December 1, 1994

SRC Research Report

129

Obliq-3D
Tutorial and Reference Manual

Marc A. Najork

digital

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

The charter of SRC is to advance both the state of knowledge and the state of the art in computer systems. From our establishment in 1984, we have performed basic and applied research to support Digital's business objectives. Our current work includes exploring distributed personal computing on multiple platforms, networking, programming technology, system modelling and management techniques, and selected applications.

Our strategy is to test the technical and practical value of our ideas by building hardware and software prototypes and using them as daily tools. Interesting systems are too complex to be evaluated solely in the abstract; extended use allows us to investigate their properties in depth. This experience is useful in the short term in refining our designs, and invaluable in the long term in advancing our knowledge. Most of the major advances in information systems have come through this strategy, including personal computing, distributed systems, and the Internet.

We also perform complementary work of a more mathematical flavor. Some of it is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. Other work explores new ground motivated by problems that arise in our systems research.

We have a strong commitment to communicating our results; exposing and testing our ideas in the research and development communities leads to improved understanding. Our research report series supplements publication in professional journals and conferences. We seek users for our prototype systems among those with whom we have common interests, and we encourage collaboration with university researchers.

Robert W. Taylor, Director

Obliq-3D
Tutorial and Reference Manual

Marc A. Najork

December 1, 1994

Abstract

Obliq-3D is an interpreted language that is embedded into the 3D animation system Anim3D. Anim3D is based on a few simple, yet powerful constructs that allow a programmer to describe three-dimensional scenes and animations of such scenes. Obliq-3D, by virtue of its interpretive nature, provides the programmer with a fast turnaround environment. The combination of simplicity and fast turnaround allows application programmers to construct non-trivial animations quickly and easily.

The first half of this report contains a tutorial to Obliq-3D, which develops the various concepts of the animation system. The second part contains a reference manual, which describes the functionality of Obliq-3D module by module.

Contents

1 Introduction **1**
1.1 Related Work 1

2 Tutorial **4**
2.1 Obliq-3D in a Nutshell 4
2.2 A First Example 4
2.3 Properties 6
2.4 The On-Line Help Facility 11
2.5 Non-Tree Scene Graphs 11
2.6 More on Property Values 13
2.7 Overloading 15
2.8 Behaviors 16
2.9 Asynchronous Behaviors 17
2.10 Synchronous Behaviors 18
2.11 Dependent Behaviors 20
2.12 Locking 21
2.13 Callbacks 24
2.14 Extending Objects 26
2.15 Naming Graphical Objects 28
2.16 Requests 29
2.17 Depth Cueing 32
2.18 Light Objects 33
2.19 Camera Objects 34
2.20 Creating Root Objects 38

3 Reference Manual **41**
3.1 The Point3 Module 44
3.2 The Matrix4 Module 45
3.3 The Anim3D Module 46
3.4 The ProxiedObj Module 46
3.5 The GraphicsBase Module 47
3.6 The X‘Pex‘Base Module 47
3.7 The AnimHandle Module 48
3.8 The GO Module 49
3.9 The GroupGO Module 54
3.10 The RootGO Module 55
3.11 The LightGO Module 57

3.12 The AmbientLightGO Module 57
3.13 The VectorLightGO Module 58
3.14 The PointLightGO Module 59
3.15 The SpotLightGO Module 61
3.16 The CameraGO Module 63
3.17 The OrthoCameraGO Module 65
3.18 The PerspCameraGO Module 66
3.19 The LineGO Module 68
3.20 The MarkerGO Module 69
3.21 The SurfaceGO Module 70
3.22 The PolygonGO Module 72
3.23 The BoxGO Module 73
3.24 The DiskGO Module 74
3.25 The SphereGO Module 75
3.26 The CylinderGO Module 76
3.27 The ConeGO Module 77
3.28 The TorusGO Module 78
3.29 The QuadMeshGO Module 79
3.30 The Prop Module 80
3.31 The BooleanProp Module 82
3.32 The RealProp Module 85
3.33 The PointProp Module 86
3.34 The ColorProp Module 88
3.35 The TransformProp Module 90
3.36 The LineTypeProp Module 93
3.37 The MarkerTypeProp Module 94
3.38 The RasterModeProp Module 95
3.39 The ShadingProp Module 96
3.40 The MouseCB Module 97
3.41 The PositionCB Module 98
3.42 The KeyCB Module 99

Acknowledgments **100**

References **101**

Index **105**

1 Introduction

Obliq [3] is a lexically-scoped, untyped, interpreted language that supports distributed object-oriented computation. The Obliq interpreter is written in Modula-3 [17], and is designed to be both easy to extend and easy to embed into other programs.

Anim3D is a Modula-3 library for building programs that use interactive, animated three-dimensional graphics. It uses the X Window System [21] as the underlying window system, and MPEX (Digital's extension of PEX [8], the 3D extension of X) as the underlying graphics library. Support for other window systems (e.g. Trestle [14, 15]) and graphics libraries (e.g. OpenGL [18]) is under way.

Anim3D is designed to interconnect easily with an embedded language, although it does not rely on one. Possible choices for such a language are Lisp, Python, or any other interpreted language that can be easily embedded. We chose Obliq, because its linguistic features closely match those of Modula-3, and because it is particularly easy to embed an Obliq interpreter into some other Modula-3 program. The resulting embedded language is called Obliq-3D.

Obliq-3D is a superset of Obliq; it provides the same syntactic constructs, as well as all the built-in libraries described in SRC Research Report 122 [3]. In addition, it provides 42 extra modules that support the construction of 3D animations.

This report is divided into two parts. The first part gives an example-driven introduction to the basic concepts of the animation system. It assumes the reader is familiar with SRC Report 122, but does not make any further assumptions. The second part contains a reference manual for Obliq-3D. The manual contains a section for each module that is specific to Obliq-3D.

Our initial motivation for developing a fast-turnaround 3D animation system was our experience in using 3D views for algorithm animation [2]. We found that building enlightening 3D animations, using a low-level graphics library such as PEXlib, was both hard and time-consuming. Since then, we have used Obliq-3D for building a variety of algorithm animations. A detailed development of one such animation (Dijkstra's shortest-path algorithm) can be found in [16]; the accompanying videotape shows a collection of animations done with Obliq-3D.

1.1 Related Work

The last 20 years have seen a variety of device-independent 3D graphics libraries, such as Core [9], GKS-3D [11], PHIGS [1] and PHIGS+ [19]. Today's most popular "traditional" 3D graphics libraries are PEXlib [8], IRIS GL [22], and its OpenGL variant [18].

The potential benefits of applying object-oriented methods to computer graphics have been pointed out since the inception of the object-oriented paradigm. Some of the earlier object-oriented libraries, such as HOOPS [27] and Doré [13], use an object-oriented design model, but provide no interface to an object-oriented language.

More recent systems, such as Grams [6], Inventor [24, 25, 26], and GROOP [12], are written in C++, and provide a C++ application programmers' interface. Inventor in particular has been quite influential. Obliq-3D adopts some of its key ideas: Scenes are modeled as directed acyclic graphs of graphical objects, and geometric primitives, cameras, and light sources are treated uniformly. But there are also some important differences. In Inventor, shapes and properties can both be added as nodes to the scene tree. This means that the order in which nodes are inserted into the tree affects the appearance of the scene: Inserting first a color node and then a sphere node will produce a different image than that obtained by adding first the sphere and then the color. Another key difference is that Inventor properties are not inherently time-variant. Animation is achieved through *engines*, which change the state of property nodes and shape nodes. Finally, Inventor lacks an interpreted language, a feature crucial to rapid prototyping of animations.

Paul Strauss' "Brown Animation Generation System" [23], or BAGS for short, is one of the first 3D animation systems to provide such an embedded interpreted language (ironically, Strauss later on designed Inventor). BAGS uses an interpreted language called SCEFO to describe the structure and the animation behavior of a scene. Although quite powerful, SCEFO is an animation language, not a full-fledged programming language. It provides assignment and iteration constructs, a set of arithmetic functions, and procedural abstraction, but no conditionals. However, BAGS allows the user to write C code stubs that can be compiled into the system and can then be called from SCEFO.

BAGS was succeeded by UGA, the "Unified Graphics Architecture" [28, 10]. UGA uses an interpreted language called FLESH that is based on the prototype-and-delegation paradigm. Scenes are modeled as collections of objects. Attached to each object is a list of "change operators" or *chops*. Each chop defines some aspect of the appearance of an object, such as its color, transformation, and even its shape. Moreover, chops are functions of time; they are the basis for animation in UGA.

Alice [4, 5] is another rapid prototyping system for creating 3D animations. It uses Python as its embedded interpreted language. There are many commonalities between Alice and Obliq-3D. Both systems use an object-oriented interpreted language to allow for rapid prototyping. Both separate the application from the rendering. (Alice uses a process to perform the rendering; Obliq-3D uses a separate animation server thread.) Both systems allow for hierarchical geometry, that is, graphical objects that contain other graphical objects. However, the two systems differ in their basic animation model: Alice treats a scene as a hierarchy of graphical objects, each of which has a list of *action routines* which are called each frame of the simulation, and which are responsible for changing the internal state of the object. New animation effects are created by defining new action routines and adding them to the graphical object. Obliq-3D, on the other hand, uses *properties* to specify the appearance of objects; these properties are inherenty time-variant. In other words, Alice performs frame-based animation, whereas Obliq-3D has an explicit notion of time.

Finally, TBAG [7, 20] is a very recent toolkit for rapid prototyping of interactive 3D animations. TBAG provides two programmer interfaces: an interpreted functional language, intended to allow casual users to utilize the existing functionality of the system; and a compiled, object-oriented language, namely C++, to allow more sophisticated users to extend the system. On the surface, TBAG and Obliq-3D look different, mainly due to the different paradigms underlying their embedded languages. Closer examination, however, uncovers a fair number of similarities: Both systems distinguish between values that define geometry and values that define attributes (such as color); both allow the user to impose a hierarchical structure on the geometry of the scene; both systems treat light sources like any other geometric primitive; both systems have the notion of time-varying values; and both support constraints (however, TBAG allows for a more general class of constraints than Obliq-3D).

2 Tutorial

The tutorial section of this report develops the fundamental concepts of Obliq-3D, using a series of gradated examples. We start with the smallest program that actually generates a 3D image (Obliq-3D's equivalent of "Hello World!"), and we work our way up to a level where we show how parts of the functionality of the system could be reimplemented.

The reader should be able to work his way through the tutorial within the course of a morning. All the examples are self-contained and short enough to be tried out during this reading[1].

2.1 Obliq-3D in a Nutshell

Obliq-3D (and Anim3D) is founded on three basic concepts: *graphical objects* for constructing scenes, time-variant *properties* for animating various aspects of a scene, and *callbacks* for providing interactive behavior.

Graphical objects subsume geometric shapes (spheres, cones, cylinders, and the like), light sources, cameras, *groups* for composing complex graphical objects out of simpler ones, and *roots* for displaying graphical objects on the screen.

Properties describe attributes of graphical objects, such as their color, size, location, or orientation. A property consists of a *name* that determines what attribute is affected, and a *value* that determines how it is affected. Property values are not simply scalar values, but rather functions that take a time and return a scalar value. Thus, property values form the basis for animation.

Callbacks can be attached to graphical objects; they define how these objects react to events such as key strokes or mouse position changes. Callbacks form the basis for interactive behavior.

2.2 A First Example

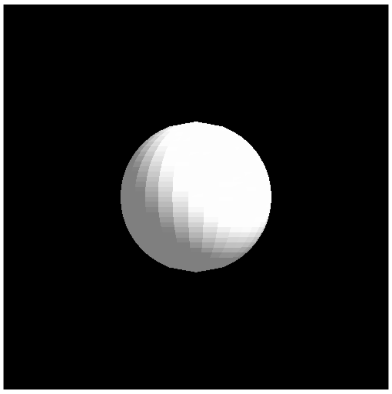
The following program is the smallest Obliq-3D program that actually creates a three-dimensional image. It opens up a graphics window on the screen, and displays a white sphere (see Figure 1a):

```
let r = RootGO_NewStd();
r.add(SphereGO_New([0,0,0],0.5));
```

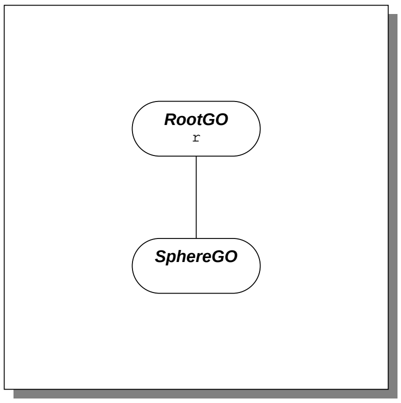
(Program 1)

The first line creates a root object, which is a special kind of graphical object, and assigns it to the variable `r`. Associated with each root object is a window on the screen, and creating the root object creates this window and installs it on the screen.

[1] If you read this document using the LECTERN Virtual Paper viewer, you can cut and paste the examples directly from the document into a window running an Obliq-3D interpreter.



(a) Window displaying the Scene



(b) Corresponding Scene Graph

Figure 1: Scene and Scene Graph created by Program 1

The expression `SphereGO_New([0,0,0],0.5)` creates a sphere object, which is another kind of graphical object. This particular sphere object represents a sphere whose center lies at the origin of the coordinate system, and whose radius is 0.5 units.

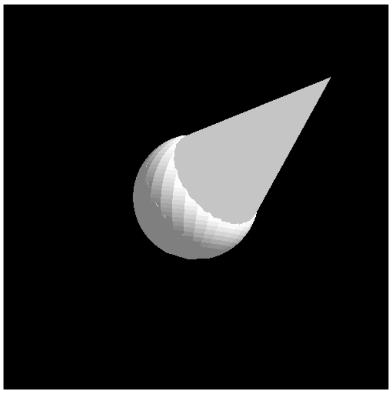
Finally, `r.add(···)` adds the sphere object to the root object, and thereby causes it to be displayed in the window associated with the root object. The user can manipulate the scene interactively, that is, he can move and rotate the displayed objects through mouse controls. This interactive behavior is not innate to root objects, but rather added to the root object by the function `RootGO_NewStd()`. This function also creates (in addition to the root itself) a camera through which the scene is viewed, and several light sources.

The `add` method, which makes an object part of a larger object, is understood by all objects of type `GroupGO`. In particular, it is understood by objects of type `RootGO`, which is the type of root objects and a subtype of `GroupGO`. This ability to group objects together into larger objects allows us to build up hierarchies of graphical objects. In general, these hierarchies must form a directed acyclic graph, called the *scene graph*. Figure 1b shows the scene graph corresponding to Program 1. For the sake of simplicity, the camera and the light sources that were created by the call to `RootGO_NewStd()` are omitted.

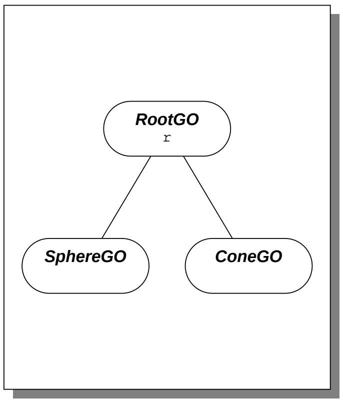
We can display a cone together with the sphere by adding the cone object to the root object. Here is a program that does this:

```
let r = RootGO_NewStd();
r.add(SphereGO_New([0,0,0],0.5));
r.add(ConeGO_New([0,0,0],[1,1,1],0.5));
```

(Program 2)



(a) Window displaying the Scene



(b) Corresponding Scene Graph

Figure 2: Scene and Scene Graph created by Program 2

The expression `ConeGO_New([0,0,0],[1,1,1],0.5)` creates an object that represents a cone whose base is centered at the origin and is of radius 0.5, and whose tip lies at point $(1,1,1)$. Figure 2 shows the window associated with `r`, and the scene graph.

All the scene graphs we have seen so far have been trees. Associated with the root of the tree was a window for viewing the scene. But it is possible to have several windows viewing the same scene. The following program creates two windows that display a scene consisting of a sphere and a cone:

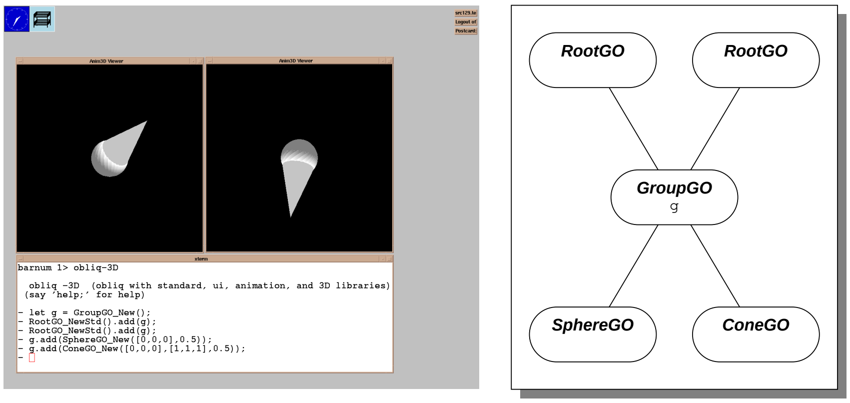
```
let g = GroupGO_New();
RootGO_NewStd().add(g);
RootGO_NewStd().add(g);
g.add(SphereGO_New([0,0,0],0.5));
g.add(ConeGO_New([0,0,0],[1,1,1],0.5));
```

(Program 3)

The first line creates a group object and assigns it to the variable `g`. The next two lines create two root objects, and add `g` to each. Creating the two roots also creates two windows on the screen. Finally, the last two lines create the sphere and the cone, just as in the previous example. This time, however, the sphere and the cone are not added directly to the roots, but to the group `g` instead. Figure 3 shows the two windows and the scene graph.

