Interactive, Animated 3D Widgets *

Jürgen Döllner   Klaus Hinrichs

FB 15, Institut für Informatik

Westfälische Wilhelms-Universität, Einsteinstr. 62, 48149 Münster, Germany

E-mail: {dollner | khh}@uni-muenster.de

Abstract

If 3D applications become large, hierarchical networks of geometric objects lead to messy specifications. Furthermore, if time- and event-dependencies are merged with the geometric modeling, the global layout of animation and interaction can hardly be achieved. We present an object-oriented architecture for interactive, animated 3D widgets which reduces the complexity of building 3D applications. 3D widgets encapsulate look (geometry) and feel (behavior) into high-level building blocks based on two types of directed acyclic graphs, geometry graphs and behavior graphs. 3D widgets construct internal geometry graphs and behavior graphs, perform operations on these graphs through high-level interfaces which hide details and raise the level of abstraction. 3D widgets define object ports which are used to link together 3D widgets. A visual language for 3D widgets allows the developer the interactive construction of 3D applications.

1. Introduction

During the last few years much progress has been made in the development of object-oriented 3D toolkits [4][5][8][11][15], and new metaphors have been devised [1][17][18]. However, the development of animated and interactive 3D applications is still difficult for several reasons:

3D toolkits provide support for geometric modeling, but do not integrate an explicit time management. Therefore, animation must be integrated at a low-level and is not supported by object-oriented animation concepts. Interaction components are typically provided as complex black-boxes (e.g. [15]) which makes it difficult to specialize these components, to reuse parts of them, and to integrate animation. Often interaction and animation components are attached to geometric components. However, this is difficult to achieve for complex animations and interactions because their relationships are more of a temporal than a spatial nature.

We present a toolkit for interactive, animated 3D widgets based on a new methodology for the symmetric modeling of geometry and behavior. Both geometry and behavior are represented by first class components, the geometry nodes and behavior nodes. These nodes are organized in two types of directed acyclic graphs, the geometry graphs and behavior graphs. A geometry graph is a renderer-independent hierarchical scene description, whereas a behavior graph manages the flow of time and events, and is responsible for time- and event-dependent constraints. Geometry graphs and behavior graphs are associated with shareable graphics objects which are visualized by geometry nodes and constrained by behavior nodes.

The toolkit provides 3D widgets, which construct geometry graphs and behavior graphs, perform operations on these graphs through a high-level interface, and define object ports which are used to link together 3D widgets. The ports of a widget specify which of its internal nodes and graphics objects are visible from outside, and how the widget can be linked to other widgets. 3D widgets simply the usage of node and object networks.

The paper is organized as follows. First, we give a short overview of geometric modeling by geometry graphs. Next, we introduce behavior nodes and behavior graphs, and show how behavioral modeling is related to geometric modeling. Finally, we explain how 3D widgets can be modeled through geometry graphs and behavior graphs. Throughout the paper, we develop an animated triangle set editor as a driving example.

2. Geometry graphs

Geometry graphs are declarative, hierarchical descriptions of 3D scenes similar to OpenInventor or VRML scene graphs. We distinguish between *graphics objects* and *graph nodes*.

2.1 Graphics objects

Graphics objects represent all kinds of 3D primitives and graphics attributes, e.g., spheres, triangle sets, colors, textures. Fig. 1 shows part of the class hierarchy.

These graphical abstract data types can be characterized by their flyweight design [2] which ensures that they are as minimal and as small as possible, and that they can be used in large numbers and implemented efficiently [9]. Graphics objects do not include any context information and do not make any assumptions about their representation in the underlying 3D rendering toolkit, i.e., they do not know how

* This work is support by the Minister of Science and Research of the State of North-Rhine Westphalia, Germany.

to render themselves. They are used to represent application data, not rendering data.

In order to render graphics objects, they are processed by virtual rendering engines which map graphics objects to appropriate calls of the underlying 3D rendering system. A virtual rendering engine is an abstract device connected to a concrete 3D rendering system such as OpenGL. Therefore, we can process the same set of graphics objects for different rendering systems. The Virtual Rendering System VRS [3] is responsible for providing virtual rendering engines.

2.2 Geometry nodes

Geometry nodes are the components for building geometry graphs. Based on their structural properties, geometry nodes are divided into leaf nodes, mono (or transient) nodes, and poly nodes. One or more geometry graphs describe the geometry and appearance of a 3D scene.

In general, geometry nodes provide higher-level operations for graphics objects. In order to render a frame or to answer ray intersection requests geometry graphs are traversed by a virtual rendering engine. Each geometry node sends its associated graphics objects to that engine (e.g., *MThing* which collects attributes and primitives). Geometry nodes can modify the scene composition by calculating transformation matrices (e.g., *MAdjuster* which fits subgraphs into target volumes) or record scene information (e.g., *MWatcher* which stores the current model view transformation matrix).

2.3 Example: A simple triangle set editor

We develop an animated editor for triangle sets. First, we explain the geometry graph for that application, then we will show how the editor can be animated by behavior graphs. Later we will simplify the application by using 3D widgets.

