Interactive 3D Geometric Modelers with 2D UI

Wu, Shin - Ting **Marcelo de Gomensoro Malheiros**

Electrical and Computer Engineering Faculty
State University of Campinas
P.O.Box 6101 13083-970 Campinas, São Paulo, Brazil
ting@dca.fee.unicamp.br

ABSTRACT

This paper presents an object-oriented framework which enhances the collaboration of three categories of experts that play fundamental role in the development of a software for interactive 3D geometric modelers. First, it supports application developers to build a graphics interface for manipulating with 2D devices their own 3D data representations, without intimate knowledge of its internal structure. Second, it provides facilities for interface researchers to create and experiment 3D widgets from reusable draggers and 2D–3D mapping strategies. Finally, it permits graphics experts to implement sophisticated draggers and complex 2D-3D mapping strategies by overriding operations of the predefined abstract classes.

Keywords: Tools and toolkits, user interface design, interactive geometric modeling, interactive 3D graphics, 2D input/output devices

1 INTRODUCTION

Despite the availability of a large number of 3D graphics libraries, such as implementations of the GKS-3D and PHIGS standards, and the cross-platform hardware supported graphics library OpenGL[Neid93], writing interactive 3D graphics applications is still a tedious, time-consuming, and difficult task. According to Strauss and Carey[Strau92], the main problem of those libraries is that they provide little help for direct 3D interaction support, because they fall short of exact 3D shape representation.

To remedy this deficiency, Strauss and Carey proposed an integrated architecture that has matured into the Open Inventor toolkit[Werne94]. Also defending the integration of application and the user interface into the same development environment, Celes and Corson-Rickert implemented the ACT library[Celes97], and Conner et al. constructed three-dimensional widgets in the Unified Graphics Architecture (UGA)[Zelez91]. In their system each 3D shape can render itself and make geometric computation when necessary. Hence, the reaction to user input may benefit from techniques such as intersection, deformation, and simulation methods associated with sophisticated geometric shapes. More natural and precise semantic feedback can be provided while the user is interacting. This approach, on the other hand, may compromise the simplicity and the modularity of the resulting design, as observed by Schroeder et al.[Schro91], because the application developers must know the specific details of the framework's internal structure.

As developers of geometric modeling techniques, we would like to concentrate our effort on solving geometric problems. To evaluate visually a geometric modeling method or to manipulate its results, we would like to be free from implementing individual rendering and common direct 3D interaction tasks within the rendered scene. As there are tools for rendering 3D objects no matter how their particular representation is, we have striven for an interactive 3D environment where application-specific 3D models may be easily integrated without programming new subclasses or subsystems. This paradigm seems to be in contradiction with the semantically supported interaction concerns, which are intimately related with the underlying 3D model.

In this paper we present our solution for this contradiction. Due to the complexity of the problem, we solved it by considering three classes of experts that are essential for developing a 3D interactive geometric modeler: (1) the researchers in geometric modeling who look for a highly tailorable framework to accommodate their 3D modeling solutions, (2) the interface developers who require an environment for rapid prototyping, experimenting and integrating new interaction metaphors, and (3) the 3D graphics experts who are concerned with the 2D–3D mapping strategies and graphics presentation.

In the next section we present a framework, called **M**anipulation **T**ool**K**it, that enhances modularity by encapsulating 3D graphics facilities offered by a variety of graphics libraries, and extensibility by providing hook methods for new graphics objects and 2D–3D mapping algorithms. Then, we explain a framework for building interaction metaphors on top of MTK – **Fra**mework for **Ma**nipulator. Next, we show a design of a black-box framework on top of MTK and FaMa that allows us to integrate any 3D modeler by aggregation. Finally, some concluding remarks are drawn.

2 MTK: A GRAPHICS TOOLKIT

MTK was implemented as a C++ programming library and aimed at providing a simple, direct interface to the fundamental operations of 3D graphics rendering and interaction.

Despite their limited graphics primitives, the successful use of OpenGL in Geomview[Geomview] for inspecting and evaluating a wide range of geometric representations has driven us to include a subset of the OpenGL functionalities in MTK, namely the set of basic geometric objects, display list (wrapped by the `Geometric Models` class), viewing (wrapped by the `Cameras` class), lighting (wrapped by the `Lights` class), and picking/selection (wrapped by the `Selection` class). In this way, for visualization purposes, the coupling of an application-specific 3D model and MTK may be reduced to a data format conversion problem.