2.3 Properties

The graphical objects in a scene graph describe what types of objects are contained in the scene. But there are other attributes to an object besides its shape: its color, location, size, etc. These



(a) The two Windows created by Program 3 (b) Corresponding Scene Graph

Figure 3: The two Windows and the Scene Graph created by Program 3

aspects of an object are described by *properties*.

Let's assume we want to create a scene that contains a red sphere. We can create such a scene by modifying Program 1 as follows:

```
let r = RootGO_NewStd();
r.add(SphereGO_New([0,0,0],0.5));
SurfaceGO_SetColor(r,"red");
```

(Program 4)

The first two lines create a root object which contains a sphere object. The last line "attaches" the color property "red" to the root object and all objects contained in it. Figure 4a shows the resulting scene.

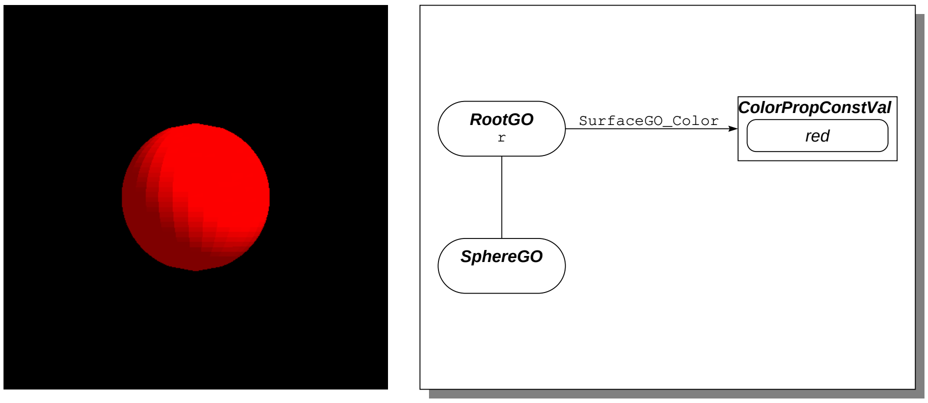
The `SurfaceGO` prefix indicates that the color property is associated with surfaces, that is, objects that are composed of polygons. (Sphere objects are actually approximated by polyhedra.)

A property consists of two parts: a *name* and a *value*. The name describes what aspect of a graphical object is affected by the property, and the value describes what this aspect looks like. In Program 4, the name of the property is `SurfaceGO_Color`, and the value is the color "red".

Many properties can be attached to a graphical object. These properties form a mapping — a partial function from names to values.

In the scene graphs shown throughout this report, we represent property values through boxes. For each property that is attached to a graphical object, we draw an arrow from the oval representing the graphical object to the box representing the property value. The arrow is labeled with the property name. Figure 4b shows the scene graph created by the program above.

Properties affect not only the graphical object they are attached to, but also all those objects that are contained in this object and that do not "override" this property. Consider the following



(a) Window displaying the Scene (b) Corresponding Scene Graph

Figure 4: Scene and Scene Graph created by Program 4

modification of Program 2:

```
let r = RootGO_NewStd();
let s = SphereGO_New([0,0,0],0.5);
r.add(s);
let c = ConeGO_New([0,0,0],[1,1,1],0.5);
r.add(c);
SurfaceGO_SetColor(r,"red");
```

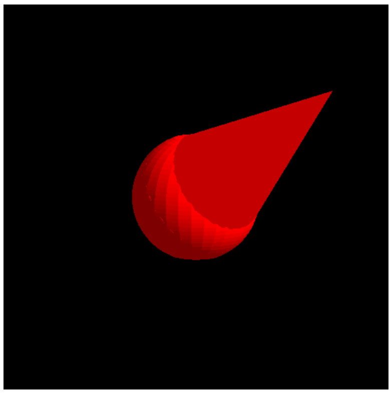
(Program 5)

The first five lines create a root object that contains a sphere and a cone. (This time, we created names for the sphere and the cone.) The last line attaches the color property "red" to the root object. This turns the root (and with it, the sphere and the cone) red. Figure 5a shows the result. Adding the statement

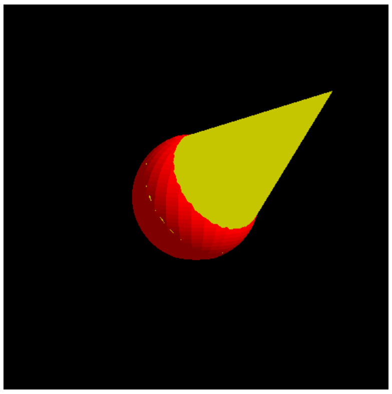
```
SurfaceGO_SetColor(c,"yellow");
```

(Program 5— continued)

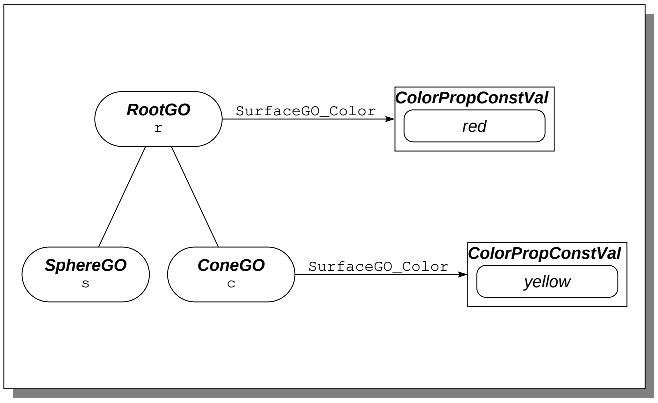
to Program 5 attaches the color property yellow to the cone. This property "overrides" the color of the root: The cone turns yellow, while the rest of the root (i.e. the sphere) remains red (see Figure 5b).



(a) Initial Scene



(b) After setting the Color of the Cone



(c) Final Scene Graph

Figure 5: Scenes and Scene Graph created by Program 5

Note that the sequence of the statements is immaterial[2] (except that a variable cannot be referenced before it is declared). The following program creates exactly the same scene:

```
let r = RootGO_NewStd();
let s = SphereGO_New([0,0,0],0.5);
let c = ConeGO_New([0,0,0],[1,1,1],0.5);
SurfaceGO_SetColor(c,"yellow");
SurfaceGO_SetColor(r,"red");
r.add(c);
r.add(s);
```

(Program 6)

Color properties are one kind of property. Other kinds include boolean properties, real properties, point properties, and so on.

Recall that properties control not only surface attributes of an object, such as its color and transparency, but also spatial attributes, such as its location and size. For example, the center of a sphere is controlled by a point property named `SphereGO_Center`, and the radius is controlled by a real property named `SphereGO_Radius`. The expression `SphereGO_New(c,r)` creates a sphere object, and attaches a property with name `SphereGO_Center` and value `c` and a property with name `SphereGO_Radius` and value `r` to the new sphere object.

It is possible to detach these properties from the sphere object, using the `unsetProp` method. Consider the following program:

```
let r = RootGO_NewStd();
let s = SphereGO_New([0,0,0],0.5);
r.add(s);
s.unsetProp(SphereGO_Center);
s.unsetProp(SphereGO_Radius);
SphereGO_SetCenter(r,[1,0,0]);
SphereGO_SetRadius(r,0.3);
```

(Program 7)

The first three lines create a root object `r` and a sphere object `s`, and add `s` to `r`. The center of the sphere is at point $(0,0,0)$, and its radius is 0.5. So far, the program is equivalent to Program 1.

The next two lines detach the center and the radius property from the sphere object. As these properties are not defined for the root object either, the sphere is now displayed at a default location and with a default radius.

The last two lines attach a center property and a radius property to the root object. The sphere is displayed centered at point $(1,0,0)$, and with a radius of 0.3.

[2] This is one of the key differences between Anim3D and Inventor [24, 25, 26]. Inventor uses a very similar model: scenes are described by a scene graph composed of nodes. However, it does not distinguish between graphical objects and properties; both are nodes in the scene graph. A property node in a scene graph affects all its "right" siblings, until it is "overridden" by another property node. So, the order of nodes in the scene graph becomes important.

So, the center property and the color property behave exactly the same: A property attached to an object (be it a group or a sphere) affects that object and all objects contained in it, except those that have a property with the same name attached.

It should also be noted that any property can be attached to any object, but that the way an object is displayed does not necessarily depend on every property. For example, it is perfectly legal to attach a `SphereGO_Center` property to a cone, but this will not affect the appearance of the cone.

2.4 The On-Line Help Facility

Obliq has a built-in facility that provides on-line help to the user. Typing

```
help;
```

shows a list of all preloaded libraries. You can also obtain more information on a particular library. For example, typing "`help SphereGO;`" reveals the types and functions exported by the `SphereGO` module:

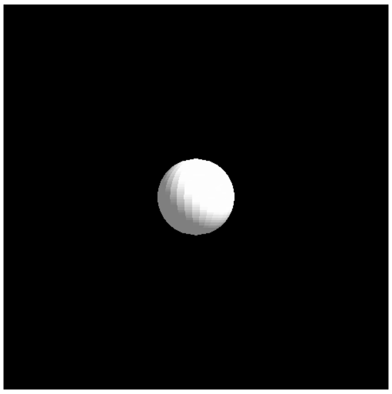
```
- help SphereGO;
  SphereGO_New(p: PointVal, rad: RealVal): SphereGO
  SphereGO_NewWithPrec(p: PointVal, rad: RealVal, prec: Int): SphereGO
  SphereGO_Center: PointPropName
  SphereGO_Radius: RealPropName
  SphereGO_SetCenter(go: GO, center: PointVal): Ok
  SphereGO_SetRadius(go: GO, radius: RealVal): Ok
WHERE
  SphereGO <: SurfaceGO
  PointVal = PointPropVal + Point3
  RealVal = RealPropVal + Real + Int
```

The type notation follows the one described in [3]; it is reviewed at the beginning of the reference manual.

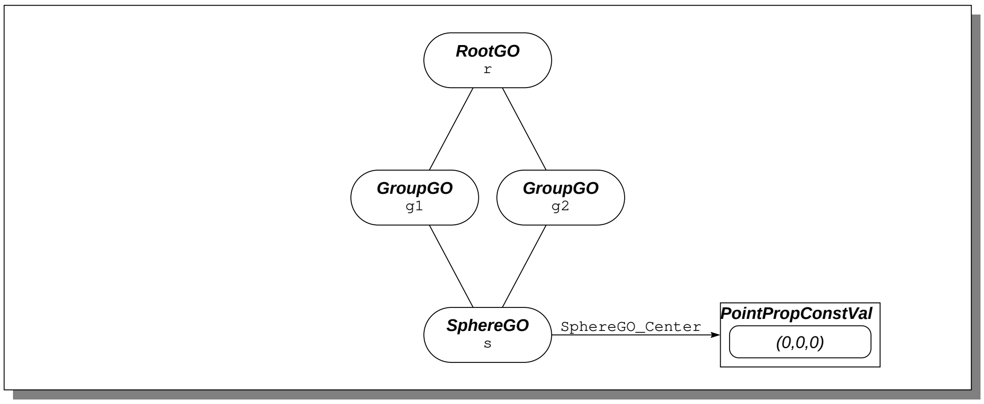
2.5 Non-Tree Scene Graphs

We mentioned before that scene graphs are directed acyclic graphs, and not simply trees. Program 3 created a scene graph with two roots (where "root" refers to a node with in-degree 0), which allowed us to view the same scene from two different vantage points.

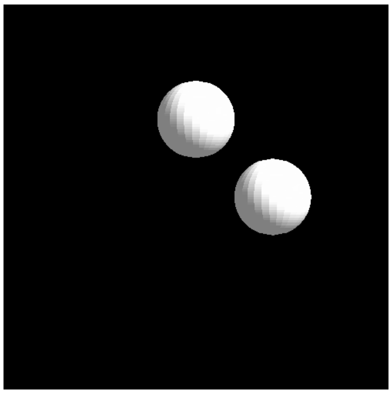
There are also meaningful scene graphs that contain more than one path between a pair of nodes. For example, consider the following program:



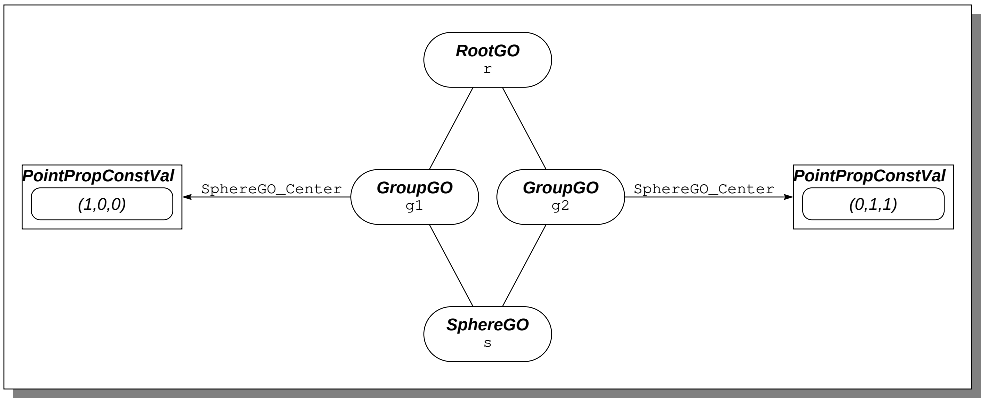
(a) Initial Scene



(b) Corresponding Scene Graph



(c) Final Scene



(d) Corresponding Scene Graph

Figure 6: Scenes and Scene Graph created by Program 8

```
let r = RootGO_NewStd();
let g1 = GroupGO_New();
r.add(g1);
let g2 = GroupGO_New();
r.add(g2);
let s = SphereGO_New([0,0,0],0.5);
g1.add(s);
g2.add(s);
```

(Program 8)

This program creates a root object that contains two group objects that in turn both contain the same sphere object. The program so far displays a scene that apparently contains a single sphere (see Figure 6a). However, adding the following three lines reveals that there are really two spheres in the scene, which just happen to be superimposed:

```
s.unsetProp(SphereGO_Center);
SphereGO_SetCenter(g1,[1,0,0]);
SphereGO_SetCenter(g2,[0,1,1]);
```

(Program 8— continued)

These statements detach the center property from the sphere object, and attach two differing center properties to the groups `g1` and `g2`. This causes the sphere to be shown at the location described by the center property attached to `g1`, and also at the location described by the center property attached to `g2` (see Figure 6c).

We can describe the semantics of scene graphs with multiple paths between any pair of nodes in a procedural fashion: For each path between a root node r and a leaf node l, l is rendered once, using the current value of the properties attached to the nodes along this path.

We can also describe it in a more declarative way: Every scene graph can be "unfolded" into a forest of scene trees. Table 1 gives an algorithm for performing the unfolding, and Figure 7 shows an example of an unfolding.

2.6 More on Property Values

Let's take a closer look at property values. A property value represents a time-variant value. It is implemented as an Obliq object that has a method `value` which takes a time and returns the value at that time.

So far, we have seen only constant property values, that is, property values that do not change over time. The expression `RealProp_NewConst(0.5)` creates a new real property value, whose value is 0.5 regardless of the current time.

The same property value may be attached to several graphical objects. Consider the following program:

INPUT: Scene graph G. G is a directed acyclic graph.
OUTPUT: Unfolded scene graph G'. G' is a forest of trees.

1. Let G' be the empty graph.
2. Add an artificial source vertex s to G. Add artificial arcs from s to every vertex in G with in-degree 0.
3. For every vertex u in G and each path p from s to u, add a vertex u_p to G'. We say that u_p corresponds to u.
4. For each arc (u, v) in G, and for all vertices u_p, $v_{p'}$ in G' such that u_p corresponds to u, $v_{p'}$ corresponds to v, and $p' = pv$, add an arc $(u_p, v_{p'})$ to G'. We say that $(u_p, v_{p'})$ corresponds to (u, v).
5. Remove the artificial source vertex and arcs from G, and the corresponding vertex and arcs from G'.

Table 1: Unfolding of Scene Graphs

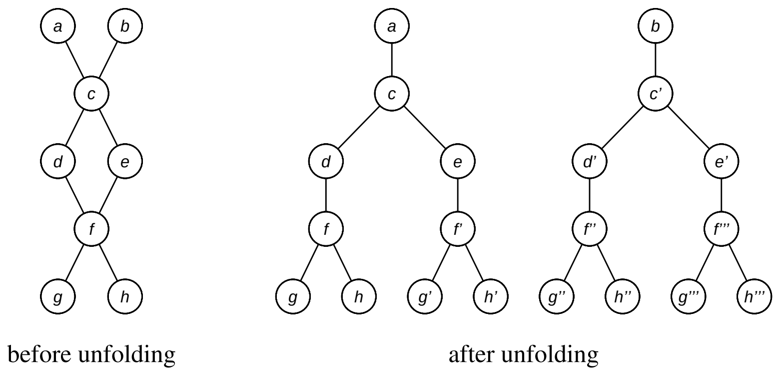
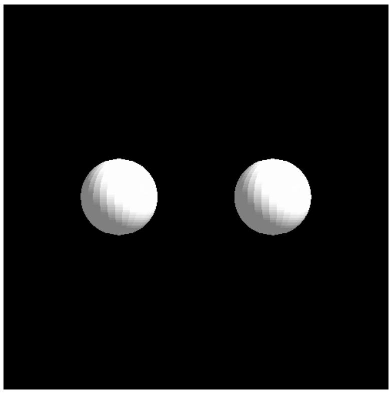
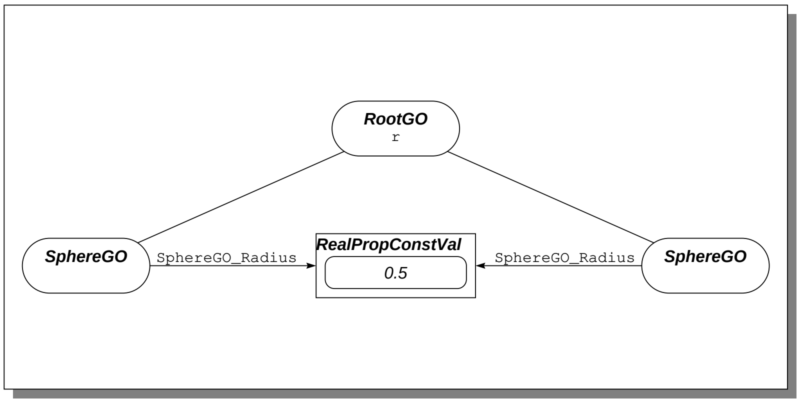


Figure 7: Example Unfolding of a Scene Graph



(a) Window displaying the scene



(b) Corresponding Scene Graph

Figure 8: Scene and Scene Graph created by Program 9

```
let r = RootGO_NewStd();
let rad = RealProp_NewConst(0.5);
r.add(SphereGO_New([-1,0,0],rad));
r.add(SphereGO_New([1,0,0],rad));
```

(Program 9)

Figure 8 shows the scene displayed by the program, together with the corresponding scene graph. The scene contains two spheres; the radius of each sphere is controlled by the real property value `rad`. Changing `rad` affects both spheres.