The editor visualizes a triangle set and its convex hull. Part of the editor's geometry graph is shown in Fig. 2. The *MAdjuster* node *exhibit* fits its subgraph into the volume *vol*

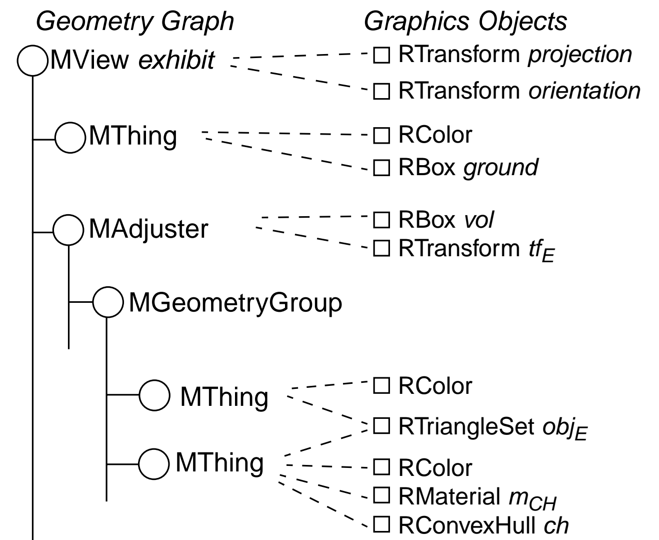


Figure 2. Geometry description of the triangle set editor.

given by an *RBox* graphics object. It applies an *RTransform* graphics object tf_E to its subgraph. Both *MThing* nodes refer to the same graphics object obj_E of type *RTriangleSet*. Additionally, a material graphics object m_{CH} is applied to the convex hull. To render an image of an exhibit, we connect its geometry graph to a canvas (Fig. 3a) which provides a rendering engine. Fig. 3 shows a dragon head rendered with an OpenGL rendering engine (b) and with a Radiance [16] rendering engine (c). A canvas is controlled by the studio, the controller for geometry graphs, behavior graphs, and 3D widgets.

The main differences between traditional approaches for hierarchical scene descriptions and our approach are:

- Graph nodes are distinguished from graphics objects which allows us to easily customize graphics objects for the needs of specific applications without having to change the scene modeling classes.

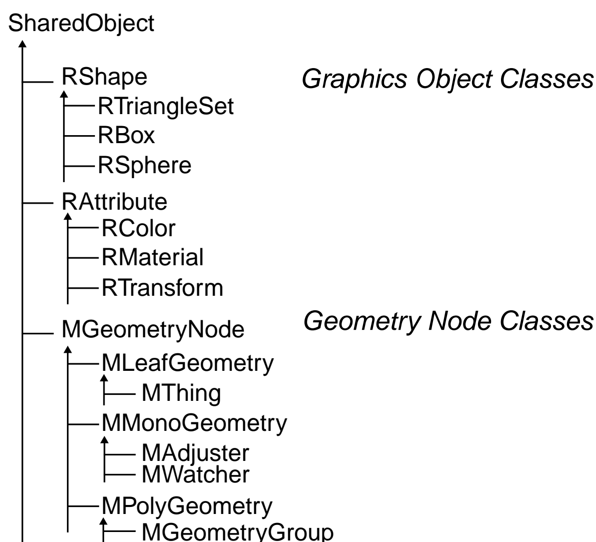


Figure 1. Graphics objects and geometry node classes.

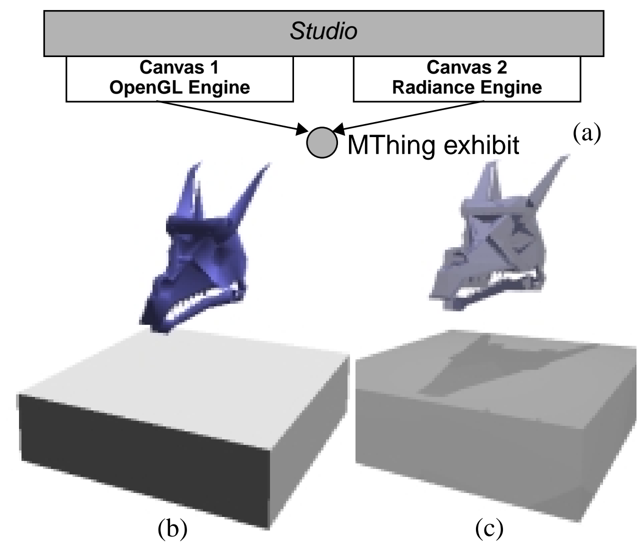


Figure 3. Root of the geometry graph (a). Image generated by OpenGL (b) and by Radiance (c).

- Graphics objects are associated with both geometry graphs and behavior graphs, and therefore inherently linked to scenes and animations.
- Graphics objects are processed by virtual rendering engines which allow us to map them to different rendering toolkits (e.g., OpenGL, RenderMan).

3. Behavior graphs

Behavior graphs specify the animation and interaction of a 3D application. They are closely related to geometry graphs because they usually apply constraints to graphics objects which are associated with geometry nodes.

Behavior nodes can be divided into two groups: time-related behavior nodes which synchronize animations and event-related behavior nodes which manage interaction events. Together with geometry nodes, behavior nodes represent the basic building blocks for 3D widgets. Fig. 4 shows part of the behavior node class hierarchy.