OpenGL, as a variety of graphics libraries, provides very little support for interaction beyond simple picking and/or selection mechanism. Therefore, in our design we focused on 3D interaction support. Bearing reusability and orthogonality in mind, we added in MTK interaction facilities: (1) a 3D-cursor for indicating visually a virtual 3D-mouse position, (2) a set of pictorial representations to improve the 3D depth perception, (3) a set of basic 3D interaction tasks, (4) a set of pictorial representations that are relevant to realize 3D interaction metaphors, and (5) a way to customize the relationship between a geometric element and a 3D interaction metaphor.

According to Foley et al.[Foley90], there are four basic interaction tasks: positioning, selecting, entering text, and entering numeric quantities. Moreover, they also stated that, with 2D interaction devices, 2D and 3D applications only differ strongly in the positioning and selecting tasks. It is because of perceiving 3D depth and ambiguous 2D into 3D mapping. With good rendering techniques, not only the picking/selection mechanisms can be efficiently implemented, but also the 3D depth perception problem can almost be solved as well. Even though, we devised the `Guides` class for providing additional visual aids when necessary. The remaining challenging issue is to perform 3D positioning with 2D devices, which is reduceable to a 2D–3D mapping problem.

The projection mapping is surjective, where a 2D device point (pixel) correspond to a infinite number of 3D points. This ambiguity may be solved by specifying the surface of interest and only considering the visible side of this surface. This surface is an additional constraint that we need to determine unambiguously the "unprojected" 3D point. Therefore, we defined the `Constraints` class in MTK. As the constraints are highly dependent on the geometric power of the underlying graphics package, explicit hook methods are provided to support variability. In the current implementation of MTK there are four types of constraining geometry: line, plane, sphere, and application-dependent geometry.

The application-dependent geometry is useful in moving controllably a virtual 3D mouse on the surface of any object in 3D scenes with a 2D-mouse. The position of the 2D-mouse is "unprojected" in 3D space through an implicit function that defines the surface. A 3D-mouse can also move freely in a 3D scene. In this case, we constrained the 2D-mouse movements on either xy or yz planes in R^3 according with the predefined operation mode. In our implementation, the position of the 3D-mouse is visually represented on the screen by the conventional 2D-cursor in crosshair shape. All these methods for managing a virtual 3D-mouse are encapsulated in a `3D Cursor` object. Figure 1 shows a 3D-cursor on the (a) outside and (b) inside of an object in a 3D scene.

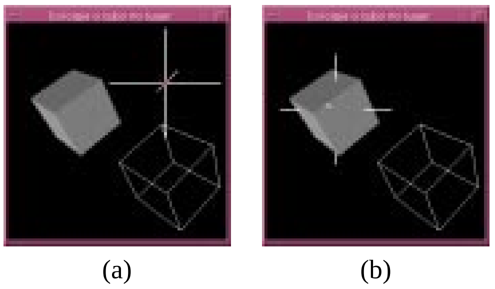
(a) (b)

Figure 1: 3D Cursor

Using pictorial representations to convey more complex manipulations, such as translation and rotation, is a common practice in 2D interfaces. These representations are, in general, designed such that actions on them are equivalent to the ones that a user performs in her/his concrete world. We call these pictorial representations **draggers**. Because of the successful use of these elements in making user interfaces more attractive and accessible, it is convenient to include the `Draggers` class in MTK. To enhance extensibility, new subclasses may be derived by the inheritance mechanism.

Both `Draggers` and `Graphics Models` objects are selectable when a user picks on their elements – `Handles`/(Sensitive) `Parts` and `MtkElement` objects, respectively – and respond by delegating the request to application-dependent objects for handling them properly. They have also capabilities for redrawing by themselves when their attributes are changed. At this point, one may argue what is the difference between them. The difference lies in the way that they handle the subsequent input events. Predefined subsequent 2D positioning events are captured by a dragger and transformed into 3D points. Then, the point is passed to the client for further processing. Whereas a geometric model is passive, in the sense that it just passes the event to the client for specialized dealing. The unambiguous 2D–3D mapping in a dragger is achieved by using a `Constraints` object to which each sensitive part must refer. Currently, three subclasses of `Draggers` are implemented:

- box: with a box geometrical representation. It comprises twenty-six sensitive parts: eight vertices, twelve edges, and six faces (Figure 2a).
- sphere: with a spherical geometrical representation. It comprises six sensitive parts: the sphere surface, three orthogonal rings, a handle, and the handle vertex (Figure 2b).
- reference frame: with a three-orthogonal-axis geometrical representation. It comprises thirteen sensitive parts: six edges and seven vertices (Figure 2c).