The ability to attach the same property value to several graphical objects provides us with an alternative to "property inheritance". In the example above, instead of attaching `rad` to each sphere, we could have attached it to their common ancestor `r` instead. If no other radius property were attached to either sphere, then the radius of both spheres would have been "inherited" from `r`. However, there are many cases in which the "natural" structure of the scene graph prevents us from using property inheritance. For example, in a scene describing a car, we might want to put two tires and an axle into the same group, but we also want the four tires to have the same color, and the two axles to have a different color. Having the tires and the axle in the same group prevents us from using property inheritance to describe their color; in such a case, shared property values are the best solution.

2.7 Overloading

You may have noticed that in Program 1, we invoked the function `SphereGO_New` with a real number as the second argument, whereas in Program 9, we used a real property value for the second argument. `SphereGO_New` is what is called an *overloaded* function: a function that

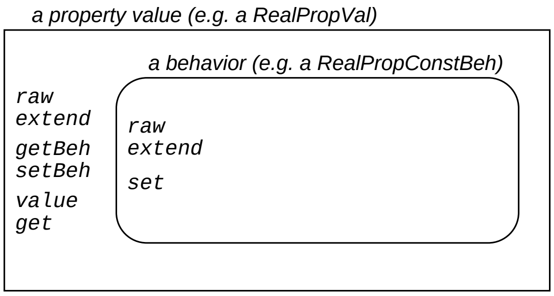


Figure 9: Properties and Behaviors

accepts arguments of different types. `SphereGO_New` accepts either points or point property values as the first argument, and either integers, reals, or real property values as the second argument. A point, in turn, is an array of three elements, which can be either integers or reals.

It is important to remember that only the 3D functions of Obliq-3D are overloaded. The rest of Obliq distinguishes between, say, integers and reals, and these types cannot be freely interchanged. For example, the expression `1 + 0.1` will cause a run-time error.

2.8 Behaviors

There are many different ways in which a property value might change over time. It might remain constant, it might change perpetually, it might start to change from one value to another when signaled, and so on.

When a property value is attached to a single graphical object, we can change the way this property value changes over time simply by replacing it with another one. But this approach is impractical when the property value is attached to many graphical objects. So, instead of replacing the entire property value by a new one (and having to update every graphical object that uses this property value), we simply change the *behavior* of the property value.

Behaviors, just like property values, are represented by Obliq objects. In a sense, property values serve merely as "clerks" for behaviors: they forward value-inquiring messages from the client program or the animation server to the behavior. Replacing one behavior with another one is transparent to the client program and the animation server because the property value acts as an intermediary. Figure 9 illustrates this encapsulation.

Constant property values are really property values that contain *constant behaviors*. A property value might change from being constant to being time-variant during its lifetime, while a constant behavior will remain constant for its entire existence.

The expression `RealProp_NewConstBeh(0.5)` creates a new real-valued constant behavior object `b`, which encapsulates the value 0.5, and returns it. `b` understands the message `set`.

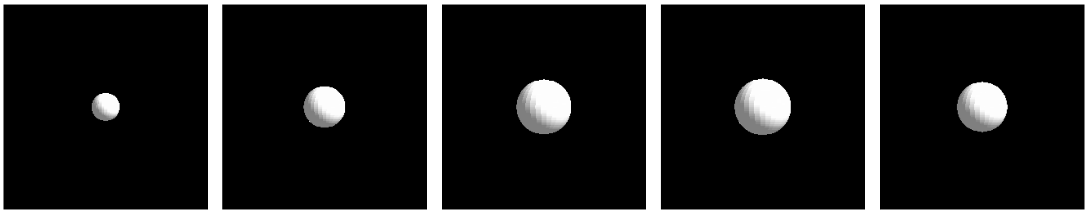


Figure 10: Snapshots of the animation created by Program 10

The call `b.set(0.7)` will change `b`'s value from 0.5 to 0.7.

We can access the behavior of a property value by sending the message `getBeh()` to the property value. Recall Program 9, which displays two spheres whose radius is controlled by the constant property value `rad`. The following statement changes the radius of the two spheres to 0.3:

```
rad.getBeh().set(0.3);
```

We can replace the behavior of a property value by sending it the `setBeh` message. For example,

```
rad.setBeh(RealProp_NewConstBeh(0.7));
```

will replace the existing behavior of `rad` with a new behavior that always evaluates to 0.7.

2.9 Asynchronous Behaviors

As we said before, both property values and behaviors represent time-variant values. A behavior implements a function from a moment in time to the actual value at that time[3]; the property value serves as an intermediary between graphical objects and behaviors, allowing us to exchange behaviors without having to update graphical objects.

In the previous section, we encountered the most primitive form of behaviors, namely constant behaviors, that is, behaviors whose value does not change (unless explicitly changed through a `set` message).

Asynchronous behaviors are the second class of behaviors in our system. Asynchronous behaviors change over time; and they change throughout their entire lifetime. They do not depend on other property values, and they do not synchronize their changes with them.

The following program displays a sphere, whose radius oscillates between 0.3 and 0.7:

```
let root = RootGO_NewStd();
let rad = RealProp_NewAsync(meth(self,time) 0.5 + math_sin(time) * 0.2 end);
root.add(SphereGO_New([0,0,0],rad));
```

(Program 10)

[3] This is a slight simplification, as behaviors, unlike mathematical functions, can have local state.

The second line creates a new real property value `rad` with an asynchronous behavior. This behavior is defined by a method that takes two arguments, the behavior itself and the current time, and returns a real. In this case, the value of `rad` at time t is defined to be $0.2\sin(t) + 0.5$.[4]

`rad` is attached as the radius property to a sphere object, which causes the sphere to pulse. Figure 10 shows a "film-strip" of successive snapshots of the scene generated by this program.

2.10 Synchronous Behaviors

A *synchronous behavior* represents a value that does not change continuously, but rather upon being signaled. Each synchronous behavior is connected to a synchronization object, called an *animation handle*. We say that the animation handle *controls* the behavior. Many synchronous behaviors can be controlled by the same handle.

Conceptually, a synchronous behavior consists of a *current value* and a *request queue*. Requests are objects that change the current value of the behavior. For each type of behavior, there is a set of predefined requests that perform linear interpolations between the current value of the behavior and a new one; in addition, the client program can define arbitrary new requests. Each synchronous behavior has methods for adding predefined and user-defined requests to its request queue; these requests will be processed once the animation handle controlling the behavior is signaled.

Associated with each request is a *start time* and a *duration*. A request with start time s and duration d starts to affect the value of the behavior s seconds after the controlling animation handle is signaled, and ceases to do so d seconds later.

The following program demonstrates the use of synchronous behaviors:

```
let ah = AnimHandle_New();
let p = PointProp_NewSync(ah,[0,0,0]);
let c = ColorProp_NewSync(ah,"red");
let r = RootGO_NewStd();
let s = SphereGO_New(p,0.5);
r.add(s);
SurfaceGO_SetColor(s,c);
p.getBeh().linMoveTo([0,1,0],1,3);
c.getBeh().rgbLinChangeTo("green",0,4);
ah.animate();
```

(Program 11)

The first line creates a new animation handle `ah`. The next two lines create a point property value `p` whose initial value is $(0, 0, 0)$, and a color property value `c` whose initial value is "red". Both `p` and `c` are synchronous property values (that is, property values containing a synchronous behavior), and both are controlled by the animation handle `ah`.

[4] It should be noted that Obliq differs from many other languages with infix operators in that all operators have the same precedence, and associate to the right. So, `4*7+3` evaluates to 40, not 31.

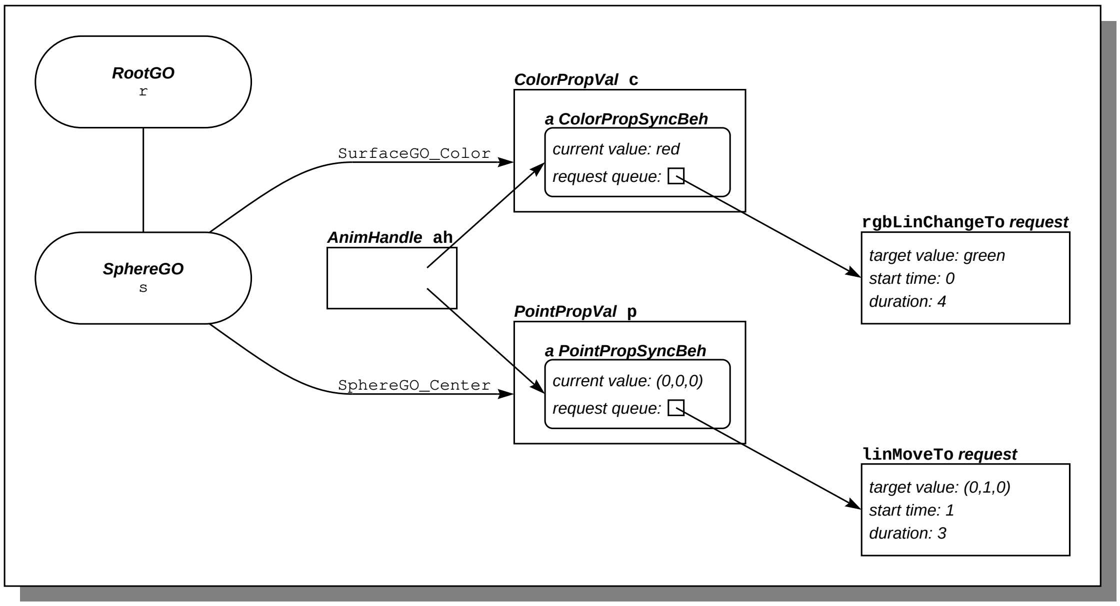


Figure 11: Scene Graph created by Program 11

The next three lines create a root object, which contains a sphere object. The center of the sphere depends on the current value of `p`, and its color depends on the current value of `c`. So, initially the sphere is red, and centered at the origin.

In the next line, we send the message `linMoveTo` to the behavior of `p`, asking it to change from its present value to $(0, 1, 0)$. This change will start 1 second after the animation handle is signaled, and will take 3 seconds to complete. Similarly, we send a message `rgbLinChangeTo` to the behavior of `c`, asking it to change from its present value (that is, red) to green. The change will start right when the animation handle is signaled, and will take 4 seconds to complete. Figure 11 shows the state of the scene graph after this message.

Up to this point, no changes have actually taken place. Sending the message `animate` to the animation handle `ah` causes the requests to be processed. The `ah.animate()` call returns only after all requests have been processed, that is, it will run for 4 seconds.

Many requests can be added to the request queue of a synchronous behavior. However, the time intervals of these requests must not overlap. So,

```
let c = ColorProp_NewSync(ah, "red");
c.getBeh().rgbLinChangeTo("green",0,2);
c.getBeh().rgbLinChangeTo("blue",2,2);
```

is legal, as the intervals $(0, 0 + 2)$ and $(2, 2 + 2)$ do not overlap. On the other hand,

```
let c = ColorProp_NewSync(ah, "red");
c.getBeh().rgbLinChangeTo("green",0,2);
c.getBeh().rgbLinChangeTo("blue",1,2);
```

is illegal, and will cause an Obliq exception to be raised. It should also be noted that zero-duration intervals are closed, whereas others are open. So

```
let c = ColorProp_NewSync(ah, "red");
c.getBeh().rgbLinChangeTo("green",0,0);
c.getBeh().rgbLinChangeTo("blue",0,2);
```

is legal, as the intervals $[0, 0]$ and $(0, 2)$ do not overlap, whereas

```
let c = ColorProp_NewSync(ah, "red");
c.getBeh().rgbLinChangeTo("green",0,0);
c.getBeh().rgbLinChangeTo("blue",0,0);
```

is illegal, because $[0, 0]$ and $[0, 0]$ overlap.

2.11 Dependent Behaviors

A *dependent behavior* is a behavior whose value depends on other property values. Dependent behaviors allow us to specify functional dependencies between property values. Such dependencies are often called one-way constraints. It is a checked run-time error if the dependency graph induced by the dependent behaviors includes any cycles.

A dependent behavior is defined just like an asynchronous behavior, through a method that is passed to its creation function. This method takes the behavior itself and the current time as arguments, and returns the value for the behavior at that time.

The following program creates a sphere and a torus. The radius of the sphere is defined by an asynchronous property value, which causes the sphere to pulsate. The radius of the torus is dependent on the radius of the sphere; it will always be 1.5 times as large.

```
let root = RootGO_NewStd();
let rad1 = RealProp_NewAsync(meth(self,time) 0.5 + math_sin(time) * 0.2 end);
root.add (SphereGO_New ([0,0,0],rad1));
let rad2 = RealProp_NewDep (meth(self,time) rad1.value(time) * 1.5 end);
root.add (TorusGO_New([0,0,0], [0,1,0], rad2, 0.1));
```

(Program 12)

The first three lines of this program are identical to Program 10. The fourth line defines a new dependent property value `rad2`, whose value at a given time is 1.5 times the value of `rad1` at that time. The last line creates a torus, whose center is $(0, 0, 0)$ and whose normal vector is $(0, 1, 0)$. The major radius of the torus is defined to be `rad1`, and the minor radius has a constant value of 0.1.

Figure 12 shows a “film-strip” of successive scenes created by this program.

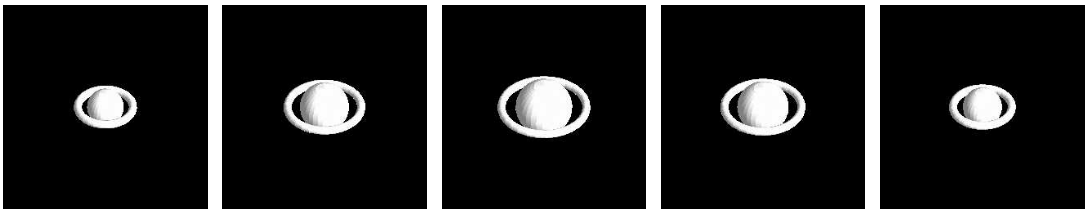


Figure 12: Snapshots of the animation created by Program 12

2.12 Locking

You may have noticed that none of the programs we have shown contained an explicit statement to draw the scene. Obliq-3D is based on a *damage-repair model*: modifying the scene graph damages the scene, that is, the scene shown on the screen is no longer consistent with the scene graph. An animation server thread detects any damage, and repairs it by redrawing the scene.

This model is intriguing because of its simplicity, and because it minimizes the amount of Obliq-3D code a user has to write. However, there are cases where we would like a set of changes to the scene graph to be atomic; that is, we do not want the animation server to redraw the scene before we have performed all the changes.

We can declare a set of operations to be atomic by surrounding them with a locking construct. There is a global lock `Anim3D_lock`, which the animation server must hold in order to perform any redrawing. So, acquiring this lock prevents the animation server from redrawing the scene.

The following program demonstrates the use of this locking facility. The aim of this program is to visualize a simple data structure, namely a binary search tree. We want to show each key as a color-coded sphere, where colors range from yellow over green, cyan, blue, and magenta to red. Yellow corresponds to a small key, whereas red corresponds to a large key.

There are two ways in which this could be done: we could write a visualization function that incrementally updates the scene graph whenever a new element is inserted into the tree, or we could write a function that, given a tree, computes the entire scene graph from scratch. The latter method implies that we have to discard the old scene graph and replace it with a new one. Such an action should certainly be atomic, otherwise the display might occasionally be empty.

```
let rec Insert =
  proc (tr, el)
    if tr is ok then
      {key => el, left => ok, right => ok}
    elsif tr.key > el then
      {key => tr.key, left => Insert (tr.left, el), right => tr.right}
    else
      {key => tr.key, left => tr.left, right => Insert (tr.right, el)}
    end
  end;

let maxkey = 10;

let KeyColor =
  proc (key) color_hsv(real_float(key)/real_float(maxkey), 1.0, 1.0) end;

let leftXf  = Matrix4_Translate(Matrix4_Scale(Matrix4_Id,0.5,0.5,0.5),-1,-2,0);
let rightXf = Matrix4_Translate(Matrix4_Scale(Matrix4_Id,0.5,0.5,0.5), 1,-2,0);

let rec VisTree =
  proc (tr)
    let g = GroupGO_New();
    if tr isnot ok then
      let node = SphereGO_New ([0,0,0],0.4);
      SurfaceGO_SetColor(node, KeyColor (tr.key));
      g.add(node);
      if tr.left isnot ok then
        g.add(LineGO_New([0,0,0],[-1,-2,0]));
        let left = VisTree(tr.left);
        GO_SetTransform(left,leftXf);
        g.add(left);
      end;
      if tr.right isnot ok then
        g.add(LineGO_New([0,0,0],[1,-2,0]));
        let right = VisTree(tr.right);
        GO_SetTransform(right,rightXf);
        g.add(right);
      end;
    end;
    g;
  end;
```

(Program 13)

```
let Run =
  proc (keylist)
    var tree = ok;
    let root = RootGO_NewStd();
    let treeRoot = GroupGO_New();
    root.add (treeRoot);
    foreach key in keylist do
      tree := Insert (tree, key);
      lock Anim3D_lock do
        treeRoot.flush();
        treeRoot.add(VisTree(tree));
      end;
      thread_pause(2.0);
    end;
  end;

Run ([5,3,7,2,10,8,4,6,1,9]);
```

(Program 13— continued)

Trees are represented as objects with three fields: `key` for the key, `left` for the left subtree, and `right` for the right subtree. The empty tree is represented by the value `ok` (Obliq's equivalent of `NIL`).