3.1 Time manager behavior nodes

The time management provided by behavior nodes is based on the following temporal abstract data types:

- *Time:* a float value, measured in milliseconds. Corresponds either to the real time or a model time.
- *Time requirements*: specify time demands of behavior nodes. A time requirement consists of the natural (i.e. desired, optimal) duration, a stretchability, and a shrinkability. For example, the time requirement [10sec, -4sec, +5sec] specifies a desired lifetime of 10 seconds. However, the time management may assign only 6 seconds lifetime due to the shrinkability of -4, or may assign at most 15 seconds lifetime. Durations can be specified as infinite.
- *Moments*: represent points in time. A moment contains the current point in time (e.g., real time), and the lifetime interval of the behavior which receives the moment. Moments are sent by time events to behavior nodes in order to active behavior nodes and trigger the synchronization of animations. Based on the lifetime contained in a moment, behavior nodes can plan and distribute their activity.

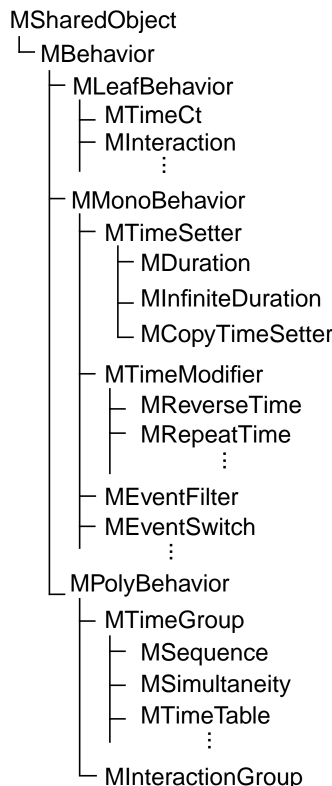


Figure 4. Behavior node classes.

3.2 Time management

Time requirements and moments are used by time managers to negotiate with behavior nodes and to synchronize their activity. To activate a behavior graph, it is connected to a time manager which requests the behavior graph's time requirement. In a bottom-up process, the behavior nodes calculate the total time requirement. Based on that information, the time manager assigns the behavior graph a lifetime. During the lifetime, the time manager sends synchronization events to the behavior graph. These events are evaluated by the behavior nodes in a top-down process (s. Fig. 5).

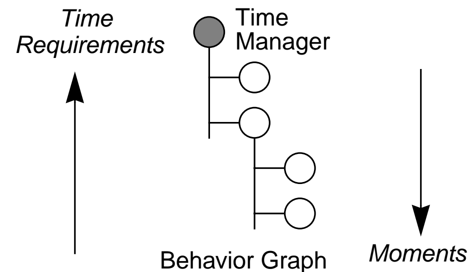


Figure 5. Time management for behavior graphs.

Time managers obtain time events from clocks which constructs time events. We distinguish between two types of clocks: real-time clocks send the actual system time, and model-time clocks send time events in an order and frequency defined by the application (e.g., to record an animation at a fixed rate of frames per second, the application must use a model-time clock).

3.3 Time-related behavior nodes

Time-related behavior nodes are categorized into time setters, time modifiers, and time groups.

Time Setters

Time setters define the time requirements of its subgraphs. It can define time stretchability and time shrinkability, natural durations (*MDuration*), and infinite durations (*MInfiniteDuration*). It can also use the time requirement of another behavior node for its body (*MDurationCopy*).

Time Modifiers

Time modifiers transform time requirements of their subgraphs. Time modifier classes include: *MTimeRepeater* behavior nodes map a moment modulo another moment, and passes the resulting moment to its subgraph. *MTimeReversal* behavior nodes invert moments and delegate the inverted moments to their subgraphs. Thus, for a subgraph the direction of the time progress is inverted. Time reversal nodes are useful to model retrograde actions.

Behavior Groups

Behavior groups use a time layout to manage their child nodes. A time layout calculates the individual lifetimes of the child nodes based on their time requirements. If a behavior group is synchronized to a new time, 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. Examples for time layouts are:

- *Sequence*: It distributes time in sequential, disjoint moments to its child nodes. Only one child node is alive at any given time during the duration of the sequence.
- *Simultaneity*: It aligns all child nodes to its own life time. If a child node defines a shorter (or longer) natural life time, the layout tries to stretch (or shrink) the child nodes' life time.
- *FadeIn* and *FadeOut*: *FadeIn* layouts assign moments to their child nodes which start in cascading order like in the case of the sequence layout, but all child nodes remain activated until the last child node is deactivated. *FadeOut* layouts are reversed *FadeIn* layouts.
- *Time Table:* Uses explicitly given lifetimes for each child node. The lifetime of a child node can be specified in absolute points in time, or relatively to the lifetime of the behavior group.

The introduction of time requirements and time setters was motivated by the space requirements and space setters developed in InterViews [9], and by the TEX [19] boxes and TEX glue model. Applied to time, it allows us to specify animations at an abstract level and makes it unnecessary to calculate the exact time of an action in an animation. Behavior groups provide an automatic time negotiation mechanism based on time layouts.

3.4 Constraints

Constraints form a main category of behavior nodes. A constraint node is associated with the constrained object and establishes its constraints at the beginning of its lifetime, and removes the constraints when its lifetime ends. Constraint networks managed by constraint solvers (e.g., SkyBlue [13]) can be anchored in behavior graphs and connected to the flow of time and events by behavior nodes.

Time-dependent functions which are applied to parameters of graphics objects represent a main category of constraint nodes. A time constraint behavior node (*MTimeCt*) associates an attribute of a graphics objects with a time-dependent function (*MTimeMap*). To maintain the encapsulation of the graphics object, the *MTimeCt* node knows the member functions (i.e., the methods) of the graphics object which set and retrieve the attribute.