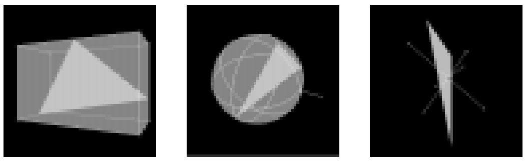
(a) (b) (c)

Figure 2: Draggers

Summarizing, MTK is comprised of the `Mtk-Core` class and eight "independent" abstract classes: `Graphics Models`, `Cameras`, `Lights`, `Selection`, `Guides`, `Constraints`, `Draggers`, and `3D Cursor` (Figure 3). The `MtkCore` coordinates the interaction between these classes by delegating requests in order to realize a semantic action. For example, the `Selection` passes to the `MtkCore` the identifier of selected elements and the method `cameraPointer()` in `MtkCore` is responsible for deciding whether a part of a dragger or a mtkElement should be notified to handle the subsequent events. Another important role of `MtkCore` is in 2D–3D mapping, which, besides the 2D position and geometrical constraint, requires the viewing parameters encapsulated in the `Cameras` object.

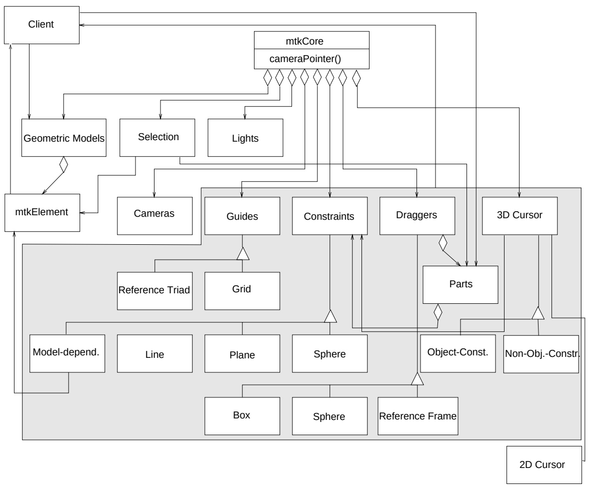

Figure 3: MTK framework

In comparison with the existing graphics libraries, four new classes, `Guides`, `Constraints`, `3D Cursor`, and `Draggers`, are included in MTK to enhance 3D interaction support. Moreover, instead of being self-triggered in response to the user actions, the visual feedback of geometric models and draggers is triggered in response to the attribute change determined by the client. In the next section, we will show that conceptually this approach improves its reusability and flexibility in building new 3D metaphors.

In principle, MTK may be developed on top of any 3D graphics libraries and windowing system. In our case, it is realized on the top of OpenGL and GTK. But for enhancing its portability, we may define the `CoreFactory` object which abstracts the process of manufacturing graphics elements on the screen space.

3 FaMa: A Framework for Manipulators

In this section we present a framework on top of MTK that simplifies the construction of a 3D graphics metaphor.

Metaphors play an important role in helping users to understand the actions that they should perform in order to correctly interact with computer systems. A metaphor may be implemented as a widget, which encapsulates geometry and behavior used to control or display information about application-dependent data. In our project we are particularly interested in graphics 3D widgets, which permits a virtual 3D cursor acting on the shape or the position of a 3D graphics object exactly as the user finger would operate a concrete one. Hence, a nice visual feedback may be provided while a user is interacting with the system. We call these 3D widgets **manipulators**.

Popular 3D manipulators, such as virtual sphere[Chen88], triad[Niels86], skitter and jack[Bier90], and tri-axes[Emmer90], were constructed on the basis of this principle. The virtual sphere restricts the cursor movements on an imaginary sphere and two subsequent input points are used to specify a rotation angle that is applied both in the virtual sphere and in the object of interest. Consequently, the user has very clear interpretation of the performed action. Triad imposes that the cursor moves in the directions of the local coordinate system of the selected object and two subsequent input points define a 3D displacement vector. Both the triad and the 3D object of interest are transformed and updated to convey the operation. Skitter/jack and tri-axis are variations of triads.

