graphics object (*RPolyhedron*) by glyphs (e.g. small arrows) on top of each facet.
Shape classes and geometry node classes are specified in separate hierarchies (Fig. 2) in order to decouple their functionality. This reduces the implementation complexity and results in lightweight objects which can be used in large numbers.
The design of the MAM/VRS shape hierarchy is semantic-oriented, i.e. similarities in class methods are used as criterion for inheritance. Traditional toolkits frequently use implementation similarities as a criterion, e.g. a line set class is derived from a point set class. However, from a user's point of view similarities in the semantics of shapes are more important than similarities between their internal implementation-dependent representations. For example, all triangle, quadrilateral, line-based and point-based shapes share methods to manipulate vertices and vertex information, therefore all of them are subclasses of an abstract base class *RVertexBased.*

2.The syntax convention for class names is as follows: Node classes start with ‘M’, graphical abstract data types start with ‘R’ (rendering/graphics objects).

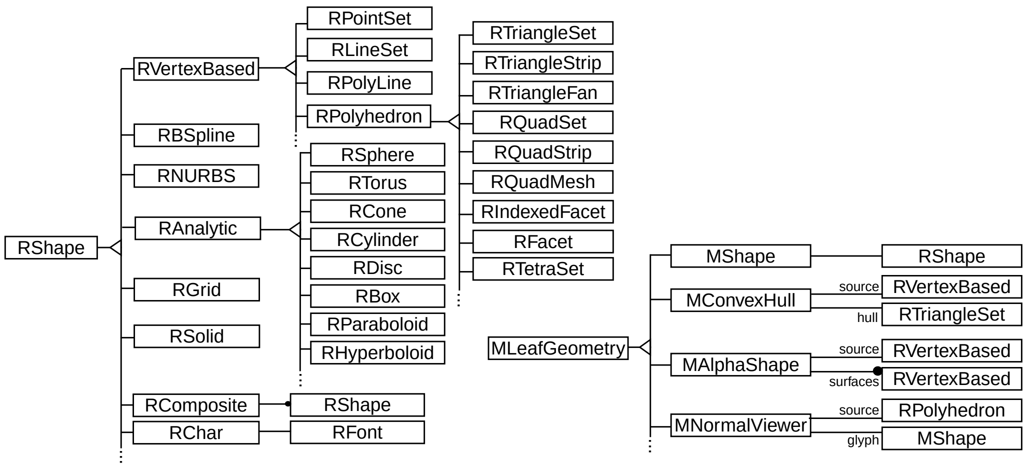


Figure 5. Shape Classes.

3.3 Attributes

Attribute nodes are geometry mononodes. They determine the visual appearance and control the coordinate systems of shape nodes. To keep the attribute mechanism side-effect free, an attribute node affects only its subgraph and does not influence sibling nodes.

The *MAttribute* node embeds an arbitrary attribute graphics object. For example, there are attribute nodes which constrain or modify attributes. The *MBrightness* node changes the brightness of the color attribute applied to its body. The *MAdjuster* node scales and translates the shapes contained in its body so they fit into a target volume. In Fig. 4 it is used to fit the main scene components into a given *RBox*.

As for shape classes, attribute classes and geometry node classes are specified in separate hierarchies (Fig. 6) in order to decouple their functionality. The design of the MAM/VRS attribute class hierarchy generalizes the visual attributes found in standard rendering libraries such as OpenGL and RenderMan. The core attributes can be evaluated by all rendering devices. To use specific capabilities of renderers, attribute subclasses can be integrated, e. g. for the

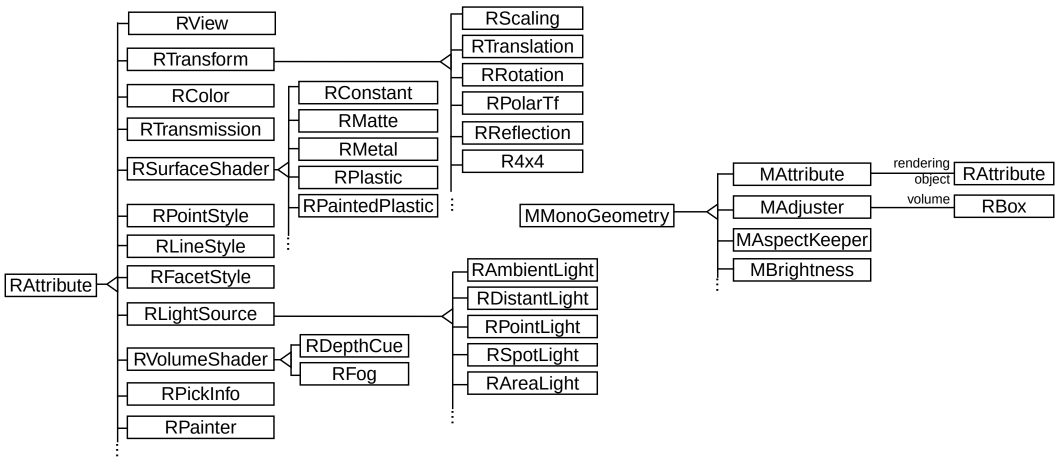


Figure 6. Attribute Classes.

RenderMan rendering device we supply more subclasses of volume shaders and surface shaders.
There are several reasons to model attributes as first-class objects. Attributes are *objects*, not parameters of shapes. Therefore, application-specific attributes can be easily integrated into the toolkit since the number of attributes attached to a shape is not fixed. Existing attribute classes can be subclassed to access specific features of the underlying rendering library. Furthermore, we can use the same technique for constraining attributes and shapes since both are represented by graphics objects.

3.4 Geometry Groups

The *MGeometryGroup* nodes are geometry polynodes which organize their child nodes according to *geometry layouts* (Fig. 7). A geometry layout can transform the modeling coordinate system of its child nodes and determine which of them are traversed. Examples:

- *Unconstrained*: No restrictions for child nodes.
- *BSplineAlignment*: Child nodes are positioned and oriented along a BSpline and optionally scaled to a target volume. For example, the character objects of the title are aligned along a BSpline ***HB*** and organized in the geometry group ***H*** (Fig. 4).
- *Switch*: Child nodes are selected according to the type of virtual rendering device used to render the node, or according to the view plane size of the child node's image.

3.5 Image Controllers

Image controllers manage the image synthesis process (Fig. 7). A *MScene* node is a geometry polynode which orientates and projects its child nodes. It defines a virtual environment with a default modeling coordinate system and default attributes. An associated *RView* graphics object determines the view orientation and projection matrix. *MBlurScene* and *MLens* are specialized scene nodes. A *MBlurScene* blurs the image. *A MLens* node restricts the drawing area for its child nodes to a certain region of the view plane area. For example, lens nodes can be used to implement magic lenses [3].
A *MCamera* object is associated with a virtual rendering device and a list of scenes. It is responsible for redrawing and performing operations on these scenes.
A *Renderer* represents a virtual rendering device which provides methods to render shapes, to push and to pop attributes, and to control the rendering buffers (e.g. z-buffer, accumulation buffer, stencil buffer). We provide virtual rendering devices for OpenGL, XGL, PEX, RenderMan and Radiance. These virtual rendering devices are implemented as specialized *Renderer* classes.
To visualize a shape, a virtual rendering device uses a rendering algorithm which is encapsulated in a *RPainter* class. A painter is an attribute (Fig. 6) which is associated with a shape class. Therefore, applications can switch between visualizations because painters can be

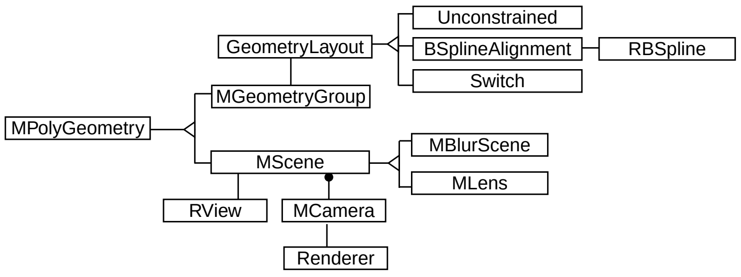


Figure 7. Geometry Groups and Image Controller Classes.

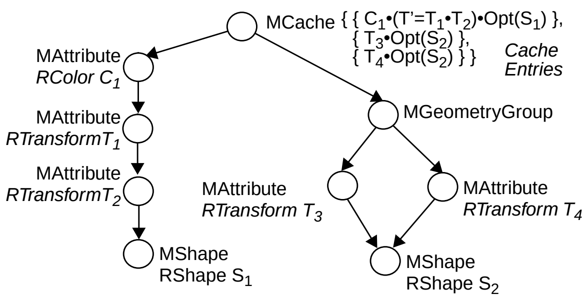
exchanged for each shape class. For example, a point set can be visualized in low-quality as pixels and in high quality as spheres.
MAM/VRS defines painters for all built-in shape classes. In order to optimize the rendering process, additional painters can be provided for specific rendering toolkits. For example, the arrow painter converts an arrow into a cylinder shape and a cone shape. MAM/VRS provides an additional faster painter for arrows which is optimized for OpenGL. The base painter class *RPainter* provides methods to convert a high-level shape into lower-level built-in shapes, e.g. a cylinder can be converted into a triangle mesh.