An *MTimeMap* represents a function which maps a given moment to a numerical value of a generic type. If the constraint node is synchronized, the mapping calculates the new parameter values and applies them to the graphics object. For example, an *MLinearTimeMap<Vector>* mapping interpolates two given vectors during the time interval defined by the moment. The interpolation proceeds slower at the beginning and at the end. A collection of mapping classes is shown in Fig. 6.

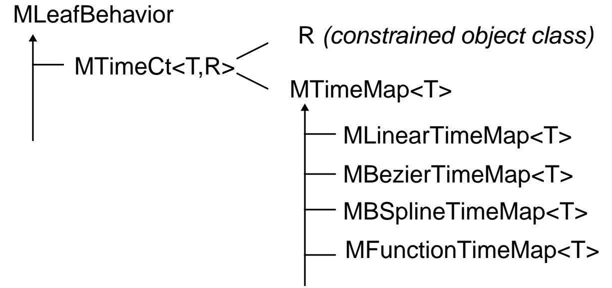


Figure 6. Time constraint classes.

3.5 Animating the triangle set editor

The animation fades in a new object loaded by the triangle set editor. In the beginning the object is hidden in its convex hull. The object's volume is enlarged to its default size. Then, the convex hull becomes transparent. To fade out the object, we reverse the fade-in behavior. Additionally, we define an automatic rotation for the object which can be activated and deactivated. The behavior graphs for these animations are shown in Fig. 7.

The fade-in behavior consists of two sequential behaviors. The first behavior consists of two simultaneous behaviors which both constrain the bounding box *vol* (the target volume into which the triangle set is pressed) during 3 seconds. Two time-dependent functions are applied to the minimal point and maximal point of the bounding box. The second behavior constrains the material m_{ch} during the next 2 seconds (it actually constrains the transparency coefficient of the material).

The fade-out behavior consists of an *MTimeReversal* node linked together with the fade-in behavior. Moments sent to the fade-out behavior node are reversed, i.e. we get the retrograde animation.

The *rotate-obj* behavior has an infinite time requirement. Infinite moments are mapped to the moment [0, 5sec] by *MTimeRepeat*. The time constraint maps incoming moments to the numeric range [0, 360] and applies the new angle coefficient to the transformation tf_E., i.e. a full rotation is performed every 5 seconds.

In the example, the fade-in behavior requires a natural time of 4 seconds. The actual time assigned to the sequential behavior group may differ but is distributed in a 3:1 proportion to both child nodes.

The behavior graphs of this example are linked together by a time manager which is controlled by the studio (Fig. 8). The triangle editor application can activate or deactivate

individual behaviors based on user input or user interface actions.

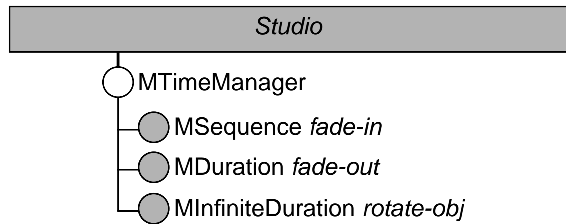


Figure 8. Behavior graphs for animating the editor.

3.6 Interaction by behavior nodes

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. Most 3D toolkits (e.g. [5][15]) provide low-level interaction techniques, but their reusability is limited due to bindings to concrete geometry types and their coarse-grained object oriented design.

In our approach, interaction can be specified by behavior nodes which evaluate time events and canvas events. These interaction nodes can be used to specify complex interactions and multi-state augmented transition networks [7].

The base class *MInteraction* defines four states: starting, processing, terminating, and canceled. The states change if start, end, termination, or cancel conditions are satisfied. Interaction nodes can be linked together to build complex interactions. State changes are propagated to child interaction nodes. Since interaction nodes use their own event types for communication, other behavior nodes can be inserted between them. Basic interaction nodes are:

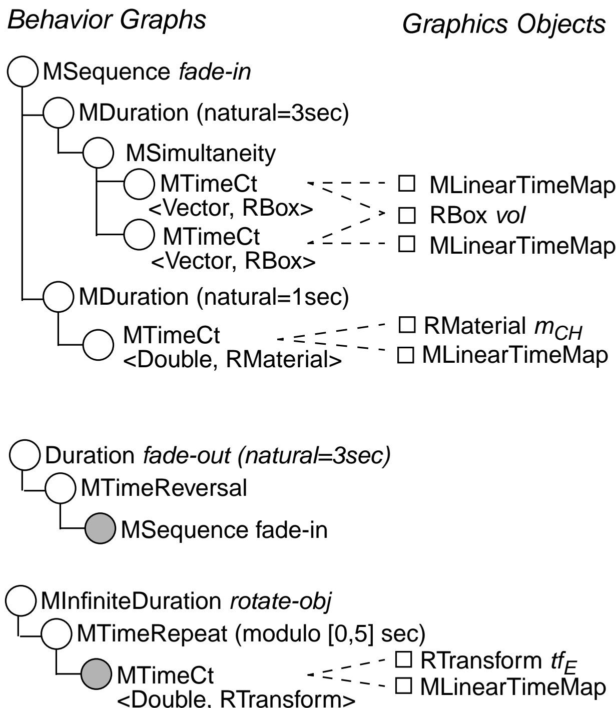


Figure 7. Behavior graphs for animating the editor.