`Insert` is a function that takes a binary search tree `tr` and an element `el` to be inserted, and returns the tree that results from inserting `el` into `tr` as a leaf node. `Insert` has no side effects.

`Keycolor` is a function that takes a key (i.e. a number) and returns a color corresponding to this number. `KeyColor` relies on the constant `maxkey`, which indicates the largest key that should be expected.

`VisTree` is a function that takes a tree and returns a graphical object representing the tree. `VisTree` works as follows: First, it creates a new group object `g`. If the tree is empty, it simply returns the empty group, otherwise, it creates a sphere centered at the origin, attaches a color corresponding to the key of the root to the sphere, and adds the sphere to `g`. If the left subtree is not empty, it also adds a line from the sphere to the position where the root of the left subtree is. It then calls itself recursively, obtaining a graphical object `left` containing the left subtree. The sphere representing the root of this left subtree again lies at the origin. `VisTree` attaches a transformation matrix `leftXf` to `left`, which shifts the entire subtree down and to the left, and also scales it down to half its size. Then, `VisTree` adds `left` to `g`. The right subtree is visualized analogously. Finally, `VisTree` returns the group `g`, which now contains graphical objects visualizing the entire tree. It should be noted that `VisTree` is entirely functional; it has no side-effects whatsoever.

`Run` is the main procedure of the program. It takes an array of keys as an argument. It starts out by creating an empty tree `tree`, a root object `root`, and a group object `treeRoot` to contain the visualization of `tree`. `treeRoot` is added as a child to `root`. The main loop of `Run` iterates over the array of keys. Each key is inserted into the tree, and then the scene graph

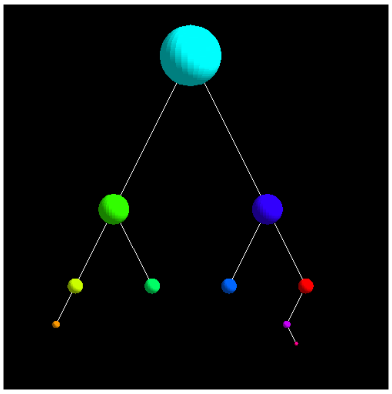


Figure 13: Final state of the animation created by Program 13

is updated. The update consists of two steps: first, we flush `treeRoot`, discarding the previous visualization of the tree. Second, we call `VisTree` to obtain a graphical object showing the new tree, and add this object to `treeRoot`. To prevent flicker, these two steps must be atomic, so they are protected by a locking construct that acquires `Anim3D_lock`. The main loop ends with a call to `thread_pause`, which causes the program to pause for two seconds during each iteration, making the visualization easier to follow.

The last line invokes our program on a small test case. Figure 13 shows the scene visualizing the final state of the tree.

2.13 Callbacks

So far, we have encountered two major concepts: *graphical objects*, which provide various geometric primitives and which allow us to arrange the primitives into hierarchies; and *properties*, which determine the appearance of graphical objects (apart from their intrinsic shape), and which are the basic vehicle for animation. The third major concept in Obliq-3D are *callbacks*: objects which are used to specify the way a graphical object reacts to input events.

We distinguish between three different kinds of events: *mouse events*, which are triggered by mouse button transitions; *position events*, which are triggered by changes in the mouse position; and *key events*, which are triggered by keyboard key transitions. For each kind of event, there is a corresponding type of *callback object*.

Let's take a detailed look at how mouse events are processed. When the user presses or releases a mouse button while a rendering window is selected, a *mouse event* is generated. This event causes the message `invokeMouseCB(mr)` to be sent to the root object associated with that window. `mr` is a *mouse event record*, an Obliq object that contains information about the mouse event (such as which button was pressed or released, whether the transition was a button press or a release, what the position of the mouse was, and what other buttons or option keys were pressed at this time).

Associated with every graphical object is a *mouse callback stack*, that is, a stack for mouse callback objects. When a graphical object receives an `invokeMouseCB(mr)` message, it checks if there are any callback objects on its mouse callback stack. If there is one, it sends the message `invoke(mr)` to the topmost callback object; if not, the mouse event is ignored.

Each mouse callback object has a corresponding `invoke` method that takes two arguments: the callback object itself and a mouse event record. The `invoke` method is responsible for deciding what action should be taken in order to respond to the callback. Possibilities are to simply ignore it, to forward it to other graphical objects (this makes sense particularly for group objects), or to change some property values (for instance transformation properties, to rotate the scene).

Mouse callback objects are created through the `MouseCB_New` function, which takes a method as its argument and assigns it to the `invoke` field of the new mouse callback object.

The client program can push new mouse callback objects onto the mouse callback stack of a graphical object by sending the `pushMouseCB` method to that object, and it can remove the top or an arbitrary callback object from the stack by sending the messages `popMouseCB` or `removeMouseCB` to the graphical object. Pushing a callback object means establishing a new reactive behavior, and popping the stack means reverting back to a previous behavior.

The following example demonstrates the use of a mouse callback. The program initially shows a sphere. Whenever the user presses the left mouse button, the sphere is replaced by a box, and when he releases the button, the sphere reappears.

```
let root = RootGO_NewStd();
let ball = SphereGO_New ([0,0,0], 1);
let box = BoxGO_New ([-1,-1,-1],[1,1,1]);
root.add(ball);
let M = meth(self,mr)
          if (mr.change is "Left") then
            if (mr.clickType is "FirstDown") or
               (mr.clickType is "OtherDown") then
              root.remove(ball);
              root.add(box);
            else
              root.remove(box);
              root.add(ball);
            end;
          end;
        end;
root.pushMouseCB(MouseCB_New(M));
```

(Program 14)

Position events and key events are handled similarly.

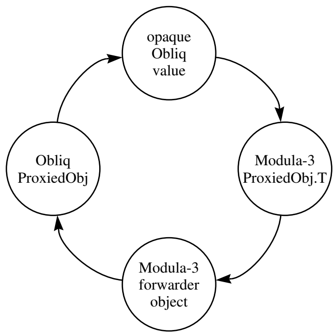


Figure 14: Obliq and Modula-3 counterparts of a Proxied Object

2.14 Extending Objects

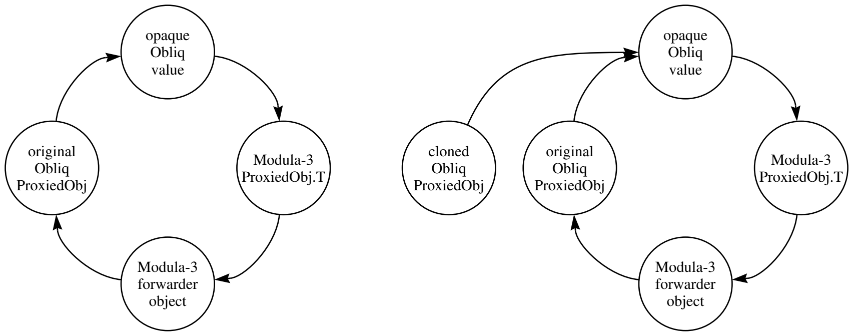
Obliq has a `clone` construct, which takes an arbitrary number of objects and returns a new object which is the "concatenation" of the arguments. More precisely, `clone` requires the sets of field names of the argument objects to be pairwise disjoint, and it returns a new object whose set of field names is the union of the sets of field names of the arguments, and whose field contents are identical to the contents of the corresponding fields of the argument objects.

The clone construct fulfills several purposes: it provides a mechanism for performing a shallow copy of an object, and it allows us to extend objects (that is, add additional fields).

There is a restriction in Obliq-3D that certain objects must not be cloned. To understand the reason for this, we have to understand the connection between Obliq-3D and the underlying Modula-3 animation library Anim3D.

Obliq-3D provides a variety of different object types: graphical objects (objects of type `GO`), property names (`PropName`), property values (`PropVal`), property behaviors (`PropBeh`), and so on. All of these are subtypes of `ProxiedObj`, the type of *proxied objects*. In our terminology, a proxied object is an Obliq object with a Modula-3 counterpart. Conversely, on the Anim3D side, there is a type `ProxiedObj.T` with subtypes `GO.T`, `Prop.Name`, `Prop.Val`, `Prop.Beh`, and so on. The two objects are connected to each other: the Obliq `ProxiedObj` will forward certain messages to the Modula-3 `ProxiedObj.T`, and the Modula-3 object can in turn invoke methods of the Obliq object. Figure 14 illustrates the relationship.

Each `ProxiedObj` has (at least) two fields: `raw` and `extend`. The `raw` field contains an opaque Obliq value, which in turn contains a reference to the corresponding Modula-3 `ProxiedObj.T`. The Modula-3 object could in turn contain a pointer back to the Obliq `ProxiedObj`. However, one of our goals in designing Anim3D was to allow for an embedded language, but to make it independent of any particular one. Therefore, we put a forwarder



(a) Before calling `clone` (b) After calling `clone`

Figure 15: The effect of `clone` on Proxied Objects

object, whose signature is not Obliq-specific, between the Modula-3 `ProxiedObj.T` and the Obliq `ProxiedObj`.

The important point to remember is that an Obliq `ProxiedObj` and a Modula-3 `ProxiedObj.T` refer to each other, and that there is a one-to-one relationship between them. So what would happen if we clone an Obliq object? Assume we have a behavior object `beh` with a Modula-3 counterpart `x`. The relationship between the two objects is shown in Figure 15a. Executing the statement

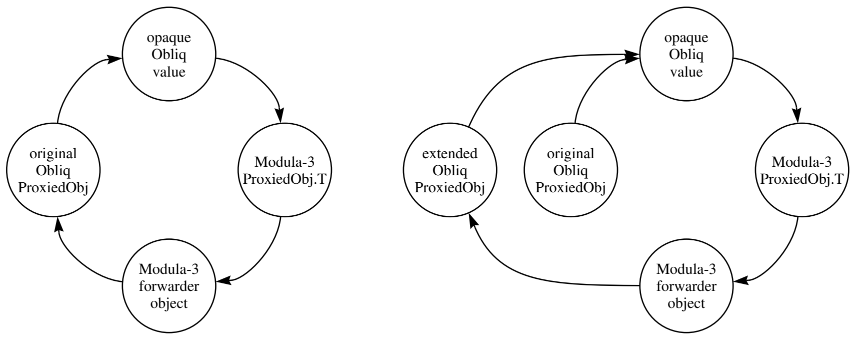
```
let beh2 = clone (beh, {newField => ...});
```

will create a new behavior `beh2`, which has an extra field `newField`. But the `raw` field of `beh2` still refers to the same opaque value and therefore to the same Modula-3 object `x`. `x` in turn still refers to `beh`, not to `beh2` (see Figure 15b). This might have fatal consequences. For instance, we would expect that

```
pv.setBeh(beh2);
```

should make `beh2` the behavior of the property value `pv`. However, `pv.getBeh()` will return `beh` instead of `beh2`, as the Modula-3 counterpart of `beh2` still points back to `beh`.

To overcome this problem, Obliq-3D offers an alternative way to extend `ProxiedObj` objects. Each `ProxiedObj p` has a method `extend`, which takes a second Obliq object `o` (which must not be a `ProxiedObj`), and returns a new object `p'`, whose set of field names is the union of the field names of `p` and `o`. In other words, the result of `p.extend(o)` is the same as the result of `clone(p,o)`. However, `extend` also modifies the forwarder to point to `p'` instead of `p`. So, after calling `p.extend(o)`, `p` should not be used any more. Figure 16 illustrates the



(a) Before calling `extend` (b) After calling `extend`

Figure 16: The effect of `extend` on Proxied Objects

effect of `extend`.

2.15 Naming Graphical Objects

There are cases where it would be convenient to find a particular graphical object within a scene graph, without having to traverse the graph.

In order to build such a facility for finding objects in a scene graph, we have to decide on how to identify the object we are searching for. We could use the object itself as a search key, but that would be somewhat useless: if we already have the object, why search for it?

An alternative is to attach a symbolic label, a *name*, to graphical objects, and then search for them by name. In Obliq-3D, a name `n` is simply a value of type `Text`. We can attach `n` to a graphical object `g` by calling `g.setName(n)`. We can also inquire the name of a given graphical object: `g.getName()` returns the name of `g` if it is defined, and `ok` otherwise. Finally, `g.findName(n)` traverses the scene graph to see if `g` or any of its descendants are named `n`; if so, it returns the object named `n`, otherwise, it returns `ok`. If there are several nodes with the same name in the scene graph, `findName` will return the one it finds first.

We have indicated before (see page 5) that the expression `RootGO_NewStd()`, which creates a new root object and returns it, also attaches a camera and several light sources to it. The naming facility give us the ability to access these objects. The name of the camera that comes with the standard root object is `default-camera`, the name of the first light source is `default-ambient-light`, and the name of the second is `default-vector-light`.

2.16 Requests

Section 2.10 explained how synchronous behaviors can be used to change a property value from its current value to a new one. At least for continuous property values (e.g. reals, points, colors, and transformations), this implies that some interpolation between the initial value and the new one must be performed, in order to make the transition appear smooth. Each synchronous behavior provides one or more default interpolation methods. These methods create a request object for performing the interpolation, and they add the request to the behavior's request queue.

Synchronous real behaviors have a method `linChangeTo`, which performs a linear, i.e. constant-speed, interpolation between the two values. Synchronous point behaviors have methods `linMoveTo` and `linMoveBy` which move at a constant speed and over a straight path through 3-space, either to a specified point, or by a specified amount. Synchronous color behaviors provide a method `rgbLinChangeTo` which interpolates between the two colors such that the intermediate values move at a constant speed over a straight path through RGB space[5]. Synchronous transformations provide a method `changeTo` that interpolates between two (slightly restricted) transformation matrices by using a rather complicated technique.

But what if you want to use a different interpolation method? For instance, you might want to change a color property value from its current value to a new one, but move on a straight path through HSV-space[6] instead of RGB-space. Obliq-3D enables you to do just that by allowing you to create arbitrary request objects, which encapsulate an interpolation method. All the interpolation methods that are provided by default are implemented in terms of request objects.

A color request object is created by calling the function `ColorProp_NewRequest`. This function takes the desired start time of the interpolation (relative to the time at which the controlling animation handle is signaled), its desired duration, and a method `m`, which will be used to interpolate between the initial color value and the desired new one. `m` takes three arguments: the request itself, a color that shall be the value to start interpolating from, and the time that has elapsed since the controlling animation handle was signaled. It returns the value the color shall have at this time.

Color request objects have three methods in addition to the normal `raw` and `extend` fields, which contain the values passed to `ColorProp_NewRequest`. The method `start` returns the time at which the interpolation shall start, the method `dur` returns its desired duration, and the method `value` performs the actual interpolation. Virtually every request object will be extended to contain some additional fields, such as the target value of the interpolation.

A request object `r` can be added to the request queue of a synchronous behavior `b` by calling

[5] RGB stands for "Red-Green-Blue". An RGB value consists of a red, a green, and a blue component, indicating the intensity of these primary colors.

[6] HSV stands for "Hue-Saturation-Value". An HSV value consists of a hue, a saturation, and a value component. The hue component identifies a rainbow color, and the saturation and value components indicate how much white and black is mixed into this rainbow color.

`b.addRequest(r)`. The time intervals of requests in the request queue must not overlap.

Let us extend the type of synchronous color behaviors by adding a new method `hsvLinChangeTo` to it. This method shall have the same signature as `rgbLinChangeTo`, but shall interpolate on a straight line through HSV space instead of RGB space.

In order to do this, we have to replace the functions `ColorProp_NewSync` and `ColorProp_NewSyncBeh` by new versions, which create the extended version of the color behavior. So, we have to "re-open" the `ColorProp` module and add to it:

```
module MyColorProp for ColorProp;

let valueMeth =
  meth (self, startcol, reltime)
    let frac = (if (self.dur() isnot 0.0) then
                  (reltime - self.start()) / self.dur();
                else
                  1.0;
                end);
    let h = color_h(startcol), s = color_s(startcol), v = color_v(startcol);
    color_hsv (h + ((self.h - h) * frac),
               s + ((self.s - s) * frac),
               v + ((self.v - v) * frac));
  end;

let hsvLinChangeTo =
  meth (self, col, start, dur)
    let c = ColorProp_NewConst (col).get(); (* make the 1st arg overloaded *)
    self.addRequest (ColorProp_NewRequest(start, dur, valueMeth).extend(
                          {h => color_h(c),
                           s => color_s(c),
                           v => color_v(c)}));
  end;

let NewSync =
  proc (ah, col)
    let pv = ColorProp_NewSync(ah, col);
    pv.setBeh(pv.getBeh().extend ({hsvLinChangeTo => hsvLinChangeTo}));
    pv;
  end;

let NewSyncBeh =
  proc (ah, col)
    ColorProp_NewSyncBeh(ah, col).extend({hsvLinChangeTo => hsvLinChangeTo});
  end;

end module;
```

(Program 15)

`valueMeth` is the interpolation method. It takes three arguments: `self`, `startcol`, the initial value of the color, and `reltime`, the time that has elapsed since the animation handle was signaled. The request object `self` contains the usual methods `start`, `dur`, and `value`; it also contains three extra fields `h`, `s`, and `v`, which hold the hue, saturation, and value of the target color of the interpolation. `valueMeth` first computes what fraction of the interpolation has already elapsed. If the interpolation has a zero duration, then this fraction is assumed to be 1. Next, `valueMeth` extracts the `h`, `s`, and `v` components of the initial color. Finally, it interpolates between the `h`, `s`, and `v` components of the initial and the final color, and returns the resulting intermediate color.