3.6 Caching of Scene Descriptions

Systems like Grams [7] and GROOP [12] represent a scene by a collection of geometric objects each of which contains all its attributes. Therefore, such collections of objects can be passed directly to the renderer. An alternative strategy organizes scene objects in directed acyclic graphs (e.g. OpenInventor [28]). In this display-list oriented approach, scenes are rendered by traversing the graphs. During the traversals the evaluation of attributes is managed by stacks.
Our geometry graphs are based on a declarative, display-list oriented composition technique which offers the following advantages:

- “Intelligent” geometry nodes modify attributes and shapes and can be integrated in the scene description. For example, an adjuster node dynamically modifies the transformation attributes in its subgraph. Systems which restrict the attributes to be declarative do not support the integration of such attribute modifiers into geometry descriptions.
- Declarative scene descriptions come close to the user’s view of 3D scenes because they encode the building process and the image synthesis process of the scene components. The construction of geometry graphs can be directly supported by a graphical user interface.

However, a traversal of the geometry graph is needed to perform geometric operations, e.g. intersection tests, computations of bounding boxes, and rendering. In order to avoid such traversals, MAM/VRS provides a caching mechanism. A cache node is a geometry polynode. Each time a shape node is encountered during the traversal of its child nodes a cache node installs a cache item. A cache item stores optimized, approximated shapes together with all attributes applied to them.
A sample node configuration is shown in Fig. 8, the cache node stores the actual transformation matrices (*T’*, T_3, T_4) and resolves the multiply referenced shape S_2. If the sample geometry graph in Fig. 8 is rendered with an OpenGL rendering device, the *Opt*(S_i) represent OpenGL display lists. Once the cache is built, the node performs geometric operations on the cache items. The cache can be updated automatically. Graphics objects and cache entries are connected to a notification center. If a graphics object has been modified, the cache receives a


MCache
{ { C_1•(T’=T_1•T_2)•Opt(S_1) },
{ T_3•Opt(S_2) },
{ T_4•Opt(S_2) } }
Cache Entries
MAttribute *RColor C_1*
MAttribute *RTransformT_1*
MAttribute *RTransformT_2*
MShape RShape S_1
MGeometryGroup
MAttribute *RTransform T_3*
MAttribute *RTransform T_4*
MShape RShape S_2
Figure 8. Cached Subgraph.

notification. Cache nodes guarantee the principle of locality [31] (i.e. attributes should be placed near the object they modify).

4. BEHAVIOR GRAPHS

A behavior graph is a directed acyclic graph (DAG) which specifies animation and 3D interaction. Behavior graphs and geometry graphs are indirectly related because a geometry node and a behavior node can be associated with the same graphics object. Geometry nodes determine the graphics objects contained in a 3D scene, whereas behavior nodes animate these graphics objects by applying constraints to them.

In general, we distinguish between time-related and event-related behavior nodes, and behavior groups. Time-related behavior nodes calculate and control the time assigned to their child nodes and maintain time-dependent constraints. Event-related behavior nodes respond to incoming events, and maintain event-dependent constraints. In analogy to geometry groups, behavior groups organize child nodes according to a time layout and an event layout.

The geometry graphs and behavior graphs of MAM/VRS applications are managed by *model controllers* which control the rendering and animation of 3D scenes.

4.1 Flow of Time and Events

Explicit modeling of the time and event flow offers several advantages:

- General and reusable behavior components can be designed because their functionality is not part of a specific geometry class.
- Behavior nodes can be combined to model complex animations and interactions because they are treated as first-class objects. The behavior node class hierarchy serves as the base for the development of application-specific behavior classes.
- Behavior graphs can be used to build story books. Behavior group nodes and time modifier nodes calculate and distribute life time intervals to their child nodes. A behavior node can specify an action relative to its life time. An action can be modified by transforming the life time interval of its behavior node. For example, to stretch an action in time, we stretch the life time assigned to its behavior node.
- Behavior nodes can be combined to model user interaction components. For example, a trackball can be realized by an interaction node which responds to mouse events and constrains a rotation graphics objects. Interaction nodes can be nested to form complex interactions. The trackball could be extended, for example, by an additional behavior node which temporarily applies a wire-frame attribute to the shape being rotated.
- Behavior graphs support the object-oriented integration of time- and event-dependent constraints in 3D applications. Behavior nodes automatically install and remove these constraints depending on their status.
- Behavior graphs and geometry graphs can be processed independently. For example, the application can use one thread for controlling the behavior graph and another one for the geometry graph.

4.2 Temporal Abstract Data Types

Most graphics systems define graphical abstract data types such as vectors, volumes, and rasters to simplify the management of geometry. To simplify the management of time, MAM/VRS defines the following temporal abstract data types:

- A *model time* represents a point in time measured in milliseconds.
- A *moment* **M** represents a point **t** in a time interval $\mathbf{[t_0, t_1]}$. A moment assigned to a behavior node determines the node's life time interval and the current point in time within this interval. Moments are essential for behavior nodes which specify animation processes. Based on

the knowledge about their life time, behavior nodes can plan and distribute their activity. Moments provide a local model time for behavior nodes. Fig. 9 shows a sample moment. Its current time is 50 seconds. This point in time belongs to the time interval [10, 90] (in seconds).

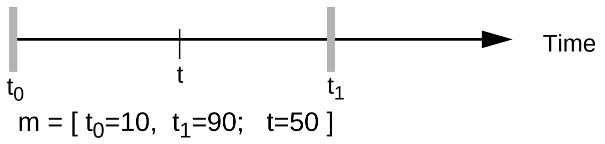


Figure 9. Time Moment.

- A *time requirement* describes the time demand of a behavior node. It consists of the natural (i.e. desired, optimal) duration $\mathbf{T_{natural}}$, the minimal duration $\mathbf{T_{min}}$, and the maximal duration $\mathbf{T_{max}}$ (Fig. 10). A time requirement can specify an infinite natural duration. Furthermore, it may define an alignment **A** which is used to position a shorter moment within a longer moment. With **A** = 0.0 the shorter moment starts at the same time as the longer moment, **A** = 1.0 causes both moments to end at the same time, and **A** = 0.5 centers the shorter moment within the longer moment.

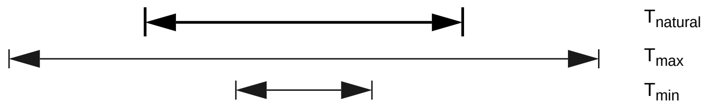


Figure 10. Time Requirement.

- A *time run* is a process which sends synchronization events to a target behavior node during a given moment and at a given frequency. It is typically implemented as a separate thread. The time runs created by behavior nodes are managed by the model controller. Fig. 11 shows a time run for the moment of Fig. 9. It sends synchronization events at a frequency of 1 event / 10 seconds to the associated target behavior node.

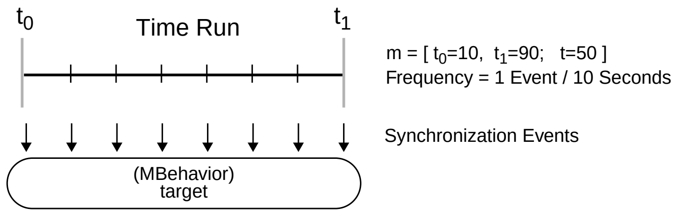

t_0
Time Run
t_1
m = [t_0=10, t_1=90; t=50]
Frequency = 1 Event / 10 Seconds
Synchronization Events
(MBehavior)
target

Figure 11. Time Run.

Due to the lightweight design, behavior nodes do not include time requirements by default. We add these requirements by including specialized mononodes in the behavior graph, called *time setters*, or calculate them implicitly through *behavior groups*. The local model time of a behavior node can be modified by *time modifiers*.

4.3 Time Setters

Time requirements are specified by *MTimeSetter* behavior mononodes. A time setter specifies or modifies the time requirements of its body. A time setter class redefines the *synchronize* method of behavior nodes. MAM/VRS defines the following time setter classes (Fig. 12):

- *MDuration* defines the time requirement of its body by a *TimeRequirement*.
- *MFlexibleDuration* redefines the time requirement of its body by adding time stretchability and shrinkability. A time stretchability and shrinkability of zero fixes the time requirement of a behavior node.