- *Manipulator nodes:* They map device events (e.g., mouse motion events) on graphics objects. Manipulators include trackballs, camera manipulators, and 3D sliders.
- *Event filter nodes:* They filter certain types of events and decide how to distribute events to subgraphs. They are useful to optimize event handling and to switch between interactions.
- *Selectors*: Selectors process ray requests and picking requests. They return information about the objects which have been hit by a ray, a line or a pixel. A request processed by a selector can be restricted to graphics objects which belong to a specified group. All graphics objects can be assigned group identifications by tag attributes.

State changes are triggered if conditions are satisfied. Conditions are modeled by condition objects which can be composed to complex conditions. For example, a key condition can be combined with a mouse motion condition, and together they could trigger the rotation of a trackball.

4. 3D widgets

Behavior nodes provide a simple method to compose animation and interaction descriptions. Geometry graphs provide also a straight forward method to compose 3D scenes. The relation between geometry graphs and behavior graphs is mainly determined by shared graphics objects. To reduce the complexity of building 3D applications even more, frequent geometry graph patterns and behavior graph patterns can be encapsulated in 3D widgets. We adopt the definition of widget as “an encapsulation of geometry and behavior used to control or display information about application objects” [1].

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

3D widgets are defined by their ports, resources and internal graphics objects and nodes.

Ports

A port determines 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 graphics 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.

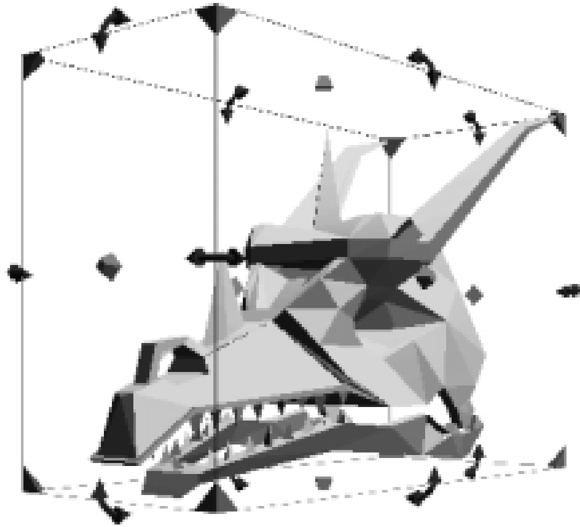


Figure 10. Transformer box applied to a shape.

We use the following symbolic notation for port specifications:

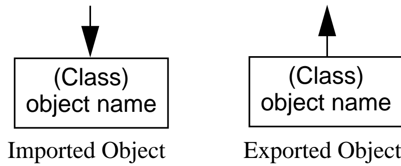


Figure 9. Notation for port specifications.

Resources

Resources specify attributes of internal geometry nodes, behavior nodes and graphics objects. A resource is defined by its name, its type, its default value, and its current value. Resources are mainly used to configure 3D widgets by resource files or interactively by a resource editor. This mechanism has been adopted from 2D user interface toolkits.

4.1 Example 1 : transformer box widget

To illustrate the internal structure of 3D widgets, we will show in the following how to construct an interactive transformer box (Fig. 10) with handles to scale and to rotate a shape. The geometry graph for this widget arranges 6 side scale handles, 8 corner resize handles, and 12 rotation handles around a wire-frame bounding box. The behavior graph for the transformer widget consists of manipulator nodes, one for each handle. The manipulator nodes constrain the associated *RTransform* graphics objects *scalingTf* and *rotationTf* which contain the actual transformation matrices. Both transformations represent the "output channels" of the widget. The graphs are illustrated in Fig. 11.

The graphs can be encapsulated in the *TransformerWidget* as shown in Fig. 12. A transformer widget expects a scaling and a rotation transformation which are connected to those geometry nodes which should reflect the actual transformation. The widget exports the handles' behavior *tfBoxBehavior* and the geometry graph *tfBoxGeometry* (see also Fig. 11).

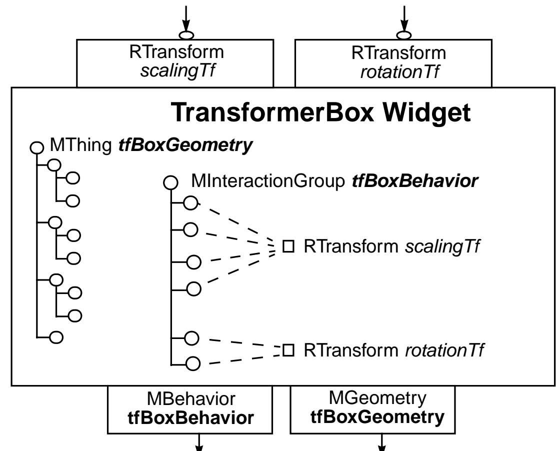


Figure 12. TransformerWidget and its ports.

Example 2 : Triangle Set Editor Widget

The editor widget for triangle sets allows users to select triangles, edges, and vertices of a triangle set. Several operations can be performed on selected parts: delete triangles, delete edges, delete vertices, add new triangles, refine selected triangles, smooth selected triangles etc. This widget implements the core functionality of the triangle editor application.

The geometry graph consists of *MThing* nodes which visualize differently the selected and unselected parts. For example, unselected vertices are visualized as 3D points whereas selected vertices are represented by small spheres. The behavior graph consists of *MSelector* nodes which perform ray intersection tests in order to determine if the user has selected parts of the triangle set.

