OpenSG Symposium (2003)
D. Reiners (Editor)

OpenManip: An Extensible Cross-Scene-Graph Framework for Direct Object Manipulation

Michael Braitmaier, Manfred Weiler, Thomas Ertl

Visualization and Interactive Systems Group
University of Stuttgart[†]

Abstract

In this paper we describe a framework that exploits 3D widgets in order to allow for the direct manipulation of scene graph objects. The design of the frameworks is inspired by the OpenInventor manipulator functionality, but additionally emphasizes transparency from the underlying scene graph system, by separating core functionality from a relatively lean scene graph abstraction layer. Thus, the framework features different scene graph APIs, in particular OpenSG and Cosmo3D. Using our framework manipulation functionality can be easily integrated into any existing application since it only introduces a few new objects that have to be connected to the application. Our framework provides different manipulators for selection, scaling, rotation, and translation of objects. Moreover a set of editors allows for the manipulation of light and material properties. We demonstrate the extensibility of our framework both, in terms of customized manipulators and porting the framework to new scene graph APIs, which is supported by a clear object-oriented structure.

1. Introduction and Related Work

Direct manipulation is a style of Human Machine Interaction (HMI) design which features a natural representation of task objects and actions in a computer environment and, therefore, realizes many benefits in contrast to traditional 2D menus or command line languages. An intuitive interface and the lack of a complex syntax, for instance, dramatically increases the learning speed and retention. Moreover a real-time feedback avoids handling errors and the reversibility of actions helps the user to gain confidence in the system. Because of these benefits, direct manipulation techniques are a widely used interaction pattern not only in computer graphics application. Dragging an unused file onto the recycle bin on the desktop is only one typical example for direct manipulation.

Considering especially VR applications direct manipulation seems to be the natural interaction pattern as well, for instance, when a data glove or a 3D pointing device is used to manipulate objects in the virtual environment, e.g. in a digital mock-up scenario. The user can interact with the presented parts in a natural and intuitive way performing the desired task.

However, not every task in an virtual environment acts on objects that are part of the scene. Kniss et al. [6], for example, use direct manipulation to modify the transfer function of a rendered volume. In such cases 3D direct manipulation widgets are widely used [3, 10, 5]. These widgets are special geometric objects rendered with a visualization and are designed to provide the user with a 3D interface. Widgets are typically rendered from basic geometric primitives such as spheres, cylinders, and cones with each sub-part of a widget representing some functionality of the widget or a parameter to which the user should have access.

Direct manipulation can also be combined with a gravity function that snap the cursor on nearby vertices, edges, surfaces or their intersection. Bier et al. [2, 1] have successfully applied this technique to build an interactive 3D solid modeling systems, that allows for an efficient and precise positioning or scaling of geometrical primitives.

[†] Universität Stuttgart, VIS, Universitätsstraße 38, 70569 Stuttgart, Germany; E-mail: `braitmml@vismail.informatik.uni-stuttgart.de`, `{Manfred.Weiler | Thomas.Ertl}@informatik.uni-stuttgart.de`.

From the different available 3D graphics API currently only OpenInventor [9] intrinsicly supports manipulation of scene graph objects by the use of special manipulator scene graph nodes, and, therefore allow an application programmer to incorporate intuitive scene graph modification into any OpenInventor application. However, OpenInventor is less suitable for VR applications since it lacks support for multi-pipelining and multi-processing in contrast to the full-immersive Performer API.

Our goal, therefore, is to provide a future-proof open source framework that allows for an easy integration of direct scene graph manipulation into a wide variety of interactive and immersive 3D applications. We achieve this by separating the manipulation core functionality form scene graph dependent code, thus, allowing to support different scene graph systems, e.g. OpenSG and Cosmo3D, and to easily adapt the framework to future scene graph APIs.

The remainder of the paper is organized as follows: In Section 2 we will first shortly review OpenInventor manipulators since this provides a better understanding of the architecture of our framework presented in Section 3. The abstraction layer that allows for independency of the underlying scene graph system is introduced in Section 4. The integration of our manipulator functionality is discussed in Section 5 before we present achieved results in Section 6.