The method `hsvLinChangeTo` takes a behavior `self`, a target color `col`, a start time `start`, and a duration `dur`. We would like to allow `col` to be overloaded, that is, to be either a `Color` or a `Text` denoting a color. But as the operations in the `color` module cannot deal with such overloading, we have first to convert `col` into a value that is guaranteed to be a color. We do so by using the overloaded function `ColorProp_NewConst` to create a constant `ColorPropVal`, and then extract its current value. Then, we call `ColorProp_NewRequest` to create a new request object with start time `start`, duration `dur`, and interpolation method `valueMeth`. We extend this object by three fields `h`, `s`, and `v`, which contain the hue, saturation, and value components of `col`, and finally we return the extended object.

The last two procedures `NewSync` and `NewSyncBeh` replace the default procedures for creating synchronous color property values and behaviors. `NewSync` first invokes the original version of `NewSync` to create a new property value `pv`. It then extracts the behavior of `pv`, extends this behavior by a new method `hsvLinChangeTo`, and replaces the original behavior with the extended one. Finally, it returns `pv`.

`NewSyncBeh` simply invokes the original version of `NewSyncBeh` to create a new synchronous behavior, extends this behavior by a new method `hsvLinChangeTo`, and returns the extended behavior.

Ending the module causes `NewSync` and `NewSyncBeh` to be accessible as `ColorProp_NewSync` and `ColorProp_NewSyncBeh`, thereby truly hiding the default procedures.

We can test the extended version of color property values with the following program:

```
let root = RootGO_NewStd();
let ball = SphereGO_New ([0,0,0], 0.5);
root.add (ball);
let ah = AnimHandle_New();
let col = ColorProp_NewSync (ah, "yellow");
SurfaceGO_SetColor (ball, col);
col.getBeh().hsvLinChangeTo ("blue", 0, 2);
ah.animate ();
```

(Program 15— continued)

This program creates a root object, which contains a sphere. Attached to the sphere is a synchronous color property value, which is initially yellow. This color property value differs from the normal ones in that its behavior understands one extra method, namely `hsvLinChangeTo`. We send such a message to the behavior, asking it to turn blue, over the course of two seconds. If we had used the `rgbLinChangeTo` request, the color would have changed from yellow over grey to blue. The `hsvLinChangeTo` message, however, causes it to move "through the rainbow", that is, from yellow over green to blue.

2.17 Depth Cueing

Depth cueing (also known as "fog") simulates the effects of atmosphere and the dimming of objects as they become more distant. Objects "fade" towards a color, called the *depth cue color*, the further they are away from the viewer.

Depth cueing can be switched on or off. If it is on, its effect is determined by five parameters: a *front plane* P_f, a *back plane* P_b, a *front scale factor* S_f, a *back scale factor* S_b, and a *depth cue color* C_d.

The part of the scene that is displayed by a given root object is surrounded by a "bounding box"[7]. The front face of the bounding box is orthogonal to the viewing direction of the camera. The box is treated as a unit cube; the z-coordinate of its front face is 1, and the z-coordinate of its back face is 0. P_f and P_b are two real numbers that describe the z-coordinates of two planes within this unit cube, so they can range between 0 and 1.

Given an object (or a part of an object) whose z-coordinate (normalized to the bounding box) is z and whose reflected color (i.e. the color determined by lighting and shading computations) is C_r, the color after depth cueing of this object is $\alpha C_r + (1-\alpha)C_d$, where:

$$\alpha = \begin{cases} S_b & \text{if } z \leq P_b \text{ (i.e. the part is behind the back plane)} \\ S_b + (S_f - S_b)\frac{z-P_b}{P_f-P_b} & \text{if } P_b < z < P_f \\ S_f & \text{if } P_f \leq z \end{cases}$$

Figure 17 plots the value of α as a function of z.

Depth cueing is associated with a root object. It affects all the objects shown in the window that corresponds to this root. C_d is controlled by the color property named `RootGO_Depthcue-Color`; P_f and P_b are controlled by the real properties `RootGO_DepthcueFrontPlane` and `RootGO_DepthcueBackPlane`; S_f and S_b are controlled by the real properties `RootGO_DepthcueFrontScale` and `RootGO_DepthcueBackScale`; and whether depth cueing is in effect or not is controlled by the boolean property named `RootGO_DepthcueSwitch`.

The following program shows how depth cueing can be used. It also makes clear that, because depth cueing is controlled by properties, all the standard mechanisms for animation

[7] In the current implementation, this bounding box is more conservative than necessary.

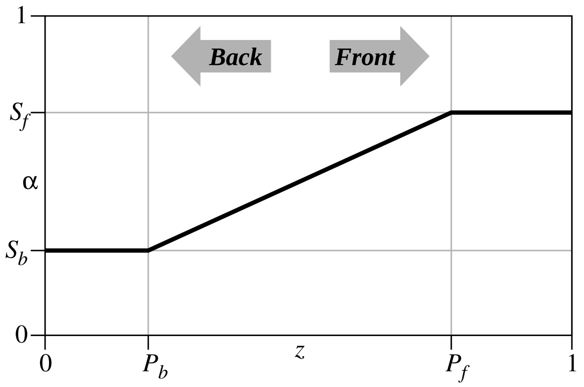


Figure 17: Relationship between fading factor α and depth z in depth cueing

are available to control it. The program constructs a scene that shows a cone. Depth cueing is switched on, and the back scaling factor is made to oscillate between 0 (far objects fade almost fully into black) and 1 (objects do not fade away).

```
let root = RootGO_NewStd();
root.add (ConeGO_New([0,0,0],[0,0,50],1));
let bs = RealProp_NewAsync(meth(self,time) 0.5 + math_sin(time) * 0.5 end);
RootGO_SetDepthcueBackScale(root, bs);
RootGO_SetDepthcueSwitch(root, true);
```

(Program 16)

2.18 Light Objects

In Section 2.2 and again in Section 2.15, we mentioned that the function `RootGO_NewStd()` creates two light sources along with a root object, and adds these light sources to the new root.

Light sources are treated as ordinary graphical objects: they can be added to groups, properties can be attached to them, and their effect is controlled by properties.

In particular, light sources are affected by the `GO_Transform` property. This feature allows us make a light source part of a more complex graphical object, say a car. Moving the car by changing a transformation property attached to it moves the light just like the other parts of the car. The following program illustrates the point:

Figure 18: Snapshots of the animation created by Program 17

```
let root = RootGO_NewStd();
root.remove(root.findName("default-vector-light"));
root.add(CylinderGO_New([0,0,-0.5],[0,0,0.5],1));
let g = GroupGO_New();
root.add(g);
g.add(ConeGO_New([0.5,0,0],[0,0,0],0.2));
g.add(SpotLightGO_New("white",[0,0,0],[0.5,0,0],1,1,0.5,0.5));
let rot = TransformProp_NewAsync (meth(self, time)
                                    Matrix4_RotateZ(Matrix4_Id, time)
                                  end);
GO_SetTransform(g, rot);
```

(Program 17)

First, we call `RootGO_NewStd()` to create a root object `root`. This call also creates an ambient and a vector light source. We remove the vector light source (which is named `"default-vector-light"`) from `root`, and add a cylinder to `root`.

Next, we create a group object `g`, add `g` to `root`, and add a cone and a spot light source to `g`. The tip and the central axis of the cone object and the light cone emitted by the spot light coincide.

Finally, we create an asynchronous transformation property value, which continuously rotates around the z axis, and attach it to `g`. This causes `g` and all objects contained in it (i.e. the cone and the spot light) to rotate around the z axis.

The animation created by this program shows that the cone of light emitted by the spot light sweeps across the walls of the cylinder. Figure 18 shows a series of snapshots of the animation in progress.

2.19 Camera Objects

In Section 2.2 and again in Section 2.15, we mentioned that the function `RootGO_NewStd()` creates not only a root object, but also a camera object. This camera is used to view the graphical objects that occur below the root node in the scene graph.

Cameras (just like light sources) are treated as graphical objects. They can be added to groups, properties can be attached to them, and their appearance is controlled by properties. The camera that is used to view a scene can be, but does not have to be, a descendant of a root object. However, if it is a descendant of a root node, there must be a unique path from the root node to the camera. The camera "inherits" the current values of properties that are attached along the path from the root to the camera, just like any other graphical object does (see Section 2.3).

In particular, cameras are affected by the `GO_Transform` property. This feature allows us to make a camera part of a more complex graphical object, say an airplane. Moving the plane by changing a transformation property attached to it moves the camera as well. This allows us to view a scene from inside the planes's cockpit.

The following program illustrates the concept:

```
let root = RootGO_NewStd();
root.add (SphereGO_New ([0,1,0],0.3));
root.add (SphereGO_New ([1,0,0],0.3));
root.add (SphereGO_New ([0,-1,0],0.3));
root.add (SphereGO_New ([-1,0,0],0.3));
var cam1 = root.findName ("default-camera");
var cam2 = PerspCameraGO_New ([0,0,0],[0.5,0,0],[0,1,0],1.0);

let g = GroupGO_New();
root.add (g);
g.add (ConeGO_New ([0.5,0,0],[0,0,0],0.2));
g.add (cam2);

let rot = TransformProp_NewAsync (meth (self, time)
                                    Matrix4_RotateZ (Matrix4_Id, time)
                                  end);
GO_SetTransform (g, rot);

let cb = KeyCB_New(meth (self, kr)
                     if (kr.change is "c") and (kr.wentDown is true) then
                       let tmp = cam1; cam1 := cam2; cam2 := tmp;
                       root.changeCamera(cam1);
                     end
                   end);
root.pushKeyCB(cb);
```

(Program 18)

The first five lines create a root object and four spheres that are arranged in a diamond formation. The next line finds the default camera that was created by `RootGO_NewStd()`, and binds it to the variable `cam1`. The following line creates a new perspective camera, and binds it to the variable `cam2`.

The next next four lines create a group `g`, add `g` to the root object, and add a cone and `cam2` to `g`. The tip of the cone coincides with the location of `cam2`, and the camera views along the central axis of the cone.

Next, we create an asynchronous transformation property value, which continuously rotates

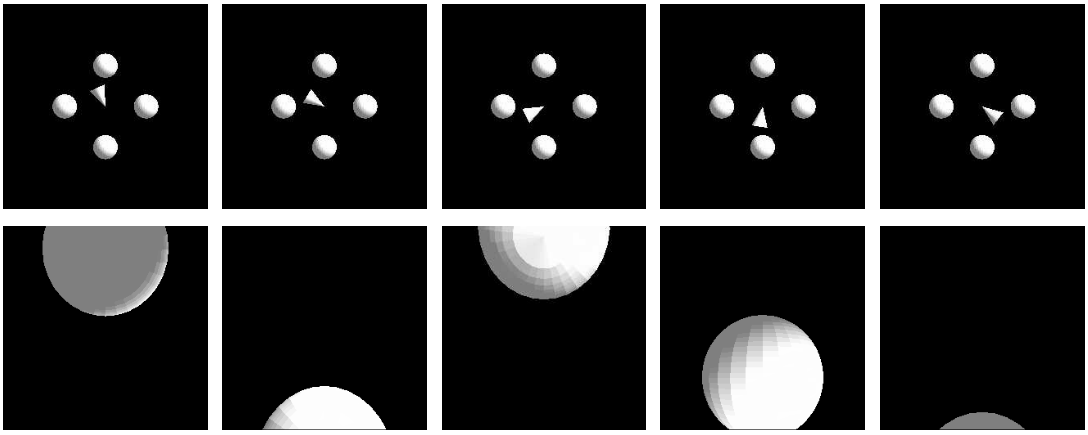


Figure 19: Snapshots of the animation created by Program 18

around the z axis, and attach it to `g`. This causes `g` and all objects contained in it (i.e. the cone and the camera) to rotate around the z axis.

Finally, we create a key callback object, and push it onto the root object's key callback stack. The `invoke` method of the callback object checks whether the 'c' key was pressed, and if so, toggles between the stationary camera and the rotating camera.

Figure 19 shows a series of snapshots of the animation in progress. The upper row of pictures shows the scene as observed by the stationary camera; the lower row shows the scene as observed by the rotating camera.

Program 18 demonstrated one way to move a camera: by modifying a transformation property that affects the camera. But all the relevant parameters of a camera, including its location and its target point (the point the camera is looking at) are controlled by properties, so instead of changing a transformation matrix, we can change the values of these properties.

The following program demonstrates this technique. It creates a scene that contains a stationary torus, and a cone that moves around in a circle. On its way, the cone passes through the torus; it "jumps through the hoop", so to speak. A perspective camera is mounted at the tip of the cone, and is set to point in the direction of movement. The program allows the user to switch between the stationary default camera and the camera mounted at the tip of the cone.

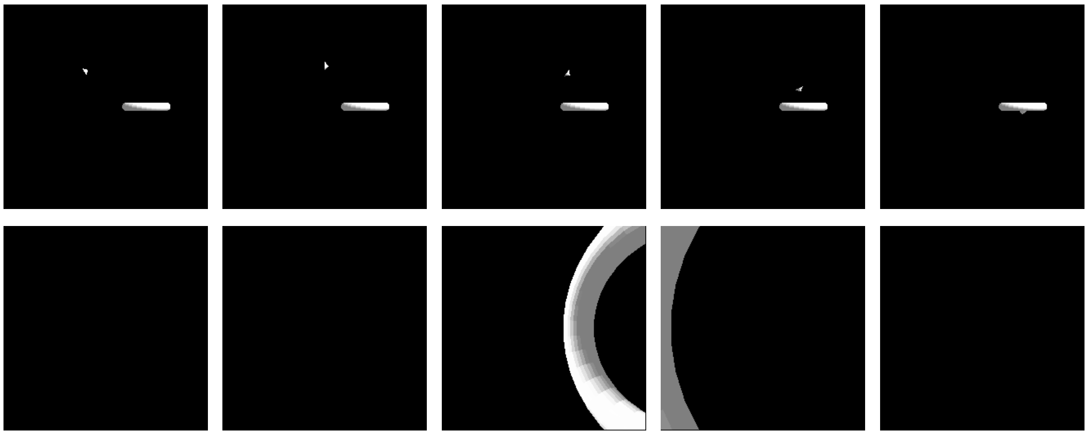


Figure 20: Snapshots of the animation created by Program 19

```
let p1 = PointProp_NewAsync (meth (self, time)
                               [math_sin(time), math_cos(time), 0]
                             end);
let p2 = PointProp_NewAsync (meth (self, time)
                               [math_sin(time+0.1),  math_cos(time+0.1), 0]
                             end);
let p3 = PointProp_NewDep (meth (self, time)
                             let v1 = p1.value(time), v2 = p2.value(time);
                             Point3_Plus (v2, Point3_Minus (v2, v1))
                           end);

let r = RootGO_NewStd ();
r.add (TorusGO_New([1,0,0],[0,1,0],0.5,0.1));
r.add (ConeGO_New (p1, p2, 0.1));

var cam1 = r.findName ("default-camera");
var cam2 = PerspCameraGO_New(p2,p3,[0,0,1],1);

r.pushKeyCB(KeyCB_New(meth (self, kr)
                        if (kr.change is "c") and (kr.wentDown is true) then
                          let tmp = cam1; cam1 := cam2; cam2 := tmp;
                          r.changeCamera(cam1);
                        end
                      end));
```

(Program 19)

The first few lines of the program create three point property values `p1`, `p2`, and `p3`. `p1` and `p2` rotate around the origin of the coordinate system, at a radius of 1 unit. The angle between the two points is 0.1 degrees radian. The third point is a point reflection of `p1` on `p2`.

The next three lines create a root object, and add a cone and a torus to it. The parameters of the torus are constant, while the base of the cone is controlled by `p1`, and its tip is controlled by `p2`.

Next, a perspective camera is created. Its location is controlled by `p2`, and its target point is controlled by `p3`.

The last few lines are identical to those of Program 18; they attach a callback object to the root that allows the user to toggle between the stationary and the moving camera.

Figure 20 shows a series of snapshots of the animation in progress. The upper row of pictures shows the scene, when viewed from the stationary camera; the lower row shows the scene, when viewed from the rotating camera.

2.20 Creating Root Objects

In all examples so far, we have used the function `RootGO_NewStd()` to create new root objects. `RootGO_NewStd()` calls a function in the underlying Modula-3 layer, which creates a graphics base, a perspective camera, and a root object, connects the root object to the graphics base, and attaches the camera to the root such that all objects in the scene graph below the root are viewed through the camera. It also creates an ambient and a vector light source, and adds them to the root. Finally, it creates mouse and position callback objects that allow the user to interactively manipulate the scene, that is, move, scale and rotate it.