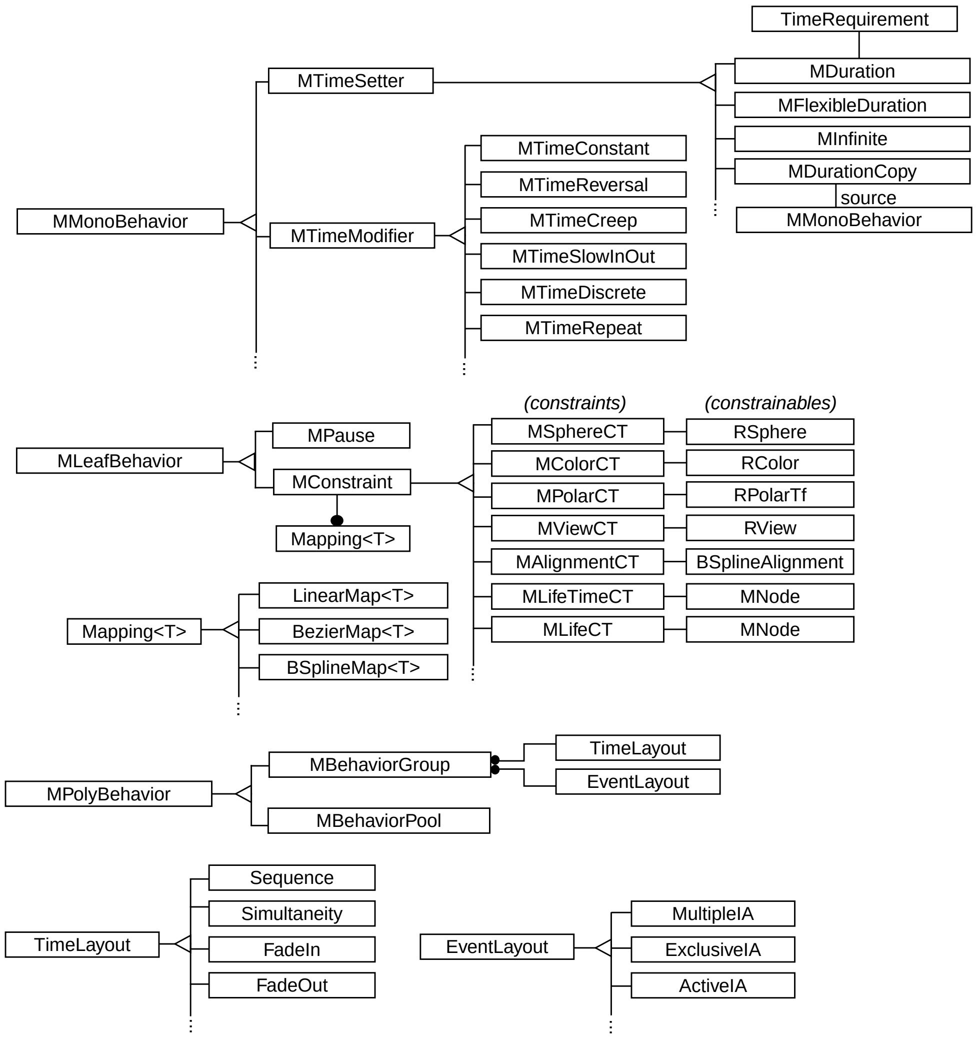


Figure 12. Behavior Node Casses.

- *MInfinite* defines an infinite duration as time requirement of its body. These time setters are used to model permanent actions.
- *MDurationCopy* uses the time requirement of another behavior node as time requirement of its body.

Fig. 13 illustrates the usage of time setters. The time setter $\mathbf{D_1}$ specifies a natural duration of 10 seconds for the behavior node $\mathbf{B_1}$ (Fig. 13a). The duration is fixed, i.e. it cannot be stretched or shrunk. $\mathbf{D_2}$ speficies a natural duration of 20 seconds for $\mathbf{B_2}$. The maximal duration assigned to $\mathbf{B_2}$ can be (20+10) = 30 seconds, the minimal duration (20-5) = 15 seconds (Fig. 13b). The *MFlexibleDuration* node (Fig. 13c) overwrites the time stretchability of $\mathbf{D_{3b}}$, i.e. the maximal duration assigned to $\mathbf{B_3}$ can be (10+30) = 40 seconds. The *MDurationCopy* node $\mathbf{D_4}$ (Fig. 13d) uses for $\mathbf{B_4}$ the time requirement defined in $\mathbf{D_2}$.

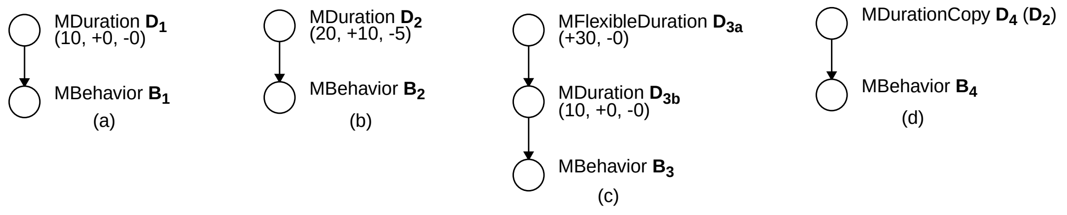


Figure 13. Time Setter Examples.

4.4 Time Modifiers

Time modifiers are used to transform the local model time of behavior nodes. A *MTimeModifier* node (Fig. 12) defines a time-to-time mapping which is applied to all moments passed through the time modifier. MAM/VRS defines the following time modifier classes:

- *MTimeConstant* assigns a constant time to its body (Fig. 14a).
- *MTimeReversal* inverts the direction of the time progress for the body (Fig. 14b). Time reversal nodes are useful to model retrograde actions.
- *MTimeCreep* defines a creeping time progress, i.e. the time progress is slow in the beginning and fast at the end of a time interval (Fig. 14c). Alternatively, the time progress can be fast in the beginning and slow at the end. The creeping is controlled by a speed coefficient. This time modifier is used to model animation processes which begin or end slowly.
- *MTimeSlowInOut* defines a time progress which is slow in the beginning and at the end of a time interval (Fig. 14d). The slow-in and slow-out speed is controlled by speed coefficients.
- *MTimeDiscrete* defines a discontinuous time progress. A time interval is decomposed in piecewise constant time intervals (Fig. 14e). This time modifier is used to model time “jumps”. *MTimeDiscrete* provides coefficients which specify the number of intervals and the time increments.
- *MTimeRepeat* maps a moment modulo a time interval, and passes the resulting moment to its body (Fig. 14f). For example, to model an action which lasts 5 seconds and which should be repeated permanently, we specify an infinite duration followed by a time repeater with the modulo moment [0, 5sec] (Fig. 14g).

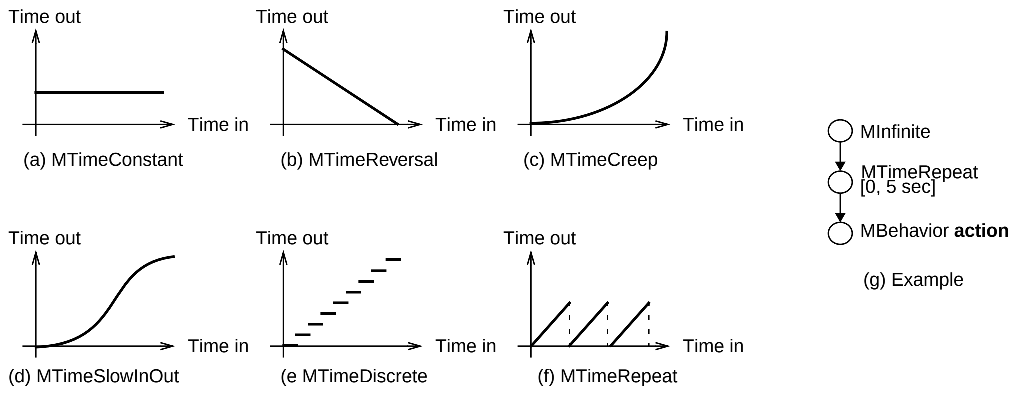


Figure 14. Time Functions Used by Time Modifiers (a-f). Behavior Graph for a Permanent Action (g).

4.5 Behavior Groups

Behavior groups provide an automatic time negotiation mechanism which allows the developer to specify animation processes at a high level of abstraction. The time negotiation is based on time layouts and event layouts. Behavior groups are implemented as behavior polynodes which associate a time layout and an event layout (Fig. 12).

4.5.1 Time Layouts

A time layout calculates the individual life times of the associated behavior group's child nodes based on their time requirements and the layout strategy. If a behavior group receives a synchronization event, the time layout checks which child nodes have to be activated or deactivated. It synchronizes all active child nodes to the new time, and assigns the calculated moments to them.

Instead of the global system time, moments are passed to behavior nodes. Since time layouts provide mechanisms to distribute and to align time intervals, the designer is relieved from calculating absolute times. Based on a relative model time, we can specify a wide class of time-dependent behaviors at a high level of abstraction, e.g. time layouts can be used to design animation processes like in story books. Examples for time layouts (Fig. 12) are:

- *Sequence*: Defines the total time requirements as sum of the time requirements of the child nodes. It distributes a received moment proportionally to the child nodes. The moments assigned to the child nodes are sequential and disjoint. Only one child node is alive at any given time during the duration of the sequence.
- *Simultaneity*: Defines the total time requirement as the maximum of the time requirements of the child nodes. It distributes a received moment to the child nodes if their natural duration is equal to the duration of the moment. If not, the simultaneity layout tries to shrink or stretch the time requirements of the child nodes to fit the duration. If they still do not match it aligns the duration of the child nodes within the moment.
- *FadeIn* and *FadeOut*: Like sequences, they define the total time requirement as the sum of the time requirements of the child nodes. *FadeIn* layouts assign moments to their child nodes. Child nodes are activated like in the case of the sequence layout, but all child nodes remain active until the life time of the last child node expires. *FadeOut* layouts are reversed *FadeIn* layouts.