The common feature of the operations that these metaphors represent is that they are decomposable into a sequence of points in 3D space. They differ from each other in (1) the way that a virtual 3D-mouse should move in 3D space to track the concrete movement of a 2D-mouse, (2) the way that a sequence of positions of the virtual 3D-mouse is mapped onto a domain-specific concept, and (3) the visual feedback that they should provide.

To ensure that a virtual 3D mouse follows appropriately the movement of a concrete 2D cursor on the screen we must solve a 2D–3D mapping problem. Its solution is already supported by MTK, by instantiating conveniently a constraint method in each sensitive part of a dragger. With this support the researchers of 3D metaphors are free from the low-level 2D input problem and may consider 3D positions as atomic inputs.

The extraction of a magnitude of the action that a manipulator represents, such as angle and displacement value, from a sequence of 3D positions is a geometric problem. For example, two distinct points on a line define a vector which may convey displacement and two distinct points on an sphere determine an angle. With the use of MTK, a manipulator should simply *contain* a set of draggers for receiving 3D positions and transform them into meaningful values by a set of mapping algorithms. What distinguishes one manipulator from others is the supplied mapping methods.

Normally, the collection of primitives provided by a geometric modeler is much more specialized than the ones provided by conventional graphics toolkits. This difference may yield discrepancies between what you see and what you get, if the updates of the application and graphics data are not appropriately coordinated. In our work we adopted the paradigm "what you see is a simplified form of what you get" to solve the almost paradoxal problem – tightly application dependent visual feedback with an application-graphics decoupled system. Our manipulator indeed encapsulates the interaction between an application and graphics libraries and provides a harmonious and meaningful visual feedback. The manipulators only supply meaningful values and delegate the visual changing to "experts" by providing them values to be changed. In our case, a client is the expert responsible for application data and, hence, for their conversion to the format required by MTK. And the visual appearance of draggers is managed by MTK itself.

We defined the `MtkManipulator` class, which keeps reference to the two loosely coupling objects and mediates the communication between them when an input event occurs (Figure 4). In our implementation, new subclasses may be created by overriding semantic mapping actions (behavior).

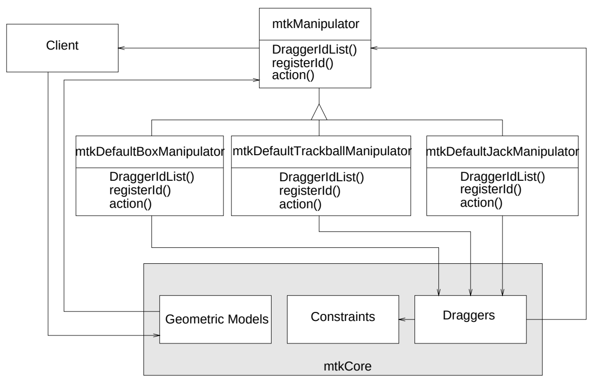

Figure 4: FaMa Framework

When an event is captured by a dragger, a manipulator is notified with a 3D position. It, in its turn, not only transforms the captured sequences of points into semantic values but also passes them to both MTK and the client. For consistent visual feedback, the client should further forward a redrawing request to MTK after updating and converting its internal data. To enhance flexibility, it is up to the client to choose the manipulator to be used for interacting with each object. Figure 5 shows the interaction diagram between these objects.