2. OpenInventor Manipulators

In OpenInventor direct manipulation of scene graph objects is obtained by two types of special scene graph objects, namely draggers and manipulators. A dragger is a node — or more precisely a node kit — in the scene graph with specialized behavior that enables it to respond to user events. Furthermore, draggers can insert geometry into the scene graph that is used for picking and user feedback. A dragger reacts on user input in a specified way and, thus, can be considered as implementing its own user interface. It has a field (or fields) reflecting its state. A very simple dragger, for example, could record the state of the CTRL-Key in an internal field which might be used to switch between flat shaded polygons and wire frame display.

OpenInventor provides simple and compound draggers. Simple draggers perform only one operation such as a scale or a translation in one dimension and have a fixed user interface. Although simple draggers can be useful by themselves, they are often combined to make a compound dragger. Compound draggers use multiple simple draggers to combine several operations, e.g. the `SoTransformBoxDragger` combines a rotation, translation, and scale.

Manipulators on the other hand are subclasses of other OpenInventor nodes such as `SoTransform` or `SoDirectionalLight` that employ draggers to respond to user events and edit themselves. Deriving a manipulator from an OpenInventor node literally defines a user interface for that type of node. Each manipulator contains a dragger as hidden child that responds directly to user events and in turn modifies the field of the manipulator. A `SoTrackballManip`, for example, is a subclassed `SoTransform` that uses a `SoTrackballDragger` to modify its transformation matrix according to the user's pulling of one of the ribbons surrounding a displayed sphere.

It is important to understand that a dragger moves only itself in response to user events, whereas a manipulator moves itself and also affects other objects in the scene graph.

OpenInventor manipulators provide a powerful mechanism for direct manipulation of scene graph objects. An application programmer can easily — without much implementation effort — allow the user to move, rotate, or scale objects, e.g. by dynamically inserting a transform manipulator node in the scene graph without having to take care about the user interaction himself. He simply removes the manipulator node after the end of a manipulation sequence, which automatically replaces the manipulator by a transform node with the same transformation matrix making the changes to the scene graph permanent. In the same way modifiable lights can easily be added to any application exploiting the light manipulator nodes of OpenInventor.

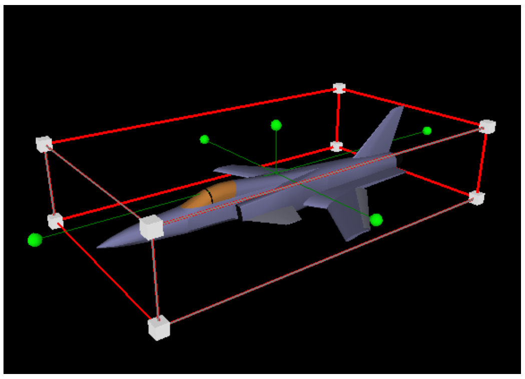

Figure 1: *The `SoTransformBoxManip` contains a `SoTransformBoxDragger` consisting of six `SoTranslate2Draggers` for the faces of the cube, three `SoRotateCylindricalDraggers`, and one `SoScaleUniformDragger`. It allows for the translation, rotation and uniform scaling of any scene graph object.*

Figure 1 shows the most prominent representative of OpenInventor manipulators. This `SoTransformBoxManip` allows for rotation of the presented fighter around any of the coordinate axises x, y, and z, by dragging one of the green spheres. Picking any face of the box translates the fighter in the plane of that face or perpendicular to the face if the CTRL-key is pressed. Dragging of any of the white cubes scales the box uniformly or non-uniformly in one direction if the SHIFT-key is pressed. Note that this is only a selection of the possible interaction modes. When the user begins manipulating, an arrow appears indicating the direction of the motion.

Several manipulator nodes are available in the OpenInventor library providing different interaction modes for the object manipulation. Moreover a rich set of over 20 draggers allows for writing own manipulators with user specific behavior.

3. OpenManip Architecture

The OpenManip architecture consists of two layers to allow for a clear separation of the different abstraction levels. The higher level realizes the manipulator functionality whereas the lower level implements the connection to the underlying scene graph API, which will be discussed in Section 4. The OpenManip functionality concerning the manipulators features an object-oriented structure, which aims at easy extensibility.