Fig. 15 shows how actions can be composed by behavior groups, and how time requirements are evaluated. The behavior nodes A_1, A_2, and S_i are processed sequentially. Behavior node S_i consists of two simultaneous behavior nodes, A_{3a} and A_{3b}. D_1, D_2, and D_3 define infinitely stretchable time requirements of 1, 2, and 1 seconds. The sequence S_e is prefixed with a duration D of 100 seconds. If S_e actually gets from its parent 100 seconds, it distributes this moment proportionally to its child nodes, i.e. A_1 and S_i get 25 seconds each, and A_2 50 seconds. Since A_{3a} can last at most (1+14) = 15 seconds, S_i centers the life time of A_{3a} within the 25 seconds. S_e activates in turn A_1, A_2 and S_i, A_{3a} and A_{3b} are activated by S_i.

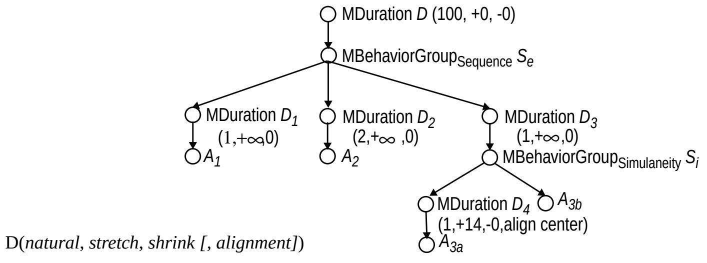

MDuration D (100, +0, -0)
MBehaviorGroup_Sequence S_e
MDuration D_1 (1,+∞,0)
A_1
MDuration D_2 (2,+∞ ,0)
A_2
MDuration D_3 (1,+∞,0)
MBehaviorGroup_Simulaneity S_i
MDuration D_4 (1,+14,-0,align center)
A_3a
A_3b
D(natural, stretch, shrink [, alignment])

Figure 15. Time Management through Time Layouts.

4.5.2 Event Layouts

Event layouts (Fig. 12) determine how events are dispatched to child nodes. For example, the *MultipleIA* event layout dispatches an event to all child nodes, whereas an *ExclusiveIA* event layout dispatches an event to child nodes until one child node has consumed the event. The *ActiveIA* distributes events to those child nodes which are activated. Event layouts in behavior groups can be used to specify complex interactions and multi-state augmented transition networks [10].

4.6 Behavior Pools

A behavior pool which is implemented as a behavior polynode (Fig. 12) collects and controls behavior subgraphs. The child nodes can be activated and deactivated individually. If a child node is activated, the behavior pool requests its time requirements and registers a time run for that node in the model controller. The synchronization events created by the time run are dispatched to the child node. If a child node is deactivated, the behavior pool deregisters the time run.

A behavior pool maintains for each of its child nodes a list of start actions and stop actions. An action is an object which performs operations (e.g. callbacks, method invocations). The start actions for a child node are executed when it is activated, and the stop actions are executed when it is deactivated. Examples for actions used in behavior pools are the *MSwitchOn* action which invokes the activation of a child node, and the *MSwitchOff* action which causes the deactivation of a child node.

4.7 Constraints

Constraints are implemented as behavior leafnodes (Fig. 12). They can constrain geometry nodes, behavior nodes, and graphics objects. A constraint node establishes its constraints at the beginning of its life time, and removes the constraints when its life time ends. We distinguish between one-way and multi-way constraints. One-way constraints know their solution strategy, whereas multi-way constraints are solved by an external constraint solver, e.g. SkyBlue [25]. Constraint networks are anchored in behavior graphs and connected to the flow of time and events by behavior nodes.

4.7.1 Mappings

In general, one-way constraint nodes for graphics objects apply time-dependent functions to parameters of graphics objects. MAM/VRS encapsulates these functions in mapping objects. A *mapping* represents a function which maps a given moment to a value (e.g. float, 2D point, 3D point). If the constraint node obtains a synchronization event, the mapping calculates the new value and applies it to the graphics objects.

- *LinearMap* maps the time interval specified by a moment onto a polyline. The polyline is defined by an ordered sequence of control points. For each moment, the mapping calculates and returns the corresponding point on the polyline.
- *BezierMap* maps the time interval specified by a moment onto a Bezier curve. The Bezier curve is defined by an ordered sequence of control points. For each moment, the mapping calculates and returns the corresponding point on the curve.
- *BSplineMap* maps the time interval specified by a moment onto a B-spline curve. The B-spline curve is defined by an ordered set of control points, a knot vector, and the curve degree. The mapping contains an interval B-spline segment cache which improves the calculation of B-spline points for a given moment.

The mapping classes are generic. For example, a *LinearMap<Vector>* mapping interpolates a set of vectors during the time interval defined by the moment.

4.7.2 Constraints for Graphics Objects

Constraint nodes can describe time-varying parameters of graphics objects. For each of the time-varying parameters, constraint nodes require a mapping object. Basically, a constraint node class defines three operations: constraint installation, constraint application, and constraint removal.

The constraint installation takes place when the constraint node is activated. Whenever the constraint node receives a synchronization event, it calculates the new parameter values by parameter mappings, and assigns the new parameter values to its constrained graphics object. In addition, it may notify the model controller about the modification of the graphics object in order to cause a redrawing of the scene. The constraint is removed when the constraint node is deactivated.

Optionally, the constraint node can restore the original state of graphics objects before the constraint node was activated. In this case, the constraint node requests and stores the original parameter values before the installation, and restores these values after its deactivation.

4.7.3 Constraints for Nodes

Constraints can control the activation status of nodes. A *MLifeTimeCT* node ensures that a target node is active during a given time interval. A *MLifeCT* node ensures that a target node is activated (respectively deactivated) when the *MLifeCT* node is activated (respectively deactivated).

For example, to turn on an actor in a 3D scene during an animation, we specify a simultaneous behavior group which consists of the animation behavior, and a *MLifeCt* which keeps the actor’s geometry node activated during its own lifetime.

4.8 Animating the Example Algorithm

The Voronoi algorithm is animated as follows: The paraboloid appears in the scene, the points are lifted onto the paraboloid, the tangential planes are enlarged, and the paraboloid disappears. Simultaneously the camera moves such that finally the Voronoi diagram becomes visible by looking into the open paraboloid. We revert the animation by moving the camera back to its original position, shrinking the planes and fading in the paraboloid. Optionally, the title should fly through the scene.

The behavior graph of the algorithm animation is presented in Fig. 16; the corresponding geometry graph is shown in Fig. 4. The animation’s behavior pool consists of the Voronoi construction, the reverse animation, and the flight of the title. Basically, these behavior nodes constrain the graphics objects which are also associated with the geometry graph.

For example, the *MViewCT* nodes constrain the look-from point of the *RView* graphics object ***C***. The mappings of these *MViewCT* nodes are *BSplineMap* objects which define the camera path taken during the life times of the *MViewCt* nodes. The first *MViewCt* is preceded by a *MTimeCreep* node. Therefore, the camera moves slowly in the beginning and fast at the end of its animation. Optionally, we could constrain the look-to point of ***C*** by providing an additional mapping. The *MTransmissionCT* nodes constrain associated *RTransmission* graphics objects. In general, their mappings interpolate the surface transmission coefficients in the range [0.0, 1.0] during their life times.

The **ReverseAnimation** behavior reuses the **EnlargePlanes** and **LiftPoints** behavior subgraphs. The sequential behavior group is preceded by a time reversal node, i.e. the time progress is inverted.

The *MLifeCT* in the **TitleFlight** behavior activates the associated geometry group ***H*** which contains the 3D characters for the title. Only activated geometry nodes are considered for rendering. The *MAlignmentCT* constrains the B-spline interval to which the child nodes of the