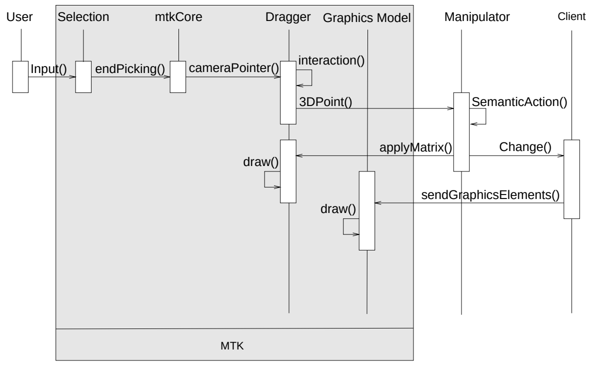

Figure 5: Sequence diagram for a metaphor

It is worth noting that the `MtkManipulator` object communicates with the `Graphics Models` object only indirectly, through its client. One may criticize the efficiency of this "duplicate database" approach. But, with most of graphics libraries designed to provide maximum access to hardware graphics capabilities and the trend towards groupware technology, this approach may favor the migration of a single-user 3D modeling system to a multi-user platform without 3D model inconsistency problems due to the differential round-off error propagation in each participating machine. Besides, it is one of the main results that we achieved towards the black-box reuse of the framework for building 3D interactive graphics modeling systems, as detailed in the next section.

Comparing with the concepts implemented in Open Inventor, our manipulator actually explores combined features of the manipulator and engine classes of Open Inventor to ensure the separability between an application and MTK data. It is similar with Open Inventor, in the sense that a manipulator *contains* at least one dragger which supplies geometry and user interface to perform a composite interaction task. However, instead of being subordinated to specific classes (e.g. transformation and light), our manipulator acts as an engine by connecting the values received from a dragger with the values needed by a client.

In the UGA development environment[Conne92] the manipulators are also tightly integrated with an application. According to the authors, this strategy "lets the application provide semantic feedback while the user is interacting". In fact, the metaphors are handled indistinguishable from the application objects, so they may benefit from "the application's power for specifying behavior and geometry". Instead, the behavior of our dragger and application data may be visually different, due to the limited geometric operations provided by MTK. But, the set of racks developed by Snibbe et al.[Snibb92] induces us to believe that, when carefully designed, simple geometric transformations on a dragger may indicate complex actions to be performed on the application-dependent data.

Let us give an example that illustrates graphically a sequence of actions that is performed internally while a user is interacting with an application-dependent 3D object. We consider that there is a sphere manipulable by a bounding box manipulator (Figure 6):

1. The modeler creates a sphere by instantiating a mtkElement. An instance of the bounding box manipulator is created and associated with it.
2. When the user clicks on the sphere, the manipulator is invoked and it inserts a box dragger around the sphere as a visual feedback.
3. When the user clicks on a vertex and drags the 2D cursor, MTK will send two subsequent captured points to the manipulator.
4. The manipulator computes the scaling factor and sends it to its box dragger for updating.
5. The manipulator sends the scaling factor to the application for updating.

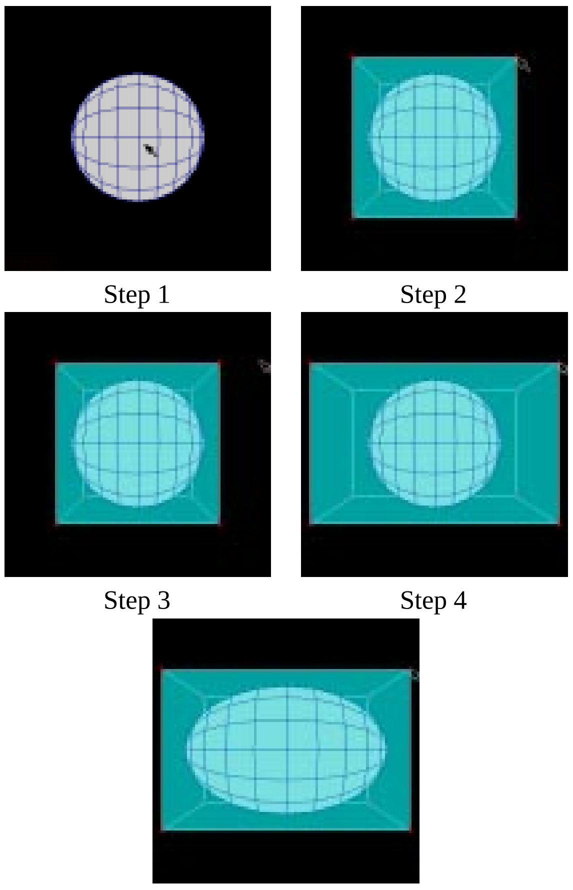

Figure 6: Manipulation with a Manipulator.