Although `RootGO_NewStd()` is defined in the Modula-3 layer, there is no "magic" involved in it. At this point, we have encountered all the machinery needed to reimplement the function at the Obliq level. We start out by reopening the `RootGO` module, and by redefining the function `NewStd`:

```
module RootGOExtension for RootGO;

let NewStd =
  proc ()
    let base = X`PEX`Base_New ("Anim3D Viewer", 10, 10, 500, 500);
    let cam = PerspCameraGO_New ([0, 0, 100], [0, 0, 0], [0, 1, 0], 0.05);
    cam.setName ("default-camera");
    let root = RootGO_New (cam, base);
    let light1 = AmbientLightGO_New ("white");
    light1.setName ("default-ambient-light");
    root.add (light1);
    let light2 = VectorLightGO_New ("white", [-1, -1, -1]);
    light2.setName ("default-vector-light");
    root.add (light2);
    GO_SetTransform (root, TransformProp_NewConst (Matrix4_Id));
```

(Program 20)

```
    let posMeth =
      meth (self, pr)
        let isin =
          proc (e, l)
            var res = false;
            foreach x in l do res := res or (x is e) end;
            res
          end;
        let dx = real_float (pr.pos[0] - self.pos[0]);
        let dy = real_float (pr.pos[1] - self.pos[1]);
        let beh = GO_GetTransform (root).getBeh();
        if isin ("Shift", pr.modifiers) then
          if self.but is "Left" then
            beh.translate (dx * 0.01, (-dy) * 0.01, 0.0);
          elsif self.but is "Middle" then
            beh.scale (1.0 + dx * 0.01, 1.0 + dx * 0.01, 1.0 + dx * 0.01)
          elsif self.but is "Right" then
            beh.translate (0.0, 0.0, dx * 0.01)
          end;
        else
          if self.but is "Left" then
            beh.rotateX (dx * 0.01);
          elsif self.but is "Middle" then
            beh.rotateY (dx * 0.01);
          elsif self.but is "Right" then
            beh.rotateZ (dx * 0.01);
          end;
        end;
        self.pos := pr.pos;
      end;
    let posCB = PositionCB_New (posMeth).extend ({pos => ok, but => ok});
    let mouseMeth =
      meth (self, mr)
        if mr.clickType is "FirstDown" then
          posCB.pos := mr.pos;
          posCB.but := mr.change;
          root.pushPositionCB (posCB);
        elsif mr.clickType is "LastUp" then
          root.popPositionCB ();
        end;
      end;
    root.pushMouseCB (MouseCB_New (mouseMeth));
    root
  end;

end module;
```

(Program 20— continued)

The first few lines create an `X'PEX'Base` object and a perspective camera named `"default-camera"`. Next, a `RootGO` object `root` is created. The `RootGO` creation function takes two arguments: the `X'PEX'Base` (meaning all graphical objects below `root` are displayed in an X window, using the PEX 3D protocol), and the camera (meaning all graphical objects below `root` are viewed through this camera).

Next, an ambient light source named `"default-ambient-light"` and a vector light source named `"default-vector-light"` are created, and added to the root object. This causes all objects below root to be illuminated by these two light sources.

Next, a constant transformation property value is created and attached as the `GO_Transform` property to `root`.

Next, a position callback object and a mouse callback object are created. The mouse callback object is immediately pushed onto `root`'s mouse callback stack, whereas the position callback object is assigned to the variable `posCB`. Finally, the new object is returned.

The `invoke` method (see Section 2.13) of the mouse callback object is called when a mouse button is pressed or released. It checks whether the button is the first button to go down, or the last button to go up. In the first case, it saves the button and the current mouse position in two fields of `posCB`, and pushes `posCB` onto `root`'s position callback stack. In the second case, it pops `posCB` from the `root`'s position callback stack. So, the `invoke` method of `posCB` will be called only when a mouse button is pressed.

In order for a callback object to be active, it must be on top of a callback stack. So, `posCB` is active only when it is on top of `root`'s position callback stack, that is, only if a mouse button is pressed. In this case, its `invoke` method is called whenever the position of the mouse changes. `posCB` checks whether the shift key is held down or not, and whether the left, middle, or right mouse button is pressed, and changes the value of `root`'s `GO_Transform` property accordingly. Holding the left button while moving the mouse causes a rotation around the X axis, holding the middle button causes a rotation around the Y axis, and holding the right button causes a rotation around the Z axis. The angle of the rotation depends on how much the mouse was moved in the x dimension. If the shift key is pressed, holding the left button while moving the mouse causes a translation in the XY plane (depending on the xy movement of the mouse), holding the middle button causes a uniform scaling, and holding the right button causes a translation in the Z direction (both depending on the x movement of the mouse).

This version of `RootGO_NewStd` is completely equivalent to the default one. In other words, including this code fragment before any of the programs we have encountered before would cause no observable difference in their behavior.

So, this function is not interesting *per se*, but because it can be used as a template to create alternative `RootGO` creation functions with slightly different behavior. As an exercise, the reader might try to construct a `RootGO` creation function whose callback functions transform all the objects in the scene, except the light sources.

3 Reference Manual

The remainder of this report contains a reference manual for Obliq-3D. As in the first part, we assume the reader is familiar with Obliq [3]. Furthermore, we assume that the reader has read the tutorial part of the report, and understands the basic concepts of Obliq-3D.

The reference manual contains a section for each module. We show the interface description provided by the on-line help system (see Section 2.4), and we explain the types and functions exported by the module in more detail. Explanations are terse, but contain references to the tutorial part where appropriate.

Notation for Obliq Types

To paraphrase Cardelli, "Although Obliq is an untyped language, every Obliq program, like any other program, respects the type discipline in the programmer's mind." [3] Obliq enables the programmer to make this discipline explicit by allowing for *type comments*, which are parsed according to a fixed grammar, but have no effect after parsing. This type notation is described in the Obliq report [3]. The following section excerpts the pieces relevant to Obliq-3D:

`Top` ▷ The type of all values.

`Ok` ▷ Contains a single value, `ok`. This value should be considered as having every type, so it can be used to initialize variables; however, its normal type is `Ok`.

`Bool` ▷ The type of boolean values. Contains the two values `true` and `false`.

`Int` ▷ The type of integer values.

`Real` ▷ The type of real values.

`Text` ▷ The type of text values.

`Color` ▷ The type of color values.

`Mutex` ▷ The type of mutex values.

`Exception` ▷ The type of exceptions.

$A_1 + A_2$ ▷ The *union* of the types A_1 and A_2. This is an extension of the notation presented in [3].

A_1 & A_2 ▷ The *concatenation* of the object types A_1 and A_2. This is an extension of the notation presented in [3].

`[A]` ▷ The type of arrays of `A`s.

`[n*A]` ▷ The type of arrays of `A`s of length `n` (where `n` is an integer).

$\mathtt{(A_1,...,A_n)\text{->}A\ !\ exc_1...exc_m}$	$\triangleright$ The type of procedures of argument types $\mathtt{A_i}$ ($\mathtt{n} \geq 0$), result type $\mathtt{A}$, and exceptions $\mathtt{exc_i}$ (where $\mathtt{!\ exc_1...exc_m}$ may be omitted).
$\mathtt{(A_1,...,A_n)\text{=>}A\ !\ exc_1...exc_m}$	$\triangleright$ The type of methods of argument types $\mathtt{A_i}$ ($\mathtt{n} \geq 0$), result type $\mathtt{A}$, and exceptions $\mathtt{exc_i}$ (where $\mathtt{!\ exc_1...exc_m}$ may be omitted). The type of the self argument is not included in $\mathtt{A_i}$.
$\{\mathtt{x_1:A_1,...,x_n:A_n}\}$	$\triangleright$ The type of objects with components named $\mathtt{x_i}$ of field type or method type $\mathtt{A_i}$.
$\mathtt{Self(X)B}\{\mathtt{X}\}$	$\triangleright$ Where $\mathtt{B}\{\mathtt{X}\}$ is a method type with possible covariant occurrences of $\mathtt{X}$. This construction is used to give a name ($\mathtt{X}$) to the type of the method's self (e.g. for methods that return self) This is a modification of the notation presented in [3].
$\mathtt{All(X\text{<:}A)B}\{\mathtt{X}\}$	$\triangleright$ Where $\mathtt{B}\{\mathtt{X}\}$ is any type with possible occurrences of $\mathtt{X}$. This is the type of values that, for all subtypes $\mathtt{A_0}$ of $\mathtt{A}$, have type $\mathtt{B}\{\mathtt{A_0}\}$. If $\mathtt{\text{<:}A}$ is omitted, it stands for $\mathtt{\text{<:}Top}$.

`A<:B` is read as "`A` is a subtype of `B`". It implies that every value of type `A` is also a value of type `B`. In particular, `A<:Top` holds for any type `A`, and `B<:{}` holds for any object type `B`.

The notation for giving the type of a procedure within an interface is as follows:

$\mathtt{p(x_1:A_1,...,x_n:A_n):A\ !\ exc_1...exc_m}$

indicates that procedure `p` has formal parameters $\mathtt{x_i}$ of type $\mathtt{A_i}$ ($\mathtt{n} \geq 0$), result type `A`, and exceptions $\mathtt{exc_i}$ (where $\mathtt{!\ exc_1...exc_m}$ may be omitted).

We extended the type notation of [3] with two new type operators: the type union operator $+$ and the object type concatenation operator $\&$.

The *type union operator* is used to describe ad-hoc polymorphism. The type `A+B` denotes the *union* of the types `A` and `B`. A value of this type belongs to either `A` or `B`. We can use the type union operator to describe the type of the overloaded function `real_float`:

```
real_float(n:Int+Real):Real
```

Note that Obliq provides no construct for discriminating between types. So, it is not possible for the user to define overloaded procedures that work for arbitrary type unions; all overloaded procedures eventually have to invoke some built-in facilities to discriminate between the possible types.

The *object type concatenation operator* is used to describe the type aspects of operations that merge several objects. The type `A&B` denotes the *concatenation* of the object types `A` and B. If $A=\{x_1:A_1,...,x_m:A_m\}$, $B=\{y_1:B_1,...,y_n:B_n\}$, and $x_i \neq y_j$ for all i, j, then $A \& B = \{x_1:A_1,...,x_m:A_m,y_1:B_1,...,y_n:B_n\}$.

We can use the object type concatenation operator to give a type to the `clone` construct: If each of the expressions x_i is of type A_i<:{} (for $1 \leq i \leq n$), then the expression $clone(x_1,...,x_n)$ is of type $A_1 \& ... \& A_n$.

Coercion Functions for Overloaded Procedures and Methods

Many of the functions and methods of Obliq-3D are overloaded, that is, the same formal parameter can be bound to arguments of different types. We introduced the type union notation to be able to describe this overloading.

It turns out that all our uses of overloading are uniform enough to allow us to define a very simple way for resolving overloading. For each type union $A_1 + ... + A_n$, we declare one of the A_i (by convention A_1) to be the *desired type*, and we give *coercion functions* from $A_2, \cdots, A_n$ to A_1. Below are the union types used in Obliq-3D, and the corresponding coercion functions:

`Num = Real + Int`
    `Int` is coerced to `Real` via `real_float`
`Col = Color + Text`
    `Text` is coerced to `Color` via `color_named`
`BooleanVal = BooleanPropVal + Bool`
    `Bool` is coerced to `BooleanPropVal` via `BooleanProp_NewConst`
`RealVal = RealPropVal + Num`
    `Num` is coerced to `RealPropVal` via `RealProp_NewConst`
`ColorVal = ColorPropVal + Col`
    `Col` is coerced to `ColorPropVal` via `ColorProp_NewConst`
`PointVal = PointPropVal + Point3`
    `Point3` is coerced to `PointPropVal` via `PointProp_NewConst`
`TransformVal = TransformPropVal + Matrix4`
    `Matrix4` is coerced to `TransformPropVal` via `TransformProp_NewConst`
`LineTypeVal = LineTypePropVal + LineType`
    `LineType` is coerced to `LineTypePropVal` via `LineTypeProp_NewConst`
`MarkerTypeVal = MarkerTypePropVal + MarkerType`
    `MarkerType` is coerced to `MarkerTypePropVal` via `MarkerTypeProp_NewConst`
`RasterModeVal = RasterModePropVal + RasterMode`
    `RasterMode` is coerced to `RasterModePropVal` via `RasterModeProp_NewConst`
`ShadingVal = ShadingPropVal + Shading`
    `Shading` is coerced to `ShadingPropVal` via `ShadingProp_NewConst`

3.1 The Point3 Module

```
- help Point3;
  Point3_Plus(a b: Point3): Point3
  Point3_Minus(a b: Point3): Point3
  Point3_ScaleToLen(a: Point3, s: Num): Point3
  Point3_TimesScalar(a: Point3, s: Num): Point3
  Point3_Length(a: Point3): Real
  Point3_Distance(a b: Point3): Real
  Point3_MidPoint(a b: Point3): Point3
WHERE
  Point3 = [3*Num]
  Num = Real + Int
```

A `Point3` is an array of three numbers, where a number is an integer or a real. One can construct a new point by writing an array constant: $[x,y,z]$ denotes a point at position (x, y, z) in cartesian 3-space. Conversely, the components of a point can be accessed through Obliq's standard array indexing mechanism: given a point `a` of value $[x,y,z]$, `a[0]` evaluates to x, `a[1]` evaluates to y, and `a[2]` evaluates to z. The type `Point3` is also used to denote vectors.

The following functions are provided by the `Point3` module:

- `Point3_Length(a)` returns the length of the vector `a`.
- `Point3_Plus(a,b)` adds the two vectors `a` and `b`.
- `Point3_Minus(a,b)` subtracts vector `b` from vector `a`.
- `Point3_Distance(a,b)` returns the distance between the points `a` and `b`, that is, it is equivalent to `Point3_Length(Point3_Minus(a,b))`.
- `Point3_TimesScalar(a,s)` multiplies the vector `a` with the scalar `s`.
- `Point3_ScaleToLen(a,s)` returns a vector parallel to `a` of length `s`, that is, it is equivalent to `Point3_TimesScalar(a,real_float(s)/Point3_Length(a))`.
- `Point3_MidPoint(a,b)` returns the midpoint between the two points `a` and `b`, that is, it is equivalent to `Point3_TimesScalar(Point3_Plus(a,b),0.5)`.

3.2 The Matrix4 Module

```
- help Matrix4;
  Matrix4_Id: Matrix4
  Matrix4_Multiply(m1 m2: Matrix4): Matrix4
  Matrix4_Translate(m: Matrix4, x y z: Num): Matrix4
  Matrix4_Scale(m: Matrix4, x y z: Num): Matrix4
  Matrix4_RotateX(m: Matrix4, a: Num): Matrix4
  Matrix4_RotateY(m: Matrix4, a: Num): Matrix4
  Matrix4_RotateZ(m: Matrix4, a: Num): Matrix4
WHERE
  Matrix4 is opaque
  Num = Real + Int
```

A `Matrix4` can be viewed as a transformation function that takes a point in 3-space and maps it onto another point in 3-space. Typical transformations are translation, scaling, and rotation. Internally, values of type `Matrix4` are represented as a 4×4 array of reals.

`Matrix4_Id` is the identity transformation, the transformation that maps every point onto itself.

`Matrix4_Multiply(`m_1`,`m_2`)` returns a transformation m that is the composition of m_1 and m_2. Applying m to a point p is the same as first applying m_2 to p and then applying m_1 to the resulting point.

`Matrix4_Translate(`m`,`a`,`b`,`c`)` creates a transformation m' that maps a point $[x,y,z]$ onto a point $[x+a,y+b,z+c]$, composes m' with m, and returns the resulting transformation.

`Matrix4_Scale(`m`,`a`,`b`,`c`)` creates a transformation m' that maps a point $[x,y,z]$ onto a point $[ax,by,cz]$, composes m' with m, and returns the resulting transformation. The scaling operation is called *uniform* if $a = b = c$.

`Matrix4_RotateX(`m`,`a`)` creates a transformation m' that takes a point p and returns p rotated by a degrees (radian) around the x-axis. It then composes m' with m, and returns the resulting transformation. `Matrix4_RotateY(`m`,`a`)` and `Matrix4_RotateZ(`m`,`a`)` perform similar rotations around the y- and z-axis, respectively.

3.3 The Anim3D Module

```
- help Anim3D;
  Anim3D_lock: Mutex
```

The `Anim3D` module provides resources that are global to the entire animation session. There is only one such resource: the mutex `Anim3D_lock`, which is used to ensure that a batch of operations on the scene graph is "atomic" (see Section 2.12).

Our animation library is based on a damage-repair model, where operations on the scene graph "damage" the scene. These damages are repaired by a dedicated animation server thread. Normally, damage-repair is transparent to the client program. However, the animation server thread must acquire `Anim3D_lock` before performing any repairs. So, by locking this mutex, the client program can prevent the animation server from updating the display. This is useful for programs that modify the scene graph such that at some intermediate points the scene graph does not reflect the intended scene.

3.4 The ProxiedObj Module

```
- help ProxiedObj;
TYPE ProxiedObj <: { extend: Self(X) All(Y<:{}) (Y) => X & Y }
       Objects of this type also contain a field "raw",
       which is for internal use only.
```

A `ProxiedObj` is an Obliq object that has a Modula-3 counterpart (see Section 2.14). All animation-specific object types used in Obliq-3D are subtypes of `ProxiedObj`.

Each `ProxiedObj` contains (at least) two fields, `raw` and `extend`. The field `raw` contains an opaque value that in turn contains a reference to the corresponding Modula-3 object. The Modula-3 object contains a reference back to the `ProxiedObj`. The client program should not attempt to modify `raw`.

The method `extend` is used to add additional fields to the object. Given a `ProxiedObj` `p` and an ordinary object `o`, the expression `p.extend(o)` returns a new object `p'`, which is equivalent to the object returned by `clone(p,o)`. In addition, the corresponding Modula-3 object now refers to `p'` instead of `p`.

The client program should never use `clone` on a `ProxiedObj`; calling `extend` is the only legal way to add fields to such an object. Furthermore, after extending a `ProxiedObj` `p`, the original object `p` must not be used any longer.

3.5 The GraphicsBase Module

```
- help GraphicsBase;
TYPE GraphicsBase <: ProxiedObj
```

Associated with each root object is a window on the screen. This window is represented by a `GraphicsBase` object. `GraphicsBase` is a subtype of `ProxiedObj`.