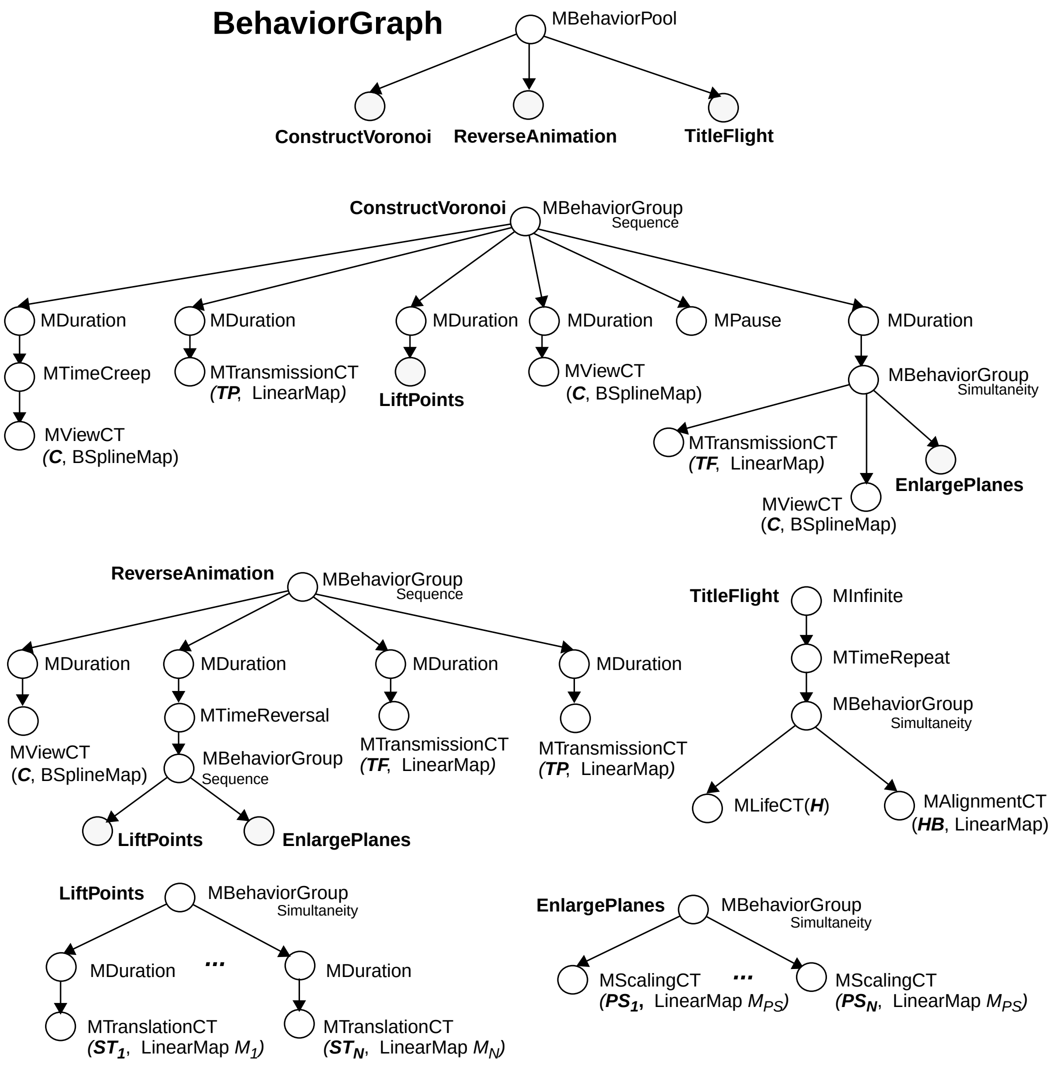


Figure 16. Behavior Graph for the Algorithm Animation.

geometry group are aligned, i.e. the characters are moved along the B-spline. Its mapping defines the section of the B-spline curve to which the characters are aligned.
The constraint relations are established (respectively removed) automatically when the behavior nodes are activated (respectively deactivated) by the behavior pool. Behavior nodes have to be activated by the application. Optionally, we could supply an interaction behavior which switches a behavior node of the behavior pool on or off according to the user’s interaction.

4.9 3D Interaction

Several research tools have been developed which explore new interaction techniques and interface styles. However, they are limited with respect to robustness, completeness, and portability [11]. In traditional 3D toolkits, interaction techniques are provided as black boxes.

Therefore, the developer cannot reuse or extend these components. Furthermore, it is difficult to combine animation components and interaction components.
The MAM/VRS concept for interactions is based on an object-oriented decomposition of their functional components such as events, conditions, and interaction processes, which are discussed in the following sections.

4.9.1 Events

MAM/VRS extends the concept of events to a general notification mechanism between behavior nodes. Low-level hardware events represent one category of events which are organized similar to the X11 event types in an object-oriented fashion. Another category of events is used for the node-to-node communication. These events are typically created by nodes or the model controller (e.g. time events are created by time runs, collision events are created by geometry nodes). Behavior nodes are responsible for dispatching events.

4.9.2 Event Conditions

A behavior node has to trigger actions if it receives the proper events. We could encode the event checking procedure in the behavior node's methods. However, this would restrict the reusability of behavior nodes since event conditions are application-specific, e.g. a shape rotation may be controlled by mouse movements but also by key events. Therefore, we provide objects (Fig. 17) which test whether an event satisfies a condition. When behavior nodes receive events, they use these event conditions to decide if the event is of interest.
For example, the *ShapeSelection* condition is satisfied by mouse click events which pick a shape in the scene, whereas the *BackgroundSelection* condition is satisfied if no shape in the scene is selected. The *ConditionGroup* combines event conditions by logical operations (e.g. and, or, not).

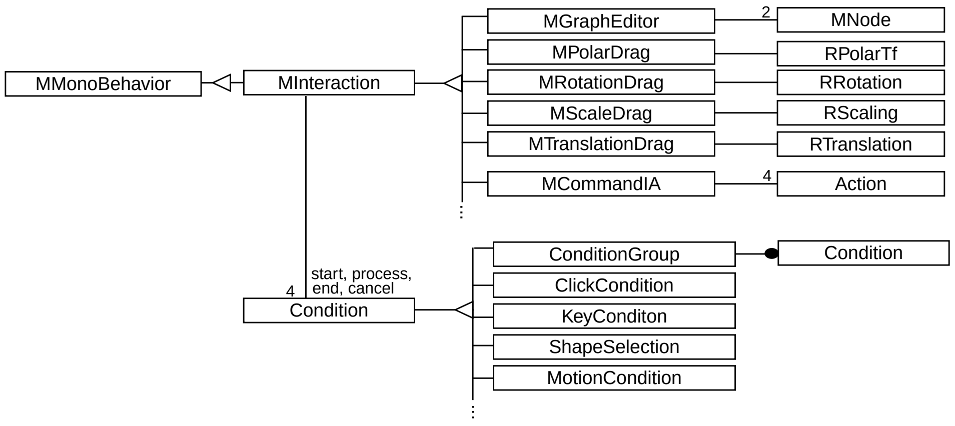


Figure 17. Event Condition and Interaction Classes.

4.9.3 Interactions

Interaction nodes are behavior mononodes which control an interaction behavior (Fig. 17). A *MInteraction* node specifies four methods: *start*, *process*, *end*, and *cancel*. These methods are called if the interaction node receives events which satisfy the corresponding event conditions, i.e. the *start* condition, *process* condition, *end* condition, and *cancel* condition.
Interaction nodes can be linked together to build complex interactions. If an interaction node starts, it sends a *start* interaction event to its body. If there is another interaction node in the body, it receives this event and starts, too. In the same way, an interaction node triggers the methods for *process*, *end*, and *cancel* in its subgraph. Because interaction nodes use their own

event types for communication, other behavior nodes can be inserted between two interaction nodes. MAM/VRS defines the following basic interaction nodes:

- *Graph-editor nodes* temporarily modify the graph structure by inserting or removing graph nodes. For example, to impose a wire-frame style on a shape during an interactive rotation, a graph-editor node can prepend an attribute node to a shape node when the interaction starts; this attribute node is removed when the interaction ends.
- *Drag nodes* map time events or device events to numerical intervals. The values are used to constrain parameters of graphics objects. The drag nodes include trackballs (associated with a polar or a rotation transformation matrix), translation draggers (associated with a translation matrix), and scale draggers (associated with two points in space which are interpolated).
- *Command nodes* are used to integrate application-specific callbacks in the behavior graph. The *MCommandIA* node executes an action if a given event condition is satisfied, and it is used to infiltrate actions in interaction processes.

4.10 Model Controller

Model controllers are used by application frameworks, they control the rendering and animation of 3D scenes. Model controllers are associated with the root nodes of the geometry graphs and the behavior graphs (Fig. 18). They install time runs which synchronize behavior nodes, dispatch external events (e.g. user input) to behavior nodes, and manage the redrawing of scenes. MAM/VRS provides specialized model controllers for Motif and [incr Tcl]/[incr Tk]. Controllers are toolkit-specific because the redrawing of scenes and the management of time runs depend on the underlying window system.

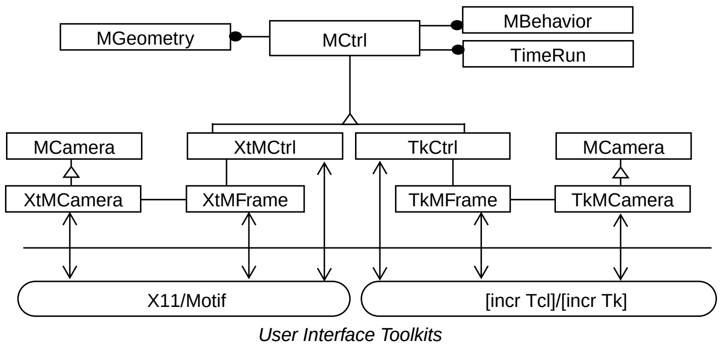


Figure 18. Model controller and its class relations.

The Motif model controller *XtMCtrl* is associated with a Motif application framework defined by *XtMFrame*. This framework provides a graphical user interface with standard menus. It installs Motif cameras defined by *XtMCamera*. Cameras are toolkit-specific since they must map events of the underlying window system to MAM/VRS events. In analogy to Motif, MAM/VRS designs a framework for [incr Tcl]/[incr Tk].

A MAM/VRS application always instantiates a framework which implicitly creates a default camera, a default controller and a default behavior pool. The application links its scenes to the camera and its behavior graphs to the default behavior pool. Finally, it activates the user interface.

5. 3D WIDGETS

One of the key goals of 3D animation toolkits is to provide a rich collection of interactive 3D widgets [32]. Conner et al. [5] define a *3D widget* as “an encapsulation of geometry and behavior used to control or display information about application objects". 2D user interface toolkits provide a universal collection of 2D widgets which can be used for many application domains. Although metaphors and paradigms for 3D interaction (e.g. the rack to perform high-level deformations [5] and the cone tree to view hierarchical information [23]) have been investigated no generic 3D toolkit is available which provides universal 3D widgets.