It is worth noting that the visual feedback of draggers and geometric models is totally decoupled. To evaluate our framework, three well-known 3D manipulation metaphors – `MtkDefaultBoxManipulator`, `MtkDefaultTrackballManipulator`, and `MtkDefaultJackManipulator` – were implemented by subclassing the `MtkManipulator` class.

4 A FRAMEWORK FOR INTERACTIVE 3D MODELERS

In this section we present, on top of MTK and FaMa, a framework that is easily tailorable to the requirements of a particular interactive 3D modeler without knowing their internal details. In this layer, MTK and FaMa are mere toolkits providing rendering facilities and 3D interaction metaphors, respectively. A 3D modeler drives their behavior, since only it knows the application context, the required graphics language precision, and the semantically supported units of information in the decoupled system.

The application decides which visible data are manipulable with which operations. Hence, it is responsible for, besides conventional viewing and lighting parameters, instantiating mtkElements and customizing the metaphors that a mtkElement is associated with. After then, MTK and FaMa take over all managing actions relative to input events and visual feedback of the metaphors. The application only affects the visual appearance when it receives a request from a manipulator to modify its data and update the corresponding graphics representation in MTK. An application should, basically, create three kinds of instances at the beginning of a work session: scene parameters, mtkElements and the metaphors that they need. During the work session it should ensure that mtkElements track user actions.

We designed the `Application3DGraphics` class to interface a 3D modeler with the framework comprising of `MtkCore` and the `MtkManipulator` classes (Figure 7).

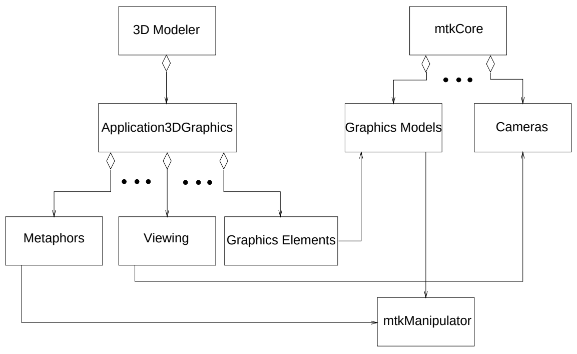

Figure 7: A Framework for 3D Modeler

The collaboration of a 3D modeler with MTK through manipulators is sketched in Figure 8.

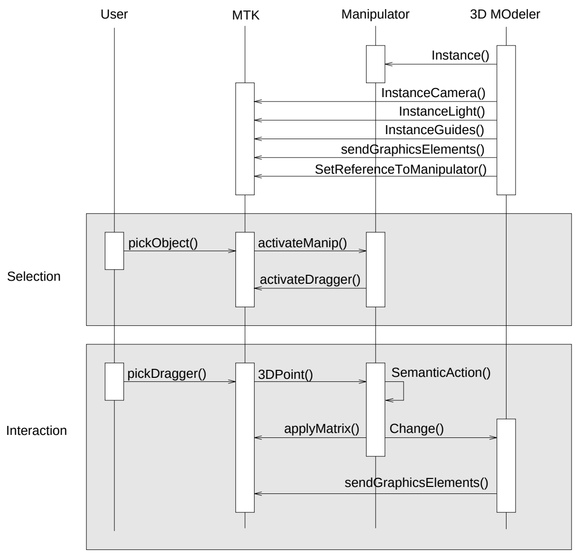

Figure 8: Sequence diagram for a modeler

To validate the reusability of this framework, three custom geometric modelers with completely distinct underlying data representation and geometric algorithms were derived.

4.1 Triangle modeler

This geometric modeler has been used as the testbed for exploring the potential of the available draggers and constraints in implementing various 3D metaphors already proposed. It provides three basic transformations – translation, rotation, and scaling – and can handle an arbitrary number of triangles (Figure 9).

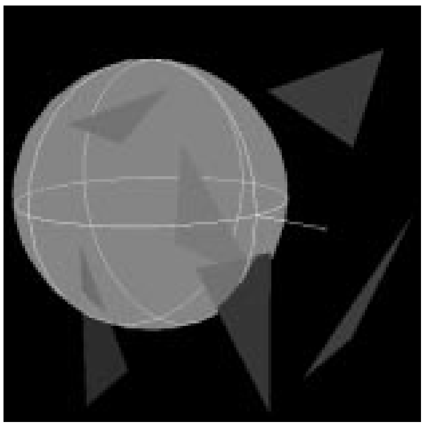

Figure 9: Triangle modeler

We used the proposed framework to implement a simple user interface which allows the user to visualize and to apply any of the three available transformations on a triangle. Our programming effort was restricted to define triangles as selectable mtkElements and its association with one of the currently available manipulators. In addition, we needed to register in each manipulator a reference to the application's handlers for delivering correctly updating requests.