The design of OpenManip was inspired by the widely known OpenInventor manipulators, but additionally emphasizes scene graph API transparency and support for the cross-platform C++ widget set Qt from Trolltech [4, 8].

The OpenManip manipulator layer is composed of two parts reflecting the major function blocks. These functional parts are the manipulator core and the event handling. For each of these parts a class hierarchy is defined that allows for easy creation of new manipulators.

The manipulator core performs all actions, related directly to the manipulation of objects and to the visual appearance of the manipulator within the application. The interface between the application's event handling and the manipulator core is provided by the OpenManip event handling. It checks whether specific events are relevant for the manipulator. We will first present a more detailed description of the manipulator core, including components and their interactions followed by the event handling.

3.1. Manipulator Core

Our framework utilizes the OpenInventor concept of manipulators and draggers presented in Section 2. However, the design does not derive manipulators from scene graph nodes, since this would have amalgamated the manipulator functionality with the underlying scene graph API and, thus, severely hampered portability between different APIs. Instead, our framework creates an own object graph parallel to the scene graph where each object links to a corresponding scene graph node. The visual representation of the manipulator uses ordinary geometry nodes of the scene graph API.

The manipulator core part consists of a small set of generic base classes that already implement base functionality. For example the `Manipulator` base class handles the administration and the deletion of all aggregated objects, in particular draggers. Thus, the specialized classes only have to take care about the dragger setup, which actually make up the particular characteristics of a manipulator. Figure 2 demonstrates this relations for a translation manipulator.

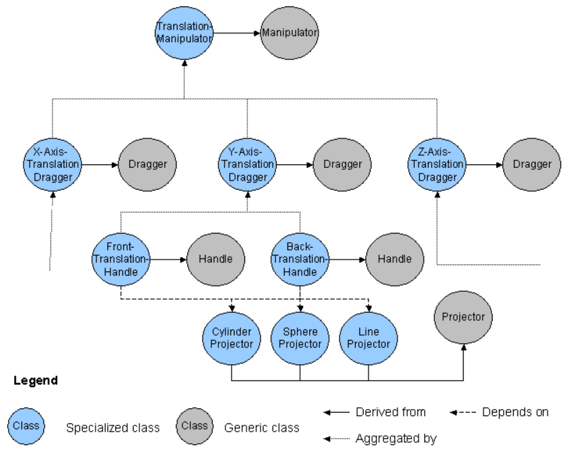

Figure 2: *The architecture of OpenManip manipulator core part. It shows the inheritance between generic and specialized classes, as well as the dependencies between the hierarchy levels.*

Besides from the handling of aggregated objects the manipulator class and its derivations are only administrative objects acting as a connection between draggers and the scene graph. For example the manipulator objects perform all tasks of directly accessing the transformation matrices of a corresponding node by calling appropriate methods from the objects of the scene graph driver.

The corresponding node is determined during the creation of a `Manipulator` object. If the parent of an object that should be modified by a manipulator is a transform node, the manipulator links to this transform node. Otherwise a transform node is inserted above the node to be manipulated and connected to the manipulator. This node stays in the scene graph for permanent effect of the manipulation action.

A `Manipulator` in the context of `OpenManip` is basically a set of `Draggers` where each dragger represents a specific transformation task or a set of draggers. Note that a transformation task refers to a basic transformation, e.g. a translation, rotation, or scaling. Although our framework features compound draggers as well, the draggers presented here use only basic transformation.

A dragger contains all geometry required for a specific type of transformation. We distinguish so called `Visuals`, that have no interaction with the user, from `Handles`. Visuals are just additional graphical elements for better orientation of the user and for displaying the current state of the manipulator. For example the lines along the edges of a bounding box are implemented as visuals. The dragger administrates its visuals including creation, deletion and changing appearance for different modes of the manipulator, e.g. bounding box lines are hidden as long as the rotation mode is active.

For providing user interaction, each dragger additionally contains a set of handles. A `Handle` uses a visual representation — the widget — and contains the main logic of the manipulator core. It keeps track of all the states by implementing a state machine and performing appropriate state switches with respect to mouse or keyboard events. A handle creates its geometry required to represent the widgets for user interaction, e.g. small boxes at the corner of the bounding box that the user can drag for scaling. These objects are registered at the event distributor component of the event handling part (see Section 3.2) so the handle can receive the events intended for it.