5.1 Structure of 3D Widgets

3D widgets represent high-level, interactive, animated 3D components. They construct internal geometry graphs and behavior graphs, and allow the developer to perform operations on these graphs through a high-level widget interface. This interface hides much of the complexity of the node and graph construction. Only a few of the internal geometry nodes and behavior nodes are visible from outside the widget.

Ports determine how widgets can be linked together. For each of its visible graphics objects or nodes, and for each graphics object or node supplied to the widget from outside, a 3D widget defines a port. A port specification includes the classes of the involved objects or nodes, the number of objects, and the read/write permissions, i.e. whether an object is imported (or exported) as read-only or readable/writable object. Fig. 19 shows the symbolic notation for port specifications.

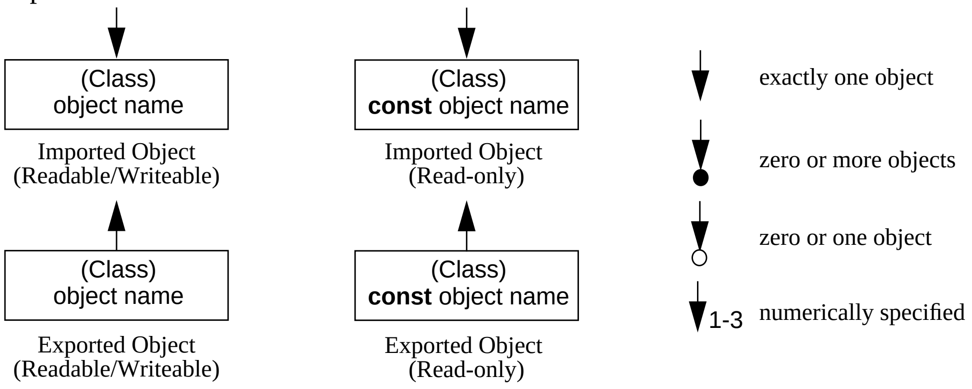


Figure 19. Notation for Port Specifications.

3D widgets simplify the application construction because developers are not directly concerned with building geometry graphs and behavior graphs. Since nodes provide a transparent implementation structure and can be combined in a flexible way, this leads to highly configurable widgets and allows the developer a straightforward implementation of application-specific widgets. In order to build a 3D application, the developer only has to instantiate 3D widgets and connect their ports.

5.2 A Sample Trackball Widget

As an example for high-level interaction and animation components, we develop the trackball widget. The trackball widget is used to rotate a shape interactively. A trackball widget imports a shape node **F**, a polar transformation graphics object **B**, and the attribute graphics object **W.** The attribute node S associated with **W** is prepended to **F** during the interaction (Fig. 20a). The widget exports a behavior subgraph composed of a polar drag node **T** and a graph-editor node. The *MPolarDrag* node maps the mouse movement to a 3D rotation which is assigned to the polar transformation **B**.

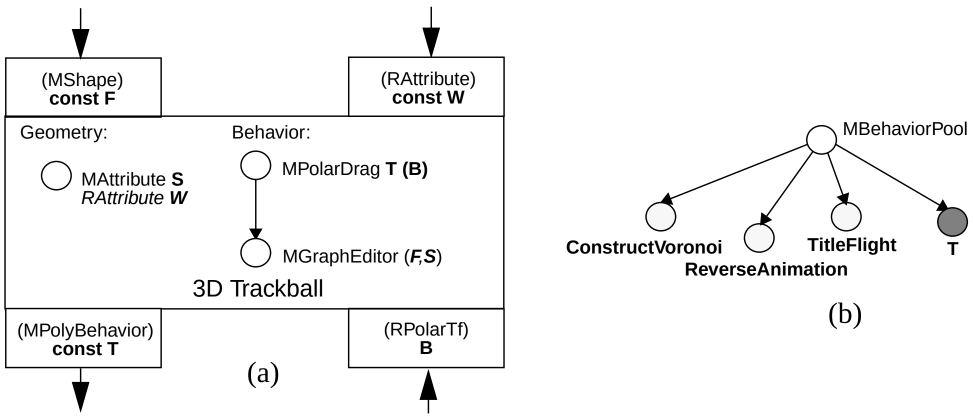


Figure 20. Trackball Widget (a). Extended Behavior Graph (b).

The trackball widget has to be connected to the application's behavior graph. In Fig. 20b a trackball behavior is connected to the behavior pool of our sample animation. The widget is associated with the *RTransform* node ***B*** and the shape node ***F*** (Fig. 4).

The start condition for the *MPolarDrag* node can be specified as '*Click(Button$_1$ down) and (ShapeSelection(F))*'. These conditions are modeled by condition groups. The end condition is specified as '*Click(Button$_{any}$ up)*', the process condition as '*Motion*', and the cancel condition as '*Click(Button$_2$ down)*'.

5.3 3D Widget Overview

MAM/VRS defines several standard 3D widgets.

- *WShapeTransformer*: A shape transformer widget allows the user to rotate, scale, and translate a shape. It uses small three-dimensional glyphs located around a bounding box of the shape. The glyphs are used like handles to perform the different transformations (Fig. 21a).
- *WCameraCockpit*: A camera cockpit widget consists of a complex geometry graph which specifies the cockpit's instruments (built by the MAM/VRS solid modeler), and a complex behavior graph which interprets the movements of the steering-wheel (Fig. 21b). These behavior nodes constrain the *RView* graphics object associated with the application's *MScene* node. The snapshot in Fig. 21b shows the camera cockpit widget and the shape transformer widget applied to a geometric object.
- *WShapeGroup*: A shape group widget controls the interactive selection from a set of shape nodes. For a selected shape, an attribute node can be installed temporarily (e.g. a brightness node to highlight the current selection). The widget implements a behavior subgraph containing interaction nodes which detect shape selections.
- *WActorCamera*: An actor camera widget provides automatic camera control. It determines the view position and direction of a virtual camera according to the positions and the weights of virtual actors [20].
- *WEnvironment*: An environment widget contains scene elements and installs lights. Additionally, it provides environment elements such as the sky (i.e. a hemisphere enclosing the whole scene) and different terrains.

6. IMPLEMENTATION

The MAM/VRS toolkit is implemented in C++ and consists of more than 120 node classes and 140 classes for graphical abstract data types. Currently, we have implemented virtual rendering devices for OpenGL, PEX, RenderMan, XGL, and Radiance. We provide user interface

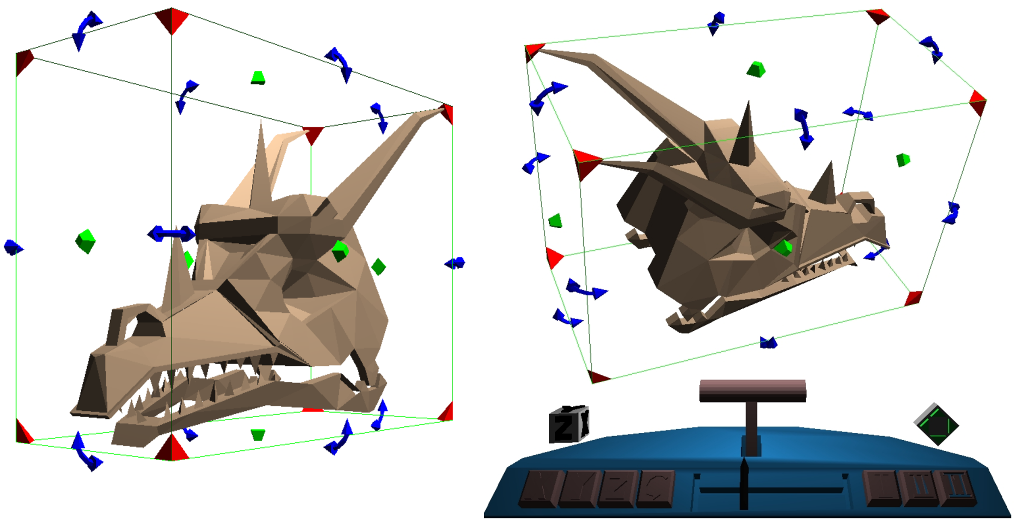


Figure 21. Shape Transformer Widget (a). Combination with the Camera Cockpit Widget (b).

bindings for OSF/Motif and for [incr Tcl]/[incr Tk], an extension of the command tool language Tcl [22].

6.1 Application Programming Interfaces

Application developers can use the C++ class interfaces or the [incr Tcl] class interfaces [16]. We use C++ for the implementation of the core classes due to its efficiency. Rapid prototyping of 3D applications is facilitated by using the interpretative [incr Tcl] language, an object-oriented extension to Tcl/Tk. For example, application-specific behavior classes which in general differ only in the included functional expressions can be represented by one generic [incr Tcl] behavior class which includes functional expressions as parameters evaluated at run-time. Thus, we avoid recompiling, relinking and extensive subclassing.

6.2 Object Management

Dealing with large numbers of objects requires mechanisms to simplify the object handling. MAM/VRS supports object management through an object sharing mechanism, object factories, and object repositories.