4.2 Boundary based modeler

A simple and intuitive geometric modeler was also implemented. An instantiation mechanism is used to create a new object and a boundary representation[Morte85] is chosen to describe internally the object data. Geometric transformations (translation, rotation, and scaling) and local vertex manipulations (repositioning) are supported. These transformations are applicable on an application object or on a group of them.

For this modeler we decided to let the user choose the preferable metaphor – bounding box, trackball, or jack – for specifying interactively the transformation parameters. One can configure the preferences through a menu. For repositioning the selectable vertex, the jack manipulator is used. To implement this decision, we define each instantiated primitive (a set of polygons) and their vertices as selectable mtkElements and set in the manipulators references to the modeler's handlers for passing to them requested semantic values. Figure 10 exemplifies the reshape of a Bézier surface through its control point.

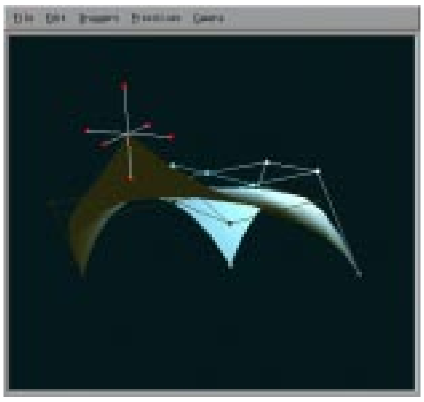

Figure 10: Manipulation through a control point

Figure 11 depicts the use of a shaded, transparent bounding box manipulator to rotate a group constituted by two objects – a torus and a cube. It is interesting to observe that, to avoid a dragger obscuring the application objects or unnecessarily burdening the scene, transparency was used.

4.3 Implicit modeler

The most interesting experiment was to develop with our framework a user interface to an implicit geometric modeler developed by our group[Malhe97, Wu99]. In the current version, an object is implicitly described as a combination of a set of spheres. By applying different geometric transformations (rotation, translation, and non-uniform scaling) on these spheres, a variety of objects can be produced.

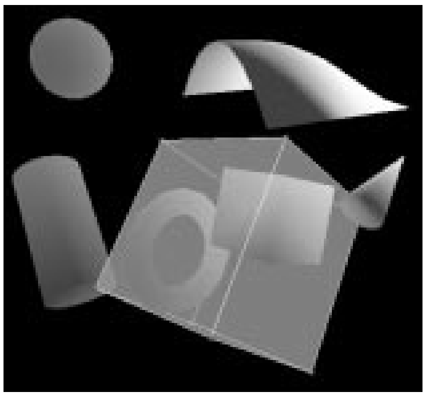

Figure 11: Boundary-based modeler

The requirement was to provide a metaphor for interactively combining and modifying the shape of the spheres in order to design complex models. From previous experiences, we opted for the bounding box manipulator. We should also decide how to visualize the modeler object with MTK, once they have completely different data structures. Two attempts were carried out. Firstly, we sampled a set of points on the object and defined it as a selectable mtkElement. We missed, however, the object's topology. The second try was to convert the implicit representation into a polygonal mesh and to pass it as a selectable mtkElement to MTK. In both cases we could then easily associate with each sphere a bounding box manipulator, either for repositioning or for reshaping.

Figure 12 elucidates the use of a "wire-framed" bounding box manipulator to manipulate a sphere primitive in order to reshape an implicit object resulting from the blending of two spheres.

5 CONCLUDING REMARKS

In this paper we presented a layered framework for graphics interactive 3D modelers. The clean separation between the three layers – MTK, FaMa and 3D modelers – gives a good support to different classes of researchers involved in the development of a graphics interactive 3D modeling system, namely the application, interface and graphics developers.