The handle also performs the required computations for mapping 2D mouse movement into 3D-space. For this task it employs so called `Projector` objects. OpenManip provides three different types of projectors, the `LineProjector` for mapping movements along a line, the `CircleProjector` for movements on a circle and the `SphereProjector` for mapping movements on a sphere. The calculation itself includes the computation of the view ray through the mouse position mapped on the near clipping plane and the intersection of the ray with the analytical geometry of the projector. The returned intersection point is used as the start point for the transformation. When the mouse is moved, a second point is calculated and the mapping between these points with respect to the projector geometry is used to transform the object.

3.2. Event Handling

The second functional part of OpenManip's top layer is the event handling. The main task of this part is the distribution of events from the application to OpenManip. Therefore, OpenManip provides a component called `EventDistributor`. The event distributor receives all events from the application and checks whether an event is intended for one of the manipulators' handles. OpenManip utilizes its own event hierarchy and, therefore, requires a translator class presented in Figure 3, which maps application events, e.g. Qt events, to OpenManip events.

These OpenManip events are then passed to the `EventDistributor` which decides whether the received event affects some interactive objects of a manipulator. Therefore, the event distributor traverses a list of `Callback` objects, that the handles registered during their creation. A callback contains a pointer to an `EventListener`, which is an interface class for objects that are capable of receiving events and only provides a method to receive events. In our case each handle acts as an event listener.

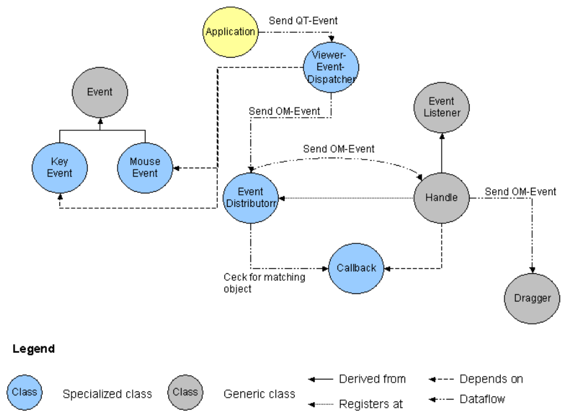

Figure 3: *The figure shows the event hierarchy and handling mechanisms of OpenManip. An event dispatcher receives application events and forwards them to the event distributor, which passes the events to a matching handle.*

In order to check whether a widget of a handle is affected, the event distributor performs a picking based on the 2D-coordinates of a mouse event and compares the resulting node with the widgets of all registered `Callback` objects. If a matching widget is found the event is passed to the event listener retrieved from the callback object. Once the event arrives at the handle, the appropriate state switches occur based on the type of event received. Eventually the event is propagated upwards to the handle's dragger, thus, it can also change its state and update the visuals belonging to the dragger.

If the event distributor does not find a matching node, a flag is set to report that no event handling occured within OpenManip, thus, the application can react accordingly and handle the event itself.

4. Scene Graph Independency

As stated before our framework was designed to be largely independent of a particular scene graph API, such that it can easily be ported. Therefore, our cross-scene-graph framework inserts a layer between the manipulator core, that handles user interaction, and the underlying scene graph library. This layer wraps the scene graph to provide an independent and stable interface for the top layer of OpenManip. Currently we have implemented scene graph drivers for OpenSG and Cosmo3D [7]. Figure 4 shows the layer concept in the context of an application based on the Qt library from Trolltech.

The design of the scene graph driver wraps the main functionality of the scene graph API into own classes which offer a common base functionality for the top layer of OpenManip. Each class wrap a specific part of the scene graph for ex-

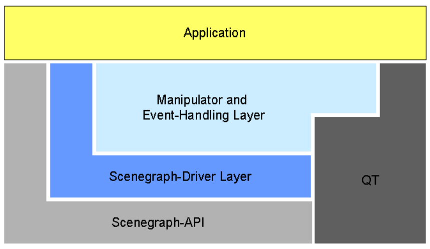

Figure 4: *The architecture of OpenManip is separated into two layers to minimize scene graph dependencies. A scene graph independent functional part can be combined with different scene graph drivers. Note that the presented application also uses the Qt widget set.*

ample a Transform node, Group node, or a specific geometry node. With this fine-grained separation we have been able to accommodate the fact that different scene graphs have different concepts on how they implement their leaves.