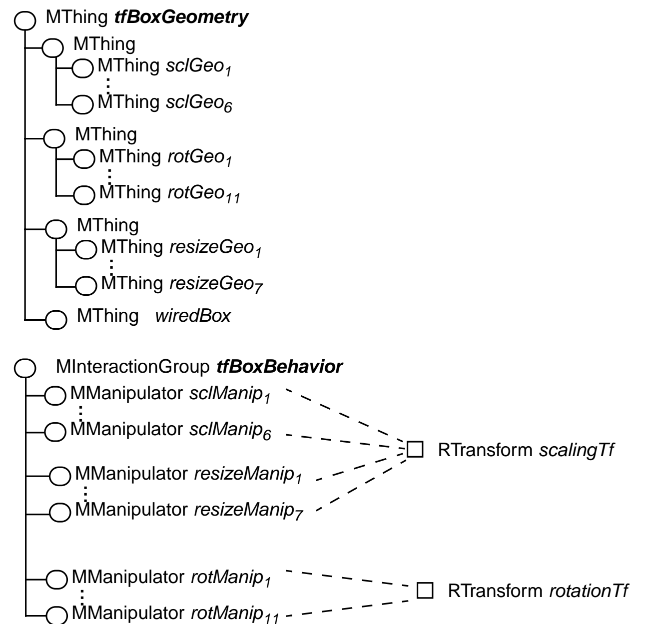


Figure 11. Geometry graph and behavior graph for an object transformer.

The widget expects a triangle set and an MWatcher node (which should be installed in the geometry graph of the application). The watcher is used to adjust the coordinate systems between the triangle set in the application's geometry graph and the editor's geometry graph. A watcher node records separately the transformation matrices for each path in a geometry graph and for each canvas which draws the geometry graph. The editor widget exports its geometry graph, behavior graph, and the current selections.

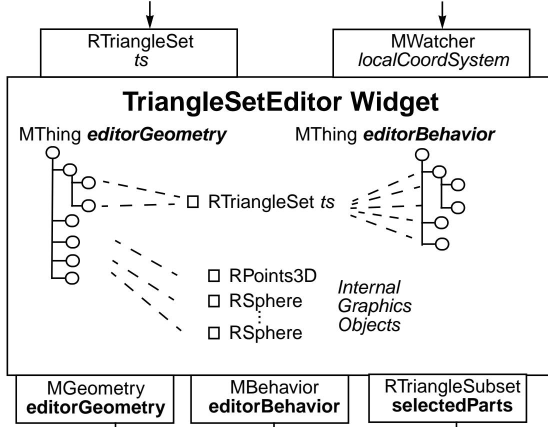


Figure 13. Triangle set editor widget and its ports.

Example (Part 3) : Pedestal Widget

The pedestal widget exhibits a shape onto a pedestal and displays a title by 3D characters on the vertical sides of the pedestal. The shape can be scaled and transformed by a linked transformer box widget. The pedestal widget is used in our example to display the editor's triangle set.

The widget's geometry graph contains the layered boxes, the 3D characters, and a node for displaying the shape. It exports the local coordinate system of the shape by a watcher node, and the transformation graphics objects used for connecting to the transformer box.

The pedestal widget's behavior graph defines behaviors to show the title (moving it out) and to hide it (moving it inside the box), to fade in the shape, and to fade out the shape. Parts of the geometry graph and behavior graphs developed in the previous sections can be reused.

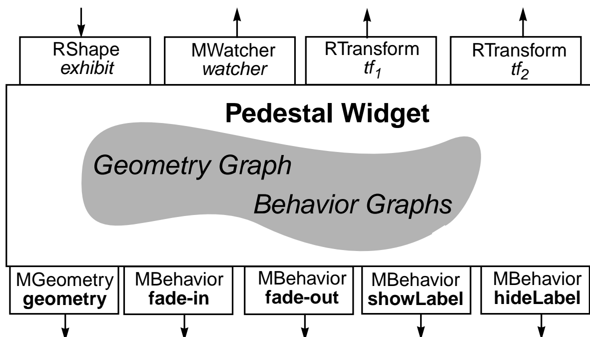


Figure 14. PedestalWidget and its ports.

4.2 3D Widget visual language

The programming of 3D applications can be simplified

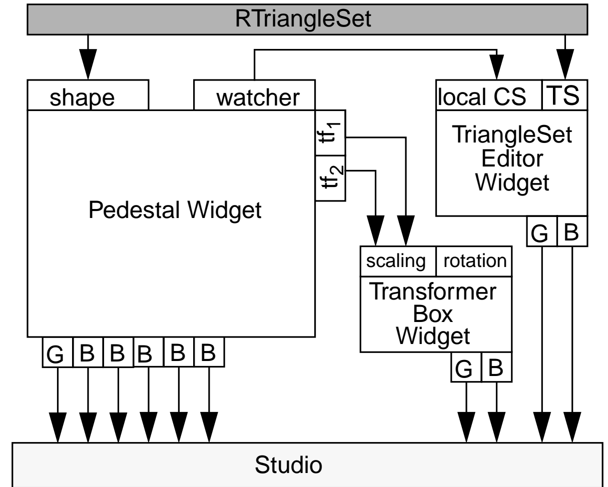


Figure 15. 3D widget configuration for the object editor.

using a visual language for 3D widgets, their ports and relations. Fig. 15 illustrates how the triangle set editor can be rewritten as a network of 3D widgets. We implement the functionality of the triangle set editor by instances of the Pedestal Widget, the TransformerBox widget, and the TriangleSetEditor widget. Their geometry graphs and behavior graphs are bound to a studio. The studio associates geometry graphs with canvas objects. It connects behavior graphs to time managers. The behaviors are activated (resp. deactivated) the user interface. Several steps of the animation, started when a new triangle set is loaded, are shown in Fig. 16.