6.2.1 Shared Objects

MAM/VRS objects are created dynamically, and can be multiply referenced. Each shared object contains a reference counter. If a shared object is referenced, this counter is incremented. If it is dereferenced, the counter is decremented. A shared object is destroyed if its counter becomes zero.
The template class *SO<T>* provides safe pointer handling for shared objects of type *T*. A *SO<T>* object stores a pointer to a shared object. If a shared object is assigned to this pointer, *SO<T>* references the shared object automatically. *SO<T>* dereferences its shared object before changing this pointer and before its destruction.

6.2.2 Object Factories

Object factories are classes which construct instances of other classes. Object factories simplify the usage of a fine-grained object-oriented system because they hide much of the complexity of class hierarchies and class constructors, and offer methods to produce frequently used graphics objects and node configurations. MAM/VRS groups the node classes and graphical abstract data types in the following factories:

- Shape factory: Creates shape graphics objects.
- Attribute factory: Creates attribute graphics objects.
- Geometry Factory: Creates shape nodes, attribute nodes, scene nodes, and camera objects.
- Behavior Factory: Creates nodes for time management, time modification, interaction, and constraints.
- Text Factory: Creates node configurations which represent 3D text objects.
- Transfer Factory: Converts standard 3D file formats into MAM/VRS node configurations.

The shape and attribute factories are merged in the *VRS* factory, and the other factories are merged in the *MAM* factory by multiple inheritance. To illustrate the object construction through factories, the example below shows part of the *ShapeFactory*:

```
class ShapeFactory : public Factory {
public:
 SO<RSphere> sphere(float r, Vector p) { return new RSphere(r,p);}
 SO<RBox> box(Vector minpoint, Vector maxpoint) { return new RBox(minpoint, maxpoint); }
 ...
};
```

The *GeometryFactory* provides generic methods to construct geometry nodes. They associate these nodes with existing graphics objects or create implicitly new graphics objects:

```
class GeometryFactory : public Factory {
public:
 // generic method for all shapes:
 SO<MShape> shape(RShape* s) { return new MShape(s); }

 // methods with implicit graphics objects:
 SO<MShape> sphere(float rad, Vector center) {
   return new MShape(new RSphere(rad, center));
 }
 SO<MShape> box(Vector minpoint, Vector maxpoint) {
  return new MShape(new RBox(minpoint,maxpoint));
 }
 ...
 // generic method for all attributes:
 SO<MAttribute> attr(RAttribute* a) {
   return new MAttribute(a);
 }

 // methods with implicit graphics objects:
 SO<MAttribute> color(float r, float g, float b) {
   return new MAttribute(new RColor(r,g,b));
 }
 ...
};
```

6.2.3 Object Repository

An object repository is a persistent description of nodes, graphics objects, and their relations. These repositories are similar to container objects which automatically reconstruct objects and object relations from a persistent representation. Object repositories can be used to build libraries for nodes and node configurations.

All MAM/VRS classes define persistency methods, i.e. methods to write the object’s state to files and to read it back from files. Since each class is registered in an object factory, an object repository uses the factories to determine whether a persistent object can be reconstructed.

6.3 C++ Implementation of the Algorithm Animation

The geometry graph and the behavior graph developed in Fig. 4 and Fig. 16 can be implemented with the C++ API as shown below. First, we define the application class *Voronoi* which is inherited from the Motif application frame *XtMFrame*. The *Voronoi* class specifies as data members the nodes and graphics objects[3)] which we will use in the geometry graph and in the behavior graph:

```
class Voronoi : public XtMFrame {
public:
 Voronoi(int argc, char** argv);
private:
 SO<RView> C;
 SO<RPolarTf> B;
 SO<MShape> F;
 SO<RTransmission> TF;
 SO<RTransmission> TP;
 Array< SO<RTranslation> > ST;
 Array< SO<RTranslation> > PT;
 Array< SO<RScaling> > PS;
 SO<MScene> Scene;
 SO<MGeometryGroup> H;
 SO<BSplineAlignment> HB;
 SO<WTrackBall> trackball;
};
```

The *Voronoi* constructor builds the graphics objects and nodes, omitted parameters are denoted by '*(...)*':

```
Voronoi::Voronoi(int argc, char** argv) : XtMFrame(argc, argv) {
 MAM mam;
 VRS vrs;

 // construct graphics objects
 C = vrs.view(...);
 B = vrs.polar_tf(...);
 TF = vrs.transmission(0.0);
 TP = vrs.transmission(0.0);

 // build geometry and behavior for the points and their tangential planes
 SO<MGeometryGroup> point_group = mam.geometry_group();
 SO<MGeometryGroup> plane_group = mam.geometry_group();
 SO<MBehaviorGroup> lift_points = mam.simultaneity();
 SO<MBehaviorGroup> enlarge_planes = mam.simultaneity();
 SO<MShape> point_glyph = mam.shape(vrs.sphere(...));

 // read point data
 for each point (i=0, ..., N-1):
    ST[i] = vrs.translation(...);
    PT[i] = vrs.translation(...); PS[i] = vrs.scaling(...);
    point_group->append_child(mam.attr(ST[i])->set_body(point_glyph));
    plane_group->append_child(mam.attr(PT[i])->set_body(
      mam.attr(PS[i])->set_body(vrs.facet(...)))
    );
    lift_points->append_child(mam.duration(...)->set_body(
      mam.translation_ct(ST[i], mam.linear_map(...))
    ));
    enlarge_planes->append_child(mam.scaling_ct(PS[i], mam.linear_map(...)));

 HB = mam.bspline_align(...);
 H = mam.aligned_text("VORONOI", HB)
}
```

Now we build the geometry graph:

3.The names of the objects correspond to the names used in the figures 4, 16, and 20.

```
SO<MScene> scene = mam.scene(
  mam.geometry_group(
    mam.attr(vrs.ambientlight(...)),
    mam.attr(vrs.distantlight(...))
  ),

  mam.adjuster(...)->set_body(
   mam.attr(B)->set_body(
    mam.geometry_group(
     mam.attr(TF)->set_body(
      mam.attr(vrs.gray(0.7))->set_body(
       F = mam.shape(vrs.box(...))
      )
     ),
     ... install paraboloid ...,
     mam.attr(vrs.plastic(...))->set_body(
      mam.attr(vrs.rgb(1,0,0))->set_body(
       point_group
      )
     ),
     ... install plane_group ...
    )
   )
  ),
  mam.attr(vrs.plastic(...))->set_body(H)
);
```

Next, we associate a XGL camera and a Radiance camera with the scene. They are provided by the application frame work and can be referenced by the methods 'xgl_camera' and 'radiance_camera':

```
xgl_camera()->append_scene(scene);
radiance_camera()->append_scene(scene);
```

The behavior graph (Fig. 16) for the animation can be constructed as follows:

```
SO<MBehaviorGroup> construct_voronoi = mam.sequence(
  mam.duration(...)->set_body(
   mam.slowinout(...)->set_body(
    mam.view_ct(C, mam.bspline_map(...))
   )
  ),
  mam.duration(...)->set_body(mam.transmission_ct(TP, mam.linear_map(1.0,0.0))),
  mam.duration(...)->set_body(lift_points),
  ... install other behavior nodes ...
);

... compose reverse_animation, title_flight analogously

trackball = mam.trackball(F,B, vrs.wire_frame());
```

Finally we connect the application's behaviors to the behavior pool of the application:

```
behavior_pool()->append_child(construct_voronoi);
behavior_pool()->append_child(reverse_animation);
behavior_pool()->append_child(title_flight);
behavior_pool()->append_child(trackball->get_behavior());

// activate first behavior:
behavior_pool()->switch_on(0);
// trigger second behavior when the first behavior stops
behavior_pool()->append_stop_trigger(0, mam.switch_on(behavior_pool(), 1));

// add controls (e.g. menu) for (de)activating the title flight behavior
}; // end of constructor
```

The main program constructs one instance of *Voronoi* and calls its run method which forces the model controller to initialize the geometry graph and behavior graph:

```
int main(int argc, char** argv) {
  Voronoi V(argc,argv); V.run();
}
```

7. RELATED WORK

In the last few years several object oriented 3D graphics and animation toolkits have been proposed, e.g. Grams [7], OpenInventor [28], GROOP [12], SWAMP [2], TBAG [26], UGA [33], and Obliq-3d [19].

Similar to our approach the architecture of Grams is separated into a rendering layer, a graphics layer, and an application layer. Grams appears to be one of the first systems providing rendering portability by strictly separating the rendering and the graphics layer. However, application objects have to be converted to objects of the graphics layer, which again have to be converted to objects of the rendering layer. The conversion involves a computational and a storage overhead, and the semantics of application-specific types is lost. Grams' architecture does not focus on animation and interaction.

GROOP is an object oriented graphics toolkit based on the camera/stage/actor paradigm. The class hierarchy stresses on user-oriented organization of 3D objects which significantly increases the level of abstraction at which 3D models and animations can be specified. Like Grams, the criterion used for subclassing is the common internal representation of shapes. This leads to an implementation-dependent class hierarchy which is not portable between different rendering-techniques. Both, GROOP and Grams do not use fine-grained object orientation to such an extend as our approach does.

SWAMP is an object-oriented graphics environment and stream-based animation system. Streams generalize animation control strategies such as keyframing and script languages. SWAMP appears to be one of the first systems which explicitly decouples geometry descriptions from behavior descriptions in an object-oriented fashion. MAM/VRS extends the stream concept to a fine-grained object-oriented organization of behavior.