Whereas Cosmo3D uses a structure, where the leaf of the scene graph — a `csShape` node — has two non-node children, a `csGeometry` and a `csAppearance`, OpenSG uses a `OSGNode` with a `OSGGeometry` core that contains a pointer to the material. The latter provides a more consistent node concept. However, with our framework design we were able to hide these issues from the top layer of OpenManip.

On the other hand Cosmo3D provides basic geometry objects which are changeable in size and position. Since such objects are not available in OpenSG we had to simulate this functionality within our scene graph driver by using a subgraph of `OSGTransform` and `OSGGeometry` nodes, maintaining the stable interface to the top layer.

OpenManip manipulators are not implemented as a nodes, because of the already mentioned reasons to accommodate different scene graph concepts of different scene graph APIs. Manipulator nodes would have significantly enlarged the driver layer by transferring most of the functionality into the scene graph driver. We still would need the wrapping of basic nodes such as geometries, transforms and groups to create visuals and widgets for the manipulator. So there would have been no obvious advantage with an extra scene graph node.

Moreover the clear separation between manipulator functionality and scene graph driver greatly simplifies porting to new scene graph APIs. All which has to be done is to reimplement the classes of the scene graph driver with respect to the new scene graph API. No modifications are required in the manipulator core or the event handling. After implementing the new scene graph driver and adapting the application to the new scene graph, OpenManip works the same way as it does on the scene graphs for which a driver already exists. Basically one would reimplement the following groups of classes:

- Basic geometry wrapper classes
- Light wrapper classes
- Transform and group wrapper classes
- Scene graph wrapper classes

In order to ensure proper functioning of the OpenManip library the new classes minimally have to provide the corresponding functionality of existing driver classes.

5. Using OpenManip

Our OpenManip framework allows for easy integration of manipulator functionality into any application utilizing a scene graph API for which OpenManip provides a scene graph driver. Currently we support OpenSG and Cosmo3D. The integration only requires a few modifications of the application such as instantiation of several OpenManip main objects which act as an interface for the application. These objects are:

- `Scenegraph`
- `EventDistributor`
- `ViewerEventDispatcher`

Additionally the application has to keep a list of selected nodes and active manipulators. The list of selected nodes is used to create manipulators for these nodes, which also incorporates a selection manipulator, as we implemented the selection as a special case of a manipulator without any widgets.

The `Scenegraph` object is the basic object of the scene graph driver layer and requires some initial settings. First the scene graph object has to know the root node of the scene, as this will be used for picking, and the camera settings, as some scene graph APIs like Cosmo3D perform the picking based on the `csCamera` object. Additionally the `RenderAction` or `DrawAction` has to be set in the scene graph object as this is needed by OpenManip to trigger redraw of the scene if necessary. The view port of the application is also required for correct calculation of the mapping from 2D mouse coordinates to 3D-coordinates.

After this initialization the application is ready to create manipulators, though still an adjustment of the application's event handling is required. The application has to deliver the following list of events to the event distributor of OpenManip:

- Mouse move events
- (Left) mouse button click and release events
- Control key press and release events
- Shift key press and release events

The way this is realized depends on the GUI library the application is built from. For most GUI systems, in particular Motif, X or glut, this would require a modification