An editor for the visual construction of 3D widgets is currently under development. Basically, a 3D widget supplies a full specification of its ports to the editor. Runtime type information for nodes and graphics objects is used together with the port specifications to check the semantic of object relations.

4.3 Collection of 3D widgets

We have defined a few experimental 3D widgets. This collection is far from being complete but may serve as a starting point for a development of a library for general purpose 3D widgets.

Transformer Box

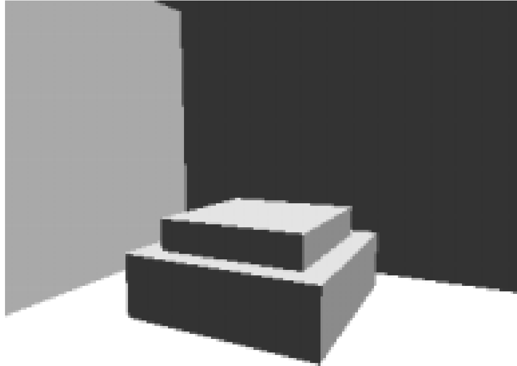
A *Transformer* widget constructs a 3D tool used to rotate, scale, and translate a shape in 3D space. Its geometry graph and its behavior graphs are basically designed as shown in Fig. 11.

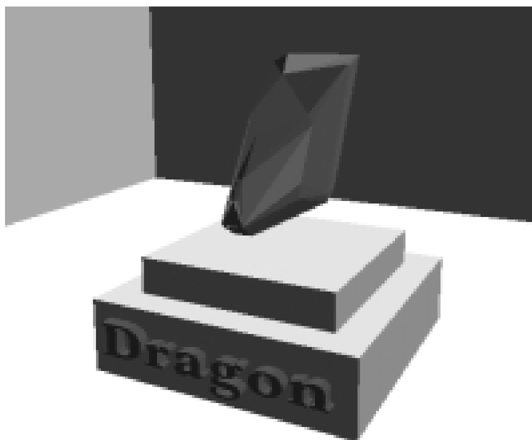
CameraCockpit

A *CameraCockpit* widget consists of a complex geometry graph which specifies the cockpit's instruments, and a complex behavior graph which interprets the movements of the steering-wheel and controls the instruments. The widget constrains an imported scene node.

Text

A *Text* widget visualizes 3D characters. The characters can be organized by different layouts, e.g. aligned along a curve. Multi-line texts are supported. The *Text* widget provides callback bindings for individual characters.





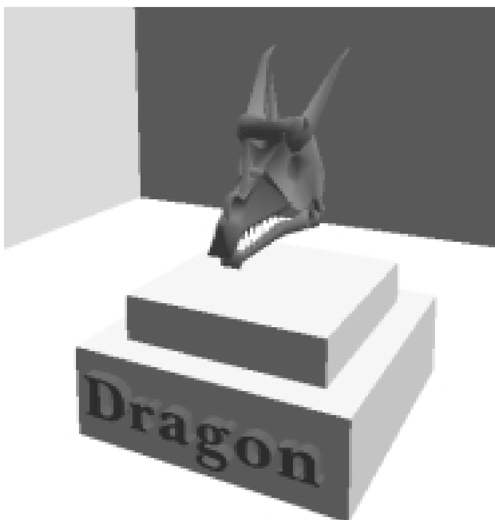


Figure 16. Object viewer during the animation.

Figure 17. CameraCockpitWidget applied to the

Editor Group

A *EditorGroup* widget controls a set of shapes. Each of the shapes can be selected or deselected. For a selected shape, graphics attributes are temporarily installed (e.g., a point light source in the center of the shape to highlight the current selection). The widget defines a behavior subgraph containing interaction nodes which detect shape selections.

Slider

A slider widget is the 3D equivalent to a 2D slider. The geometry graph defines the look of the slider. The behavior graph provides a slider handler.

Button

Like its 2D equivalent, a *Button* widget is associated with actions defined by a command interaction. The widget imports a geometry graph whose shape is aligned with the button's geometry.

Shadow

A *Shadow* widget calculates shadows of an associated shape object for a set of shadow planes. Its geometry graph embeds the shadow polygons in the 3D scene. Its behavior graph defines interactions for each individual shadow. This widget can be used to implement interactive shadows.

5. Implementation

The presented concepts have been realized in *MAM/VRS*, a 3D graphics and visualization library implemented in C++. The *Virtual Rendering System* VRS supports the rendering libraries OpenGL, RenderMan, POV Ray, and Radiance. The *Modeling and Animation Machine* MAM is an ongoing project which serves as framework for implementing interaction and animation strategies. As application programming interface, developers can choose the C++ API or an embedding of MAM/VRS in Tcl/Tk [12]. Using the interpretative Tcl/Tk language facilitates the rapid pro-

totyping of 3D applications and guarantees portability across different operation systems. For more information, see `http://wwwmath.uni-muenster.de/~mam`.