A `GraphicsBase` also serves as an abstraction of the underlying window system and graphics system. Currently, Obliq-3D supports only the X window system [21] and the MPEX graphics library (MPEX is Digital's extension of PEX [8], the 3D extension of X).

`GraphicsBase` is an abstract class; the module does not provide any creation functions.

3.6 The X'Pex'Base Module

```
- help X`PEX`Base;
  X`PEX`Base_Failure: Exception
  X`PEX`Base_New(title: Text, x y w h: Int): X`PEX`Base ! X`PEX`Base_Failure
  X`PEX`Base_NewStd(): X`PEX`Base ! X`PEX`Base_Failure
WHERE
  X`PEX`Base <: GraphicsBase & { changeTitle: (Text) => Ok,
                                 awaitDelete: () => Ok,
                                 destroy: () => Ok }
```

An `` X`PEX`Base `` object represents a 3D rendering window that is displayed using the X window system and the MPEX graphics library.

There are two functions for creating `` X`PEX`Base `` objects: `` X`PEX`Base_New(t,x,y,w,h) `` returns a new `` X`PEX`Base ``, and as a side effect it creates a window of width `w` and height `h` (in pixels), `x` pixels right of and `y` pixels below the upper left corner of the screen. It also puts the text `t` into the title bar of the window. `` X`PEX`Base_NewStd() `` is equivalent to `` X`PEX`Base_New("Anim3D Viewer",10,10,500,500) ``.

`` X`PEX`Base `` objects contain three extra methods: `b.changeTitle(t)` changes the title of the window associated with the base `b` to `t`; `b.awaitDelete()` suspends until the window associated with `b` is deleted, and `b.destroy()` deletes the window associated with `b`.

3.7 The AnimHandle Module

```
- help AnimHandle;
  AnimHandle_New(): AnimHandle
WHERE
  AnimHandle <: ProxiedObj & { animate: () => Ok }
```

An `AnimHandle` (or *animation handle*) is an object that is used to synchronize a set of animation requests. As described in Section 2.10, clients can create property values with synchronous behaviors. Each synchronous behavior is controlled by an animation handle, and many behaviors can be controlled by the same handle. Clients can send animation requests to those behaviors. These requests are not executed immediately, but instead are held in request queues. Requests issued to a behavior are processed only when the animation handle controlling that behavior is signaled.

`AnimHandle_New()` creates a new animation handle and returns it. In addition to the fields common to all subtypes of `ProxiedObj`, an `AnimHandle` has one extra method, `animate`. Given an animation handle `ah`, the call `ah.animate()` signals all the behaviors controlled by `ah` to process their requests.

3.8 The GO Module

```
- help GO;
  GO_PropUndefined: Exception
  GO_StackError: Exception
  GO_Transform: TransformPropName
  GO_SetTransform(go: GO, xf: TransformVal): Ok
  GO_GetTransform(go: GO): TransformPropVal ! GO_PropUndefined
WHERE
  GO <: ProxiedObj &
    { setProp: (PropName,PropVal) => Ok
      unsetProp: (PropName) => Ok ! GO_PropUndefined,
      getProp: (PropName) => PropVal ! GO_PropUndefined,
      setName: (Text) => Ok,
      getName: () => Text,
      findName: (Text) => GO,
      pushMouseCB: (cb: MouseCB) => Ok,
      popMouseCB: () => Ok ! GO_StackError,
      removeMouseCB: (cb: MouseCB) => Ok ! GO_StackError,
      invokeMouseCB: (mr: MouseRec) => Ok,
      pushPositionCB: (cb: PositionCB) => Ok,
      popPositionCB: () => Ok ! GO_StackError,
      removePositionCB: (cb: PositionCB) => Ok ! GO_StackError,
      invokePositionCB: (mr: PositionRec) => Ok,
      pushKeyCB: (cb: KeyCB) => Ok,
      popKeyCB: () => Ok ! GO_StackError,
      removeKeyCB: (cb: KeyCB) => Ok ! GO_StackError,
      invokeKeyCB: (mr: KeyRec) => Ok }
  TransformVal = TransformPropVal + Matrix4
```

A `GO` (or *graphical object*) is an object that describes some part of the scene. Geometric primitives, such as lines, spheres, tori, etc. are special kinds (and therefore subtypes) of graphical objects, and so are light sources and cameras. There is also a subtype of `GO`, namely `GroupGO`, that allows us to combine several graphical objects into a single one, thereby allowing us to impose a hierarchical structure on the scene.

Graphical objects are subtypes of `ProxiedObj`. In addition to the fields and methods common to all `ProxiedObj`'s, each graphical object `g` has a number of additional methods. We can group these methods into three categories: property-related, name-related, and callback-related methods.

How Properties Affect the Scene

A *property* describes some aspect of the appearance of a graphical object, such as its color, its size, or its location. Properties consist of a *name* and a *value*. The name defines what aspect of a graphical object is affected (for example, its color), and the value describes how this aspect appears at a given time.

In order for a property (n, v) to influence the appearance of a graphical object g, (n, v) has to be *attached* to g.

Conceptually, every graphical object contains a property mapping, a partial function from property names to property values. The function is partial, as it may be undefined for some property names. Attaching a property (n, v) to a graphical object means replacing the object's property mapping M by the extended mapping $M[n \to v]$.

Alternatively, we can assume that every graphical object contains an association list. Association lists are lists of name-value pairs such that each name is unique; they are commonly used to implement finite functions. In this model, attaching a property to a graphical object means adding it to the object's association list.

If we view the graphical objects in a scene as nodes, and the containment relationships induced by `GroupGO` objects as arcs, we obtain a directed graph, called a *scene graph*. Scene graphs must be acyclic. If a group g contains another graphical object o, we say that g is a *parent* of o, or conversely, that o is a *child* of g. We use the terms *ancestor* and *descendant* to describe the transitive closures of the *parent* and the *child* relationship, respectively.

Our system uses a damage-repair model. A dedicated thread (called the *animation server thread*) redisplays the scene when it changes. It does so by traversing the scene graph, starting from the roots (the vertices with in-degree 0). So, whenever a graphical object G_1 is drawn, there is a path `[`g_1`,..,`g_n`]` from g_1 to a root g_n (Section 2.5 explains how scene graphs can be "unfolded" into forests of trees).

The animation server has a *state vector*, which contains the current transformation, surface color, line width, etc. For each property name, there is a corresponding element in the state vector. Initially, the state vector contains default values. When the scene is rerendered, and the traversal process reaches a particular graphical object, all properties attached to this object are used to update the state vector. Upon backtracking, the state vector reverts back to its previous state.

When a `GO_Transform` property is encountered, its current value m_1 is multiplied with the current transformation entry m_0 of the state vector, that is, the new entry of the state vector is `Matrix4_Multiply(`m_1`,`m_0`)`. When any other property is encountered, its current value simply replaces the current entry of the state vector.

The effect of all this is that properties that are attached to a graphical object affect the appearance of this object and all those objects contained in it. However, if a descendant of the graphical object has a property with the same name attached to it, then this property will

$\texttt{Val} = \texttt{Bool} + \texttt{Real} + \texttt{Point3} + \texttt{Color} + \texttt{Matrix4} + \cdots$	(ordinary values)
$\texttt{PropMap} = \texttt{PropName} \to \texttt{PropVal}$	(property mappings)
$\texttt{PropVal} = \texttt{Time} \to \texttt{Val}$	(property values – functional view)

D : $\texttt{PropName} \to \texttt{Val}$
$D(n)$ is the default value of property name n

P : $\texttt{GO} \to \texttt{PropMap}$
$P(g)$ is the property mapping of graphical object g

S : $[\texttt{GO}] \to \texttt{PropName} \to \texttt{Time} \to \texttt{Val}$
$S([g_1, \cdots, g_n])(n)(t)$ is the value of property n at graphical object g_1 and time t

S is defined as follows:

$S([\,])(n)(t) = D(n)$ for all n, t

$$S([g_1, \cdots, g_m])(n)(t) = \begin{cases} S([g_2, \cdots, g_m])(n)(t) \\ \quad \text{if } P(g_1)(n) \text{ is undefined} \\ P(g_1)(n)(t) \\ \quad \text{if } P(g_1)(n) \text{ is defined and } n \text{ is not } \texttt{GO\_Transform} \\ \texttt{Matrix4\_Multiply}(S([g_2, .., g_m])(n)(t), P(g_1)(n)(t)) \\ \quad \text{if } P(g_1)(n) \text{ is defined and } n \text{ is } \texttt{GO\_Transform} \end{cases}$$

Table 2: From Properties to State

"override" the value of the ancestor (or, in the case of transformation properties, will modify it).

Table 2 formalizes how the current state is determined. For the purpose of this explanation, we view property values not as objects, but simply as functions from time to some value.

We want to define the *state function* S. S takes a path $[g_1, ..., g_m]$ in the scene graph (represented as a list of graphical objects, with the root being the last element), a property name n, and a time t, and returns the value of n at time t used for rendering object g_1 during the rendering traversal along the path $[g_1, ..., g_m]$.

We define S in terms of two other functions, D and P. D is a function from property names to values; it describes the default values for each property. For instance, the transformation property is by default the identity matrix. P is a function from graphical objects to property mappings. $P(g)$ denotes the property mapping of g, that is, the set of properties attached to g.

Property-Related Methods

The following methods are used to manipulate the property mapping of a graphical object `g`:

- `g.setProp(n,v)` attaches a property with name `n` and value `v` to `g`. If there is already a property named `n` attached to `g`, its value will be replaced by `v`.
- `g.unsetProp(n)` detaches the property named `n` from `g`. If no such property is attached to `g`, the exception `GO_PropUndefined` is raised.
- `g.getProp(n)` checks whether a property named `n` is attached to `g`. If so, the value of the property is returned; otherwise, the exception `GO_PropUndefined` is raised.

Name-Related Methods

Obliq-3D allows the client program to attach a *name*, a value of type `Text`, to a graphical object. Names serve as symbolic labels, they are used to retrieve elements of a scene, without having to traverse the scene graph explicitly.

Here are the name-related methods common to all graphical objects:

- `g.setName(t)` attaches the text `t` as a name to `g` (see Section 2.15). An existing name will be replaced.
- `g.getName(t)` returns the name of `g`, and `ok` if no name is attached.
- `g.findName(t)` traverses the scene graph below `g` to see if `g` or any of its descendants are named `t`. It returns the first graphical object named `t` that it finds; and `ok` if no such object can be found.

Callback-Related Methods

Obliq-3D allows the client program to specify the interactive behavior of individual graphical objects. Input actions (such as key presses or mouse movements) trigger *events*. We distinguish between three different kinds of events: *mouse events* (caused by mouse button transitions), *position events* (caused by mouse movements), and *key events* (caused by keystrokes). Events are handled by *callback objects*, objects that specify what action should be taken in response to an event. There are three types of callback objects (mouse, position, and key callback objects), corresponding to the three kinds of events.

Events occur within the scope of a window w of the screen. The relevant data associated with each event is wrapped into an *event record*, and sent to the root object associated with w. Each graphical object (including root objects) contains three callback object stacks, one for each type of callback object. When a mouse event record `mr` is sent to a graphical object `g`, `g` will check whether it has any callback objects on its mouse callback stack. If so, it sends the message `invoke(mr)` to the topmost object on the stack, otherwise, it ignores the event. Position and key events are treated similarly.

Here are the methods that interact with the callback stacks:

- `g.pushMouseCB(cb)` pushes a mouse callback object `cb` onto the mouse callback stack of `g`.
- `g.popMouseCB()` removes the topmost mouse callback object from the mouse callback stack of `g`. If the stack is empty, the exception `GO_StackError` is raised.
- `g.removeMouseCB(cb)` removes the mouse callback object `cb` from the mouse callback stack of `g`. If `cb` is not contained in the stack, the exception `GO_StackError` is raised.
- `g.invokeMouseCB(mr)` sends the message `invoke(mr)` to the topmost element of the mouse callback stack of `g`.
- `pushPositionCB`, `popPositionCB`, `removePositionCB`, and `invokePositionCB` provide similar functionality for the position callback stack.
- `pushKeyCB`, `popKeyCB`, `removeKeyCB`, and `invokeKeyCB` provide similar functionality for the key callback stack.

Transformations

All graphical objects are affected by a property named `GO_Transform`, which describes how the graphical object is *transformed* relative to its parent(s). `GO_Transform` associates with `TransformPropVal` property values. These property values can compute a *transformation* based on the current time.

A transformation is a mapping from points to points. Particular transformations are *translations*, which move points by a fixed amount, *scalings*, which move points towards or away from the origin by a fixed factor, and *rotations*, which rotate points by a fixed angle around one of the major axes of the coordinate system. Internally, transformations are represented as values of type `Matrix4` (see Section 3.2).

`GO_SetTransform(g,t)` takes a graphical object `g` and a transformation property value `t` (or a matrix, which is coerced into a transformation property value, as described on page 43). Modulo coercion, the expression is equivalent to `g.setProp(GO_Transform,t)`. In other words, the following equivalences hold:

```
GO_SetTransform(g,x) ≡
      g.setProp(GO_Transform,x)
            if x is a TransformPropVal
      g.setProp(GO_Transform,TransformProp_NewConst(x))
            if x is a Matrix4
```

`GO_GetTransform(g,t)` is a shortform for `g.getProp(GO_Transform)`.

3.9 The GroupGO Module

```
- help GroupGO;
  GroupGO_BadElement: Exception
  GroupGO_New(): GroupGO
  GroupGO_NewWithSizeHint(size: Int): GroupGO
WHERE
  GroupGO <: GO & { add: (GO) => Ok,
                    remove: (GO) => Ok ! GroupGO_BadElement,
                    flush: () => Ok,
                    content: () => [GO] }
```

A `GroupGO` (or *group object*) is a graphical object that can contain other graphical objects. Group objects are the basic mechanism for imposing a hierarchical structure onto the graphical objects in a scene.

`GroupGO` is a subtype of `GO`. It adds four extra methods: `add`, `remove`, `flush`, and `content`.

- `g.add(o)` adds a graphical object `o` to the group `g`. Afterwards, we say that `o` is contained in `g`.
- `g.remove(o)` removes a graphical object `o` from the group `g`. If `o` is not contained in `g`, the exception `GO_BadElement` is raised.
- `g.flush()` removes all graphical objects from `g`. `g` is empty afterwards.
- `g.content()` returns an array of the graphical objects contained in `g`.

`GroupGO_New()` creates a new group object and returns it. Initially, the group is empty.

Internally, a group object contains an array of pointers to its children. When a group is created, a small initial array (5 elements) is allocated; when the array fills up, it is automatically grown.

`GroupGO_NewWithSizeHint(i)` also creates and returns a new group. As a performance optimization, it allows the client program to request the array of children to be `i` elements wide, potentially avoiding the need for having to grow the array later on.

3.10 The RootGO Module

```
- help RootGO;
  RootGO_New(cam: CameraGO, base: GraphicsBase): RootGO
  RootGO_NewStd(): RootGO
  RootGO_NewStdWithBase(base: GraphicsBase): RootGO
  RootGO_Background: ColorPropName
  RootGO_DepthcueSwitch: BooleanPropName
  RootGO_DepthcueColor: ColorPropName
  RootGO_DepthcueFrontPlane: RealPropName
  RootGO_DepthcueBackPlane: RealPropName
  RootGO_DepthcueFrontScale: RealPropName
  RootGO_DepthcueBackScale: RealPropName
  RootGO_SetBackground(go: GO, c: ColorVal): Ok
  RootGO_SetDepthcueSwitch(go: GO, b: BooleanVal): Ok
  RootGO_SetDepthcueColor(go: GO, c: ColorVal): Ok
  RootGO_SetDepthcueFrontPlane(go: GO, r: RealVal): Ok
  RootGO_SetDepthcueBackPlane(go: GO, r: RealVal): Ok
  RootGO_SetDepthcueFrontScale(go: GO, r: RealVal): Ok
  RootGO_SetDepthcueBackScale(go: GO, r: RealVal): Ok
WHERE
  RootGO <: GroupGO & { changeCamera: (CameraGO) => Ok }
  BooleanVal = BooleanPropVal + Bool
  RealVal = RealPropVal + Real + Int
  ColorVal = ColorPropVal + Color + Text
```

A `RootGO` (or *root object*) is a graphical object that forms a root in the scene graph.

Associated with each root object `r` is a `GraphicsBase` (see Section 3.5), which represents a window on the screen, and a connection to the window system and to the graphics system.

Also associated with each root object `r` is a camera object `cam` (see Section 3.16). The window associated with `r` shows the part of the scene graph below `r`, as seen through `cam`. The camera can be, but does not have to be, part of the scene graph. If it is part of the scene graph, then there must be a unique path from `r` to `cam`. Transformation properties along this path affect the location and orientation of the camera.

There are three `RootGO` creation functions:

- `RootGO_New(cam,base)` creates and returns a new root object that is associated with a graphics base `base`, and views the scene through a camera `cam`.
- `RootGO_NewStd()` creates and returns a new root object. The root object is associated with the graphics base created by `X'PEX'Base_NewStd()`. The camera is set to be

`PerspCameraGO_New([0,0,100],[0,0,0],[0,1,0],0.05)`. In addition, an ambient light source and a vector light source are created and added to the root, and mouse and position callbacks for rotating, translating, and scaling the scene are attached to the root. Program 20 shows how `RootGO_NewStd` could be implemented.

- `RootGO_NewStdWithBase(base)` is similar to `RootGO_NewStd()`, but allows the client program to supply the graphics base.

`RootGO` is a subtype of `GroupGO`, and thus supports all the group-specific methods. In particular, it allows the client program to add other graphical objects to the root object. In addition, it provides one extra method: `root.changeCamera(cam)` changes the camera through which the scene is viewed to be `cam`.