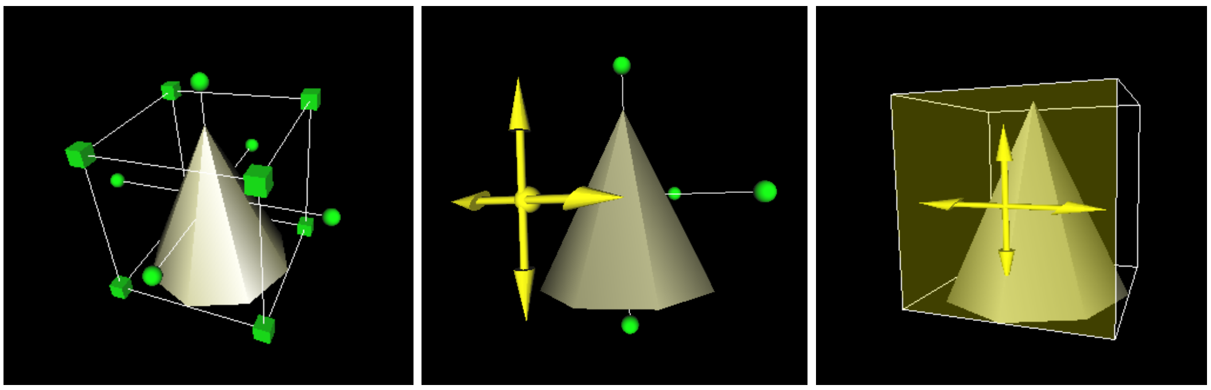

Figure 5: *The OpenManip* `TransformManipulator` *used to transform a cone. With activated rotation mode (middle image) it allows for the rotation perpendicular to the x- and y-axis. The translation mode (right image) allows for the translation within the highlighted plane.*

of the corresponding event callbacks. Since our application is built from the Qt widget set, we can exploit its signal-slot-mechanism here. We use a class `ViewerEventDispatcher`, which provides Qt slots for the required events. These slots have to be connected to the corresponding signals of the application.

The event dispatcher has to exclusively receive the Qt events, as long as a manipulator is active. The slots of the `ViewerEventDispatcher` gather the information from the Qt event and construct an appropriate OpenManip event, either of type `KeyEvent` or `MouseEvent`. The event is then sent to the `EventDistributor` object instantiated within the application.

Note that the application still has to deal with events not handled by OpenManip. With Motif or X callbacks this would normally be achieved by querying the flag, that indicates whether an event has been handled, after dispatching an event to OpenManip. If the flag is false the callback can proceed with the original callback code, otherwise it can directly return from the callback. In our Qt implementation we instead use a signal `noObject()` emitted by the `ViewerEventDispatcher` that has to be connected to a slot of the application.

OpenManip employs a clear object-oriented design for easy extension of the top layer of OpenManip. This extensibility allows to fit a manipulator to the developer's needs. Writing new manipulators and draggers can be accomplished by deriving new classes from the generic base classes presented in Figure 2. A new manipulator class only has to include code for creating an appropriate set of draggers and a new dragger code for creating handles and visuals. The implementation of visuals require additional code for changing their appearance with regard to the different states the dragger might be in. Extending handles require the most work, as this is the place where the behavior of the widgets is defined by the use of a state machine. Also the desired transformation has to be applied here in addition to the calculation of the orientation and the proper position of the widgets in 3D-space.

6. Results

In this section we present the results we achieved with OpenManip including screen shots of OpenManip in action. Currently OpenManip provides support for two scene graph APIs: OpenSG and Cosmo3D. We tested OpenManip in applications on both platforms. The Cosmo3D viewer was basically developed as a demonstration application for the OpenManip framework, whereas the OpenSG application is an independently developed viewer into which OpenManip was integrated. The integration only took a few days and comprised about 150 lines of code, whereas about 50 lines are related to menue items and callbacks.

Both applications are based on the Qt GUI library. Note that our framework could also be integrated with different GUI libraries, e.g. Motif, X, or glut, and is not restricted to Qt applications, although we added special support for Qt to the OpenManip library (See Section 5). We provide configure files and make files that allow for switching between the two scene graph APIs. In the following we list the features of OpenManip.

- A transform manipulator
- A selection manipulator
- Light manipulators (for directional light and point light)
- A material color editor
- A head light editor

OpenManip provides two manipulator types: the selection manipulator which is visualized as the bounding box of an object and the transform manipulator demonstrated in Figure 5. To activate the manipulators the user has to select the pick mode. When an object is picked a selection manipulator is inserted in the scene by default. An item in the manipulator menu can be used to replace the active selection manipulator by a transform manipulator, as it can be seen in the left image of Figure 5. The same item can also be used to change the default manipulator.

The widgets of the transform manipulator indicate the interaction mode by changing their color when the mouse cursor is moved over them. When the left mouse button is clicked on one of these widgets the color changes to yellow and the possible moving directions are displayed by arrows as can be seen in the middle image and right image of Figure 5.