6. Related work

In the last few years several object-oriented 3D graphics toolkits have been proposed, e.g. GRAMS [4], OpenInventor [15], GROOP [8], TBAG [5], Obliq-3D [11]. These toolkits introduce object-oriented concepts applied to 3D graphics.

GRAMS appears to be one of the first toolkits with rendering-independent 3D graphics. OpenInventor provides geometric node classes based on the OpenGL rendering library. It supports basic interaction handling mechanisms. These toolkits concentrate more on 3D graphics than on user interface construction.

TBAG is characterized by its functional approach to 3D graphics based on constrainable, time-variant graphical data types. Obliq-3D uses time-variant properties for animating objects. Our approach complements these object-oriented concepts by an object-oriented design for behavioral modeling.

Based on UGA [18][17][1][14] several new metaphors for 3D widgets and concepts for visual programming of 3D widgets have been developed. 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 on an object-oriented design for 3D widgets and its implementation structures, i.e. geometry nodes and behavior nodes.

7. Conclusions and Future Work

The key features of the presented architecture for animated, interactive 3D widgets are:

- Graphics objects are separated from geometry nodes which makes it easy to install dependencies between geometry and animation. Graphics objects can be shared throughout the whole application. Application-specific graphics objects allow developers to embed their own data structures directly into the graphics toolkit.
- Look and feel of a 3D widgets can be expressed by geometry nodes and behavior nodes. Since behavior nodes are treated as first-class objects, general and reusable behavior components can be designed without being part of a specific geometry class. The time management facilitates the design of complex time flows.
- 3D widgets raise the level of abstraction at which 3D applications are developed. The construction and the maintenance of hierarchical object networks is completely encapsulated into 3D widgets which represent geometry graph patterns and behavior graph patterns in a compact way.
- The ways 3D widgets can be combined are defined by their ports which offer the prerequisites for the visual programming of 3D widgets.

The presented object-oriented architecture for interactive, animated 3D widgets provides a framework for the rapid development of large 3D applications. Future work includes the development of specialized 3D widgets for visualizing and exploring multi-dimensional geo-data.

References

[1] D. Conner, S. Snibbe, K. Herndon, D. Robbins, R. C. Zeleznik, A. van Dam, *Three-Dimensional Widgets*. Computer Graphics (1992 Symposium on Interactive 3D Graphics), Vol. 25, No. 2, March 1992, pp. 183-188.

[2] P. R. Calder, M. Linton, Glyphs: Flyweight Objects for User Interfaces. *Proceedings of the ACM SIGGRAPH Third Annual Symposium on User Interface Software and Technology*, 1990.

[3] J. Döllner, K. Hinrichs: The Design of a 3D Rendering Meta System. In: *Eurographics Workshop on Programming Paradigms for Graphics '97*, Budapest, 1997.

[4] P. K. Egbert, W. J. Kubitz, Application Graphics Modeling Support Through Object -Orientation, *Computer*, October 1992, pp. 84-91.

[5] C. Elliot, G. Schechter, R. Yeung, S. Abi-Ezzi SunSoft, Inc. *TBAG: A High Level Framework for Interactive, Animated 3D Graphics Applications. Proceedings of SIGGRAPH '94*, pp. 421-434.

[6] M. Gleicher, A. Witkin. Through-the-lens camera control. *Proceedings of SIGGRAPH'92*, pp. 331-340.

[7] M. Green, A Survey of Three Dialogue Models. *ACM Transactions of Graphics*, Vol. 5, No. 3, July 1986, pp. 244-275.

[8] L. Koved, W. L. Wooten, GROOP: An object-oriented toolkit for animated 3D graphics. *ACM SIGPLAN NOTICES OOPSLA'93*, Vol. 28, No. 10, October 1993, pp. 309-325.

[9] M. Linton, J. Vlissides, and P. Calder. Composing user interfaces with InterViews. *IEEE Computer*, pp. 8-22, February 1989.

[10] B. Myers, User-interface tools: Introduction and survey. *IEEE Software*, Jan. 1989, pp. 15-23.

[11] M. Najork, M. Brown, Obliq-3D: A High-Level, Fast-Turnaround 3D Animation System. *IEEE Transactions on Visualization and Computer Graphics*, Vol 1, No. 2, Juni 1995, pp. 175-192

[12] J. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.

[13] M. Sannella. SkyBlue: A Multi-Way Local Propagation Constraint Solver for User Interface Construction. *Proceedings of UIST'94*, pp. 137-146.

[14] M. Stevens, R. C. Zeleznik, J. F. Hughes, An Architecture for an Extensible 3D Interface Toolkit. *Proceedings of UIST'94*, pp. 59-67.

[15] P. Strauss, R. Carey, An object-oriented 3D graphics toolkit. *SIGGRAPH'92 Proceedings*, Vol. 26, No. 2, pp. 341-349.

[16] G. J. Ward, The RADIANCE Lighting Simulation and Rendering System, *Proceedings of SIGGRAPH' 94*, pp. 459-472.

[17] R. C. Zeleznik et al., An object-oriented framework for the integration of interactive animation techniques. *Proceedings of SIGGRAPH' 91*, Vol. 25, No. 4, pp. 105-112.

[18] R. C. Zeleznik et al., An Interactive 3D Toolkit for Constructing 3D Widgets. *Proceedings of SIGGRAPH'93*, pp. 81-84.

[19] D. E. Knuth. *The TEXbook*. Addison-Wesley, Reading, MA, 1984.