The framework is implemented in C++. To validate our concept, some constraints and draggers were implemented in the first layer – MTK. They were

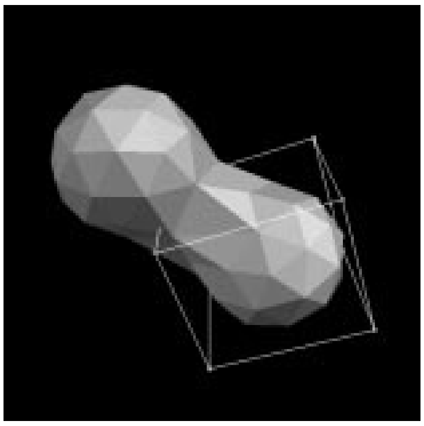

Figure 12: Implicit modeler

used in the second layer – FaMa – for constructing three subclasses of manipulators without demanding the knowledge of internal organization of MTK. Only references to constraints and draggers were necessary. Finally, three custom interactive modelers were developed in the third layer without requiring the knowledge of the underlying architecture of MTK and FaMa.

Our experimentation and evaluation let us conclude that our framework may be considered as a solution towards one of the challenging problems in the interface design – conciliation of modularity (of systems) and diversity (in geometric representations). Currently, we are moving the triangle modeler from a single-user to a multi-user platform to test the responsiveness of the system over a network.

6 Acknowledgments

We would like to acknowledge FAPESP for financial support under the grant number 96/0962-0.

REFERENCES

[Bier90] E.A. Bier. Snap-dragging in three dimensions. *Proc. of 1990 Symposium on Interactive 3D Graphics*, 193–204, March 1990.

[Celes97] W. Celes and J. Corson-Rikert. Act: an easy-to-use dynamically extensible 3D graphics library. *Proc. of SIBGRAPI '97*, 26–33, October 1997.

[Chen88] M. Chen and J. Mountford. A study in interactive 3D rotation using 2D control devices. *Computer Graphics*, 22:121–129, August 1988.

[Conne92] D.B. Conner, S.S. Snibbe, K.P. Herndon, D.C. Robbins, R.C. Zeleznik, and A. van Dam. Three-Dimensional Widgets. *Computer Graphics*, 25(2):183–188, March 1992.

[Emmer90] M. van Emmerik. A direct manipulation technique for specifying 3D object transformations with a 2D input device. *Computer Graphics Forum*, 9:355–361, 1990.

[Foley90] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, 1990.

[Geomview] Geomview: 3D Visualization Software. The Geometry Center, `http://www.geom.umn.edu/`.

[Malhe97] M. de G. Malheiros, and S.-T. Wu. Hierarchical skeleton-based implicit modeling. *Proc. of SIBGRAPI '97*, 65–70, October 1997.

[Morte85] M.E. Mortenson. *Geometric Modeling.* John Wiley & Sons, 1985.

[Neid93] J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, release 1.* Addison-Wesley, 1993.

[Niels86] G.M. Nielson and D.R. Olsen Jr. Direct manipulation techniques of 3D objects using 2D locator devices. *Proc. of 1986 Workshop on Interactive 3D Graphics*, 175–182, October 1986.

[Schro91] W.J. Schroeder, K.M. Martin, and W.E. Lorensen. The design and implementation of an object-oriented toolkit for 3d graphics and visualization. *Proc. of Visualization '96*, 93–100, November 1991.

[Snibb92] S.S. Snibbe, K.P. Herndon, D.C. Robbins, D.B. Conner, and A. van Dam. Using deformations to explore 3D widget design. *Computer Graphics*, 26(2):351–352, July 1992.

[Strau92] P.S. Strauss and R. Carey. An Object-Oriented 3D Graphics Toolkit. *Computer Graphics*, 26(2):341–347, July 1992.

[Werne94] J. Wernecke. *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor.* Addison-Wesley, 1994.

[Wu99] S.-T. Wu, and M. de G. Malheiros. On Improving the Search for Critical Points of Implicit Functions. *IS99: The Fourth International Workshop in Implicit Surface*, September 1999.

[Zelez91] R.C. Zeleznik, D.B. Conner, M.M. Wloka, D.G. Aliaga, N.T. Huang, P.M. Hubbard, B. Knep, H. Kaufman, J.F. Hughes, and A. van Dam. An object-oriented framework for the integration of interactive animation techniques. *Computer Graphics*, 25(4):105–111, July 1991.