The use of the SHIFT key offers additional functionality. These buttons can either be pressed before or after the mouse click. SHIFT restricts the movements along one axis in order to make positioning easier. To deactivate the manipulator the user has to click outside of the manipulator's area indicated by a bounding box or a bounding sphere.

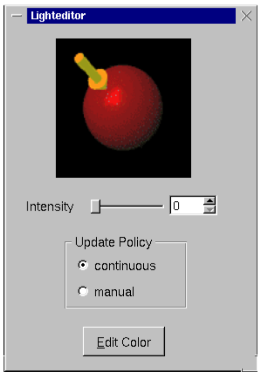

Figure 6: *A Qt-based editor for the manipulation of head light parameters.*

Figure 6 shows an editor for manipulating the head light parameters. The implementation is based on the Qt widget library. Note that the drawing area utilizes the underlying scene graph system in order to visualize and manipulate the head light direction. Our editor for manipulating material properties is shown in Figure 7.

7. Conclusion

We have presented an extensible cross-scene-graph framework that allows for the direct manipulation of scene graph objects using 3D widgets. Using our framework any existing application can easily be enhanced by manipulation functionality. The framework features different scene graph systems, in particular OpenSG and Cosmo3D and can be extended to future scene graph APIs with small effort. With our framework we have demonstrated two manipulators for scaling, rotation and translation of objects, and discussed customized manipulators, which are naturally supported by a clear object-oriented structure of the framework. Additionally two editors for manipulating light and material properties have been shown. Unfortunately up to now we did not have the opportunity to introduce OpenManip in a working environment, so that we lack results for the usability. This will be addressed by future work.

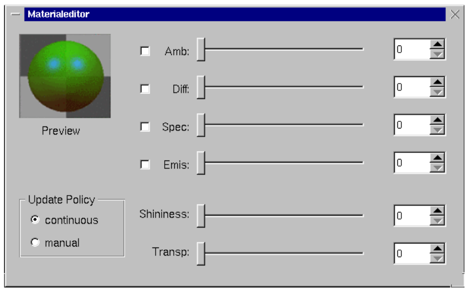

Figure 7: *The material editor allows for the specification of different material aspects. e.g. ambient, diffuse and specular color.*

References

1. Eric A. Bier. Snap-dragging in three dimensions. In *Proceedings of the 1990 symposium on Interactive 3D graphics*, pages 193–204. ACM Press, 1990. 1
2. Eric A. Bier and Maureen C. Stone. Snap-dragging. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 233–240. ACM Press, 1986. 1
3. D. Brookshire Conner, Scott S. Snibbe, Kenneth P. Herndon, Daniel C. Robbins, Robert C. Zeleznik, and Andries van Dam. Three-Dimensional Widgets. In *Proceedings of the 1992 Symposium on Interactive 3D Graphics, Special Issue of Computer Graphics, Vol. 26*, pages 183–188, 1992. 1
4. Matthias Kalle Dalheimer. *Programming with Qt*. O'Reilly, Cambridge, 1999. 3
5. Kenneth P. Herndon and Tom Meyer. 3D Widgets for Exploratory Scientific Visualization. In *ACM Symposium on User Interface Software and Technology*, pages 69–70, 1994. 1
6. Joe Kniss, Gordon Kindlmann, and Charles Hansen. Interactive Volume Rendering Using Multi-Dimensional Transfer Functions and Direct Manipulation Widgets. In *IEEE Visualization'01*, pages 255–262, October 2001. 1
7. SGI. *Cosmo3D Programming Guide, Version 1.2*. SGI, Online Dokumentation. 4
8. Trolltech. Qt C++ GUI Application Development Toolkit. http://doc.trolltech.com/. 3
9. Josie Wernecke. *The Inventor Mentor*. Addison-Wesley, Reading, MA, 1998. 2
10. Robert C. Zeleznik, Kenneth P. Herndon, Daniel C. Robbins, Nate Huang, Tom Meyer, Noah Parker, and John F. Hughes. An Interactive 3D Toolkit for Constructing 3D Widgets. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 81–84, 1993. 1