OpenInventor has introduced a powerful node-based and graph-based construction paradigm for 3D applications which is extended by our approach to behavioral modeling. OpenInventor scene graphs rely on OpenGL's rendering pipeline. Nodes are highly order-dependent and rely on side-effects of sibling nodes. In our toolkit, the structural properties of nodes overcome this problem. OpenInventor attributes and shapes are not portable between different rendering techniques and rendering libraries. Image synthesis processes are not represented by nodes. Behavioral aspects of 3D models are integrated by procedural extensions to nodes, but are not modeled explicitly through behavior classes. We believe, however, that an object-oriented approach to behavior modeling can dramatically reduce its complexity like object orientation does for geometric modeling. OpenInventor, like Grams and GROOP, does not provide temporal abstract data types. Behavioral aspects are linked by callbacks to geometry nodes. In our toolkit we use behavior graphs to model behavior. Behavior graphs provide a clear and comprehensible organizational structure for time- and event-dependent animation and interaction. In contrast to our approach in which constraints are represented by behavior nodes, OpenInventor does not provide a constraint management.

The object oriented animation system Obliq-3D represents scenes by graphical objects which are related to time-varying properties and callbacks. Obliq-3D's class hierarchy (like for Grams and GROOP) models only graphical objects. However, it does not provide the flexibility given by our constrainable and shareable graphical abstract data types and behavior classes. Obliq-3D's behavior components operate on low-level attributes. Obliq-3D does not

support a high-level organization of the time and event flow as our approach does with behavior groups based on a time and event layout.
Other related systems are TBAG and UGA. TBAG is based on a functional approach. Similar to our approach, TBAG integrates constraints and graphical abstract data types as first-class objects. UGA is one of the first systems with a close integration of geometry and animation. Our approach has been motivated by similar goals, and concentrates more on an object-oriented design of geometry and behavior.

8. CONCLUSIONS AND FUTURE WORK

MAM/VRS as an object-oriented framework for 3D modeling, animation and interaction supports behavioral modeling at the same level of abstraction as geometric modeling. It separates geometry nodes from behavior nodes, provides for both a wide range of node classes, and uses constraints to relate them. Geometry graphs represent all graphical aspects of 3D scenes including shape and attribute nodes which visualize and evaluate instances of graphical abstract data types, geometry groups, and image controllers. Behavior graphs explicitly organize the flow of time and events through time modifiers, behavior groups and constraint nodes. Time negotiations and temporal abstract data types allow the developer to specify complex animation processes at a high level. Interaction nodes and event conditions represent a set of elementary toolkit components which can be easily combined to 3D widgets. The polymorphic nodes of our toolkit facilitate the construction of animated, interactive 3D applications through a fine-grained object-oriented framework. The semantic-oriented class hierarchies guarantee portability and reusability. Due to the C++ and [incr Tcl]/[incr Tk] application programming interfaces, the toolkit can be integrated in an interactive environment for the development of 3D applications.
Future directions of our work include the automation of the camera control, the integration of more animation techniques such as physically-based animation, and the addition of high-level behavior nodes for the control of articulated figures.

REFERENCES

1. AVS, Inc., ‘AVS Developer’s Guide’, V. 3.0, 1991
2. M. P. Baker, ‘An Object-Oriented Approach to Animation Control’. in *Computer Graphics Using Object-Oriented Programming*, S. Cunningham et al. (eds.), Wiley & Sons, 187-212, 1992.
3. E. Bier, M. Stone, K. Pier, W. Buxton, T. DeRose. ‘Toolglasses and magic lenses: the see-through interface’. *Proceedings of SIGGRAPH’93*, 73-80 (1993).
4. P. R. Calder, M. A. Linton, ‘Glyphs: Flyweight Objects for User Interfaces’, *Proceedings of the ACM SIGGRAPH Third Annual Symposium on User Interface Software and Technology*, 1990.
5. D. B. Conner, S. S. Snibbe, K. P. Herndon, D. C. Robbins, R. C. Zeleznik, A. van Dam, ‘Three-Dimensional Widgets’. *Computer Graphics* (1992 Symposium on Interactive 3D Graphics), 25, (2), 183-188 (1992).
6. A. Tal, D. Dobkin, ‘Visualization of Geometric Algorithms’, *IEEE Transactions on Visualization and Computer Graphics*, 1, (2), 194-204 (June 1995).
7. P. K. Egbert, W. J. Kubitz, ‘Application Graphics Modeling Support Through Object-Orientation’, *Computer*, 84-91, October 1992.
8. B. Erkan, B. Özgüç, ‘Object-Oriented Motion Abstraction’, *The Journal of Visualization and Computer Animation*, 6, (1), 49-65 (1994).
9. M. Gleicher, A. Witkin. ‘Through-the-lens camera control’, *Proceedings of SIGGRAPH’92*, 331-340 (1992).

10. M. Green, ‘A Survey of Three Dialogue Models’, *ACM Transactions of Graphics*, 5, (3), 244-275 (1986).
11. K. P. Herndon, A. van Dam, M. Gleicher, ‘Workshop of the Challenges of 3D Interaction’, *SIGCHI Bulletin*, 26, (4) (1994).
12. L. Koved, W. L. Wooten, ‘GROOP: An object-oriented toolkit for animated 3D graphics’, *ACM SIGPLAN NOTICES OOPSLA’93*, 28, (10), 309-325 (1993).
13. J. Lasseter, ‘Principles of Traditional Animation Applied to 3D Computer Animation’, *Proceedings of SIGGRAPH’87*, 35-44 (1987).
14. M. A. Linton, C. Price, ‘Building Distributed User Interfaces with Fresco’, *Proceedings of the Seventh X Technical Conference*, Boston, Massachusetts, , 77-87, 1993.
15. N. Magnenat-Thalmann, D. Thalmann, *Computer Animation. Theory and Practice*. Computer Science Workbench, Springer Verlag, 1985.
16. M. J. McLennan’ ‘[incr Tcl]: Object-Oriented Programming in Tcl’, *Proceedings of the Tcl/Tk Workshop 1993*, University of California, http://www.wn.com/biz/itcl.
17. M. J. McLennan, ‘[incr Tk]: Building Extensible Widgets with [incr Tcl]’, *Proceedings of the Tcl/Tk Workshop 1994*, New Orleans, http://www.wn.com/biz/itcl.
18. B. A. Myers, ‘User-interface tools: Introduction and survey’. *IEEE Software*, 15-23, Jan. 1989.
19. M. A. Najork, M. H. Brown, ‘Obliq-3D: A High-Level, Fast-Turnaround 3D Animation System’, *IEEE Transactions on Visualization and Computer Graphics*, 1, (2), 175-192 (June 1995).
20. T. Noma, N. Okada, ‘Automating Virtual camera Control for Computer Animation’, *Creating and Animating the Virtual World*, N. Magnenat-Thalman, D. Thalman (Eds.), 177 - 187 (1992).
21. J. O’Rourke, *Computational Geometry in C*, Cambridge University Press, 1994.
22. J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.
23. G. G. Robertson, J. D. Makinlay, S. K. Card. ‘Cone trees: Animated 3D visualizations of hierarchical information’, *Proceedings ACM SIGCHI*, 189-194 (April 1994).
24. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented Modeling and Design*. Prentice Hall, 1991.
25. M. Sannella, *The SkyBlue Constraint Solver,* TR-92-07-02, Dept. of Computer Science, University of Washington, 1992.
26. C. E., G. Schechter, R. Yeung, S. Abi-Ezzi (SunSoft), ‘TBAG: A High Level Framework for Interactive, Animated 3D Graphics Applications’, *Proceedings of SIGGRAPH ‘94*, 421-434 (1994).
27. M. P. Stevens, R. C. Zeleznik, J. F. Hughes, ‘An Architecture for an Extensible 3D Interface Toolkit’, *UIST Proceedings ‘94*, 59 - 67 (1994).
28. P. S. Strauss, R. Carey, ‘An object-oriented 3D graphics toolkit’, *SIGGRAPH’92 Proceedings*, 26, (2), 341-349 (1992).
29. St. Upstill, *The RenderMan Companion. A Programmer’s Guide to Realistic Computer Graphics*, Addison-Wesley, 1990.
30. G. J. Ward. ‘The RADIANCE Lighting Simulation and Rendering System’, *Proceedings of SIGGRAPH’ 94*, 459-472 (1994).
31. P. Wisskirchen. *Object-Oriented Graphics: from GKS and PHIGS to Object-Oriented Systems,* Springer-Verlag, Berlin, 1990.
32. R. C. Zeleznik, K. P. Herndon, D. C. Robbins, N. Huang, T. Meyer, N.h Parker, J. F. Hughes, ‘An Interactive 3D Toolkit for Constructing 3D Widgets’. *Proceedings of SIGGRAPH’93*, 81-84 (1993).
33. R. C. Zeleznik, D. B. Conner, M. M. Wloka, D. G. Aliaga, N. T. Huang, Ph. M. Hubbard, B. Knep, H. Kaufman, J. F. Hughes, A. van Dam, ‘An object-oriented framework for the integration of interactive animation techniques’. *Proceedings of SIGGRAPH’ 91*, 25, (4), 105-112 (1994).

Images taken during the animation of the Voronoi algorithm.