Root objects, like all graphical objects, are affected by the `GO_Transform` property. In addition, there are a few more properties that affect root objects:

- The color property named `RootGO_Background` controls the color of the background of the window associated with the root object. By default, this color is black.
- The boolean property named `RootGO_DepthcueSwitch` controls whether depth cueing (see Section 2.17) is used or not. By default, depth cueing is off.
- The color property named `RootGO_DepthcueColor` controls the depth cue color. By default, this color is black.
- The real property named `RootGO_DepthcueFrontPlane` controls the front plane z-value P_f. By default, P_f is 1.
- The real property named `RootGO_DepthcueBackPlane` controls the back plane z-value P_b. By default, P_b is 0.
- The real property named `RootGO_DepthcueFrontScale` controls the front scale factor S_f. By default, S_f is 1.
- The real property named `RootGO_DepthcueBackScale` controls the back scale factor S_b. By default, S_b is 0.

`RootGO_SetBackground`, `RootGO_SetDepthcueSwitch`, `RootGO_SetDepthcueColor`, `RootGO_SetDepthcueFrontPlane`, `RootGO_SetDepthcueBackPlane`, `RootGO_SetDepthcueFrontScale` and `RootGO_SetDepthcueBackScale` are overloaded convenience procedures for attaching the line-specific properties to graphical objects; they are similar to `GO_SetTransform` (see Section 3.8).

3.11 The LightGO Module

```
- help LightGO;
  LightGO_Color: ColorPropName
  LightGO_Switch: BooleanPropName
  LightGO_SetColor(go: GO, c: ColorVal): Ok
  LightGO_SetSwitch(go: GO, b: Bool): Ok
WHERE
  LightGO <: GO
  ColorVal = ColorPropVal + Color + Text
  BooleanVal = BooleanPropVal + Bool
```

A `LightGO` is a graphical object representing a light source. Like all graphical objects, light sources are affected by transformation properties, that is, attaching a property named `GO_Transform` to a light source object may affect the location and orientation of the light source.

In addition, there are two other properties specific to light sources: The property named `LightGO_Color` defines the color of the light emitted by the light source, and the property `LightGO_Switch` determines whether the light source is on or off. `LightGO_Color` associates with color property values, and `LightGO_Switch` associates with boolean property values.

`LightGO_SetColor` and `LightGO_SetSwitch` are overloaded convenience procedures for attaching the `LightGO_Color` and `LightGO_Switch` properties to graphical objects; they are similar to `GO_SetTransform` (see Section 3.8).

`LightGO` is an "abstract class", therefore the module does not contain any function for creating light sources.

3.12 The AmbientLightGO Module

```
- help AmbientLightGO;
  AmbientLightGO_New(c: ColorVal): AmbientLightGO
WHERE
  AmbientLightGO <: LightGO
  ColorVal = ColorPropVal + Color + Text
```

An `AmbientLightGO` is a graphical object that describes an ambient light source, that is, a light source that emits undirected, non-positional, non-attenuating light. Another way to describe ambient light is that its effect on surfaces does not depend on their location or orientation.

AmbientLightGO is a subtype of LightGO; therefore, it is affected by the LightGO_Color and LightGO_Switch properties. However, although it is (by transitivity) a subtype of GO, it is not affected by the GO_Transform property, as it has no spatial attributes.

The function call AmbientLightGO_New(c) takes a value c, which can be a ColorPropVal, a Color, or a Text identifying a color, and returns a new ambient light source, which emits light of the specified color. One can imagine AmbientLightGO_New to be defined as:

```
proc(c)
     let l = a new AmbientLightGO;
     LightGO_SetColor(l,c);
     LightGO_SetSwitch(l,true);
     l
end
```

3.13 The VectorLightGO Module

```
- help VectorLightGO;
  VectorLightGO_New(c: ColorVal, dir: PointVal): VectorLightGO
  VectorLightGO_Direction: PointPropName
  VectorLightGO_SetDirection(l: VectorLightGO, dir: PointVal): Ok
WHERE
  VectorLightGO <: LightGO
  PointVal = PointPropVal + Point3
  ColorVal = ColorPropVal + Color + Text
```

A VectorLightGO is a graphical object that describes a vector light source, that is, a light source that emits directed, non-positional, and non-attenuating light. One can think of vector light sources as being infinitely far away (and extremely bright), so that all rays that reach the scene are parallel to each other, and are of constant intensity. The closest real-life example of a vector light source is the sun (sun rays that reach the earth are very close to being parallel and non-attenuating; however, the sun has a well-defined position).

VectorLightGO is a subtype of LightGO, and as such, it is affected by the LightGO_Color and LightGO_Switch properties. It is also (by transitivity) a subtype of GO, and therefore is affected by the GO_Transform property (the rotation component of the transformation affects the direction of the light rays). In addition, there is a property named VectorLightGO_Direction which affects only vector light sources. This property describes the direction of the light rays; its value component is of type PointPropVal. A PointPropVal can be viewed as a function that, given a time, returns a Point3; in this context, the point is interpreted as a vector.

VectorLightGO_SetDirection is an overloaded convenience procedure for attaching

VectorLightGO_Direction properties to graphical objects; it is similar to GO_SetTransform (see Section 3.8).

VectorLightGO_New(c,d) is an overloaded function for creating new vector light sources. c can be a ColorPropVal, a Color, or a Text identifying a color; d can be a PointPropVal or a Point3. The new light object has LightGO_Color, LightGO_Switch, and VectorLightGO_Direction properties attached to it. One can imagine VectorLightGO_New to be defined as:

```
proc(c,d)
     let l = a new VectorLightGO;
     LightGO_SetColor(l,c);
     LightGO_SetSwitch(l,true);
     VectorLightGO_SetDirection(l,d);
     l
end
```

3.14 The PointLightGO Module

```
- help PointLightGO;
  PointLightGO_New(c: ColorVal, orig: PointVal,
                   att0 att1: RealVal): PointLightGO
  PointLightGO_Origin: PointPropName
  PointLightGO_SetOrigin(go: GO, orig: PointVal): Ok
  PointLightGO_Attenuation0: RealPropName
  PointLightGO_SetAttenuation0(go: GO, att: RealVal): Ok
  PointLightGO_Attenuation1: RealPropName
  PointLightGO_SetAttenuation1(go: GO, att: RealVal): Ok
WHERE
  PointLightGO <: LightGO
  PointVal = PointPropVal + Point3
  RealVal = RealPropVal + Real + Int
  ColorVal = ColorPropVal + Color + Text
```

A PointLightGO is a graphical object that describes a point-shaped light source, which is emitting directional light uniformly in all directions. The amount of light that strikes another graphical object depends on the distance between this object and the point light source. This phenomenon is called *attenuation*.

In a scene with positional light sources (vector lights or spot lights), there is an *attenuation factor* associated with every point in the scene. The attenuation factor is the fraction of light from the light sources that reaches this point.

Consider a scene with a single point light source. Given a point, let d be the distance between this point and the light source. In real life, the attenuation factor is proportional to $\frac{1}{d^2}$. Unfortunately, computer-generated scenes that use this lighting model tend to look unnatural, because real-life scenes are not only illuminated by light from a single source, but also by light reflected from other scene objects.

Therefore, PEX and OpenGL use different formulas for the attenuation factor. In PEX, it is defined to be $\frac{1}{c_0+c_1 d}$; in OpenGL, it is defined to be $\frac{1}{c_0+c_1 d+c_2 d^2}$. c_0 is called the *constant attenuation coefficient*, c_1 is called the *linear attenuation coefficient*, and c_2 is called the *quadratic attenuation coefficient*.

Obliq-3D started out as a PEX-based animation system; therefore it allows the client program to control only the constant and the linear attenuation coefficients. Control for the quadratic attenuation coefficient is a likely extension.

`PointLightGO` is a subtype of `LightGO`, and as such, it is affected by the `LightGO_Color` and `LightGO_Switch` properties. It is also (by transitivity) a subtype of `GO`, and therefore is affected by the `GO_Transform` property. In addition, the module provides three extra properties:

- The point property named `PointLightGO_Origin` controls the location of the point light source. By default, point lights are located at the origin.
- The real property named `PointLightGO_Attenuation0` controls the constant attenuation coefficient c_0. By default, c_0 is 1.
- The real property named `PointLightGO_Attenuation1` controls the linear attenuation coefficient c_1. By default, c_1 is 1.

`PointLightGO_SetOrigin`, `PointLightGO_SetAttenuation0`, and `PointLight-GO_SetAttenuation1` are overloaded convenience procedures for attaching the corresponding properties to graphical objects; they are similar to `GO_SetTransform` (see Section 3.8).

`PointLightGO_New(c,p,a0,a1)` is an overloaded function for creating new point light sources. `c` can be a `ColorPropVal`, a `Color`, or a `Text` identifying a color; `p` can be a `PointPropVal` or a `Point3`; `a0` and `a1` can be `RealPropVal`, `Real`, or `Int` values. The new light object has `LightGO_Color`, `LightGO_Switch`, `PointLightGO_Origin`, `PointLightGO_Attenuation0`, and `PointLightGO_Attenuation1` properties attached to it. One can imagine `PointLightGO_New` to be defined as:

```
proc(c,p,a0,a1)
     let l = a new PointLightGO;
     LightGO_SetColor(l,c);
     LightGO_SetSwitch(l,true);
     PointLightGO_SetOrigin(l,p);
     PointLightGO_SetAttenuation0(l,a0);
```

```
        PointLightGO_SetAttenuation1(l,a1);
        l
    end
```

3.15 The SpotLightGO Module

```
- help SpotLightGO;
  SpotLightGO_New(c: ColorVal, orig dir: PointVal,
                  conc spread att0 att1: Real): SpotLightGO
  SpotLightGO_Origin: PointPropName
  SpotLightGO_SetOrigin(go: GO, orig: PointVal): Ok
  SpotLightGO_Direction: PointPropName
  SpotLightGO_SetDirection(go: GO, dir: PointVal): Ok
  SpotLightGO_Concentration: RealPropName
  SpotLightGO_SetConcentration(go: GO, conc: RealVal): Ok
  SpotLightGO_SpreadAngle: RealPropName
  SpotLightGO_SetSpreadAngle(go: GO, spread: RealVal): Ok
  SpotLightGO_Attenuation0: RealPropName
  SpotLightGO_SetAttenuation0(go: GO, att: RealVal): Ok
  SpotLightGO_Attenuation1: RealPropName
  SpotLightGO_SetAttenuation1(go: GO, att: RealVal): Ok
WHERE
  SpotLightGO <: LightGO
  PointVal = PointPropVal + Point3
  RealVal = RealPropVal + Real + Int
  ColorVal = ColorPropVal + Color + Text
```

A `SpotLightGO` is a graphical object that describes a positional light source that is emitting a cone of directed, colored light. Light from a spot light source attenuates with increasing distance. Attenuation follows the same formula and is controlled by the same parameters as attenuation of light emitted by point light sources (see Section 3.14).

The geometry of the cone of emitted light is determined by the *position* p of the light source, the *direction* d of the light, and the *spread angle* α of the cone. Figure 21 illustrates the relationship.

The *concentration exponent* γ determines if and by how much the intensity of a ray decreases, the further it is away from the central axis of the cone. Given a ray whose angle to the main axis is β (where $0 < \beta < \alpha$), the intensity of this ray is $\cos^{\gamma}(\beta)$ times the intensity of the light along the central axis. Choosing γ to be 0 means that the intensity of the light is constant throughout the cone (modulo attenuation), choosing a higher γ means that light rays are less intense the more they are off-center.

`SpotLightGO` is a subtype of `LightGO`, and as such, it is affected by the `LightGO_Color`

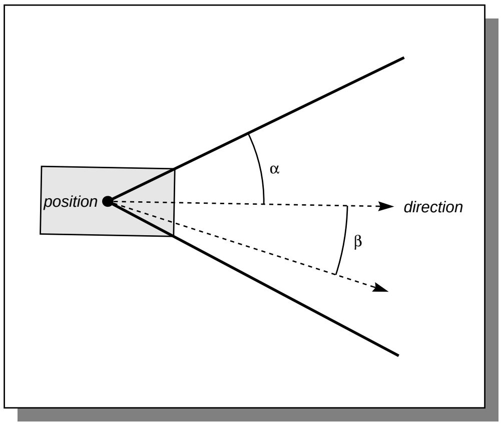


Figure 21: Geometry of the light cone emitted by a spot light source

and `LightGO_Switch` properties. It is also (by transitivity) a subtype of `GO`, and therefore is affected by the `GO_Transform` property. In addition, the module provides several extra properties:

- The point property named `SpotLightGO_Origin` controls the location of the spot light source. By default, spot lights are located at the origin.
- The point property named `SpotLightGO_Direction` controls the direction of the spot light source. By default, spot lights point along the vector $(1, 1, 1)$.
- The real property named `SpotLightGO_Concentration` controls the concentration exponent γ of the spot light source. By default, γ is 1.
- The real property named `SpotLightGO_SpreadAngle` α controls the spread angle of the spot light source. By default, α is 1.
- The real property named `SpotLightGO_Attenuation0` controls the constant attenuation coefficient c_0. By default, c_0 is 1.
- The real property named `SpotLightGO_Attenuation1` controls the linear attenuation coefficient c_1. By default, c_1 is 1.

`SpotLightGO_SetOrigin`, `SpotLightGO_SetDirection`, `SpotLightGO_SetConcentration`, `SpotLightGO_SetSpreadAngle`, `SpotLightGO_SetAttenuation0`, and `SpotLightGO_SetAttenuation1` are overloaded convenience procedures for attaching the corresponding properties to graphical objects; they are similar to `GO_SetTransform` (see Section 3.8).

`SpotLightGO_New(col,pos,dir,conc,sprd,a0,a1)` is an overloaded function for creating new spot light sources. `col` can be a `ColorPropVal`, a `Color`, or a `Text` identifying a color; `pos` and `dir` can be `PointPropVal` or `Point3` values; `conc`, `sprd`, `a0` and `a1` can be `RealPropVal`, `Real`, or `Int` values. The new light object has `LightGO_Color`, `LightGO_Switch`, `SpotLightGO_Origin`, `SpotLightGO_Direction`, `SpotLightGO_Concentration`, `SpotLightGO_SpreadAngle`, `SpotLightGO_Attenuation0`, and `SpotLightGO_Attenuation1` properties attached to it. One can imagine `SpotLightGO_New` to be defined as:

```
proc(col,pos,dir,conc,sprd,a0,a1)
     let l = a new SpotLightGO;
     LightGO_SetColor(l,col);
     LightGO_SetSwitch(l,true);
     SpotLightGO_SetOrigin(l,pos);
     SpotLightGO_SetDirection(l,dir);
     SpotLightGO_SetConcentration(l,conc);
     SpotLightGO_SetSpreadAngle(l,sprd);
     SpotLightGO_SetAttenuation0(l,a0);
     SpotLightGO_SetAttenuation1(l,a1);
     l
end
```

3.16 The CameraGO Module

```
- help CameraGO;
  CameraGO_From: PointPropName
  CameraGO_To: PointPropName
  CameraGO_Up: PointPropName
  CameraGO_Aspect: PointPropName
  CameraGO_SetFrom(go: GO, PointVal): Ok
  CameraGO_SetTo(go: GO, PointVal): Ok
  CameraGO_SetUp(go: GO, PointVal): Ok
  CameraGO_SetAspect(go: GO, RealVal): Ok
TYPE
  CameraGO <: GO
  PointVal = PointPropVal + Point3
  RealVal = RealPropVal + Real + Int
```

A `CameraGO` is a graphical object that describes a *camera*. Camera objects are used for viewing the scene. Every root object has a window and a camera associated with it (see Section 3.10), and the picture that is shown in the window is the part of the scene graph below the root, as seen through the camera.

A scene can contain many cameras, but each root object uses only one camera at any given time; however, the client program can change what camera is being used (see Section 3.10).

A camera behaves like any other graphical object. In particular, it may be part of a group, and it is affected by the `GO_Transform` property. So, it is for instance possible to build a group that describes a car, place a camera inside the car, then move the car by changing a transformation property attached to it, and view the scene from inside the moving car. Program 18 shows a similar (but simpler) program that moves a camera.

Cameras depend on several parameters: Their *position*, their *target*, their *up-vector*, and their *aspect ratio*. The *position* is the point where the camera is located. The *target* is a point in the center of the camera's field of vision; in other words, it determines the viewing direction. The *up-vector* determines what direction "up" is in the projected image. A line parallel to this vector appears vertically in the viewing window. Finally, the *aspect ratio* controls width/height distortion. Our use of the term "aspect ratio" differs from the standard usage. Normally, aspect ratio refers to the ratio between width and height (for instance, NTSC TV has a 4-to-3 aspect ratio). However, we define the aspect ratio not to be the absolute width/height ratio, but rather the width/height distortion. An aspect ratio of 1, leaves the picture undistorted, an aspect ratio of 2 flattens objects in the scene[8].

`CameraGO` is a subclass of `GO`. Camera objects do not provide any additional methods, however, they are affected by four properties in addition to `GO_Transform`.

- The point property named `CameraGO_From` controls the position of the camera. By default, the camera is located at point $(0, 0, 100)$.
- The point property named `CameraGO_To` controls the target point of the camera. By default, the camera is looking at the origin.
- The point property named `CameraGO_Up` controls the up-vector of the camera. By default, it is $(0, 1, 0)$.
- The real property named `CameraGO_Aspect` controls the aspect ratio of the camera. By default, it is 1.

`CameraGO_SetFrom`, `CameraGO_SetTo`, `CameraGO_SetUp`, and `CameraGO_SetAspect` are overloaded convenience procedures for attaching the corresponding properties to graphical objects; they are similar to `GO_SetTransform` (see Section 3.8).

`CameraGO` is an abstract class, hence there is no function for creating new camera objects.

[8] In the current system, the active area of the rendering window is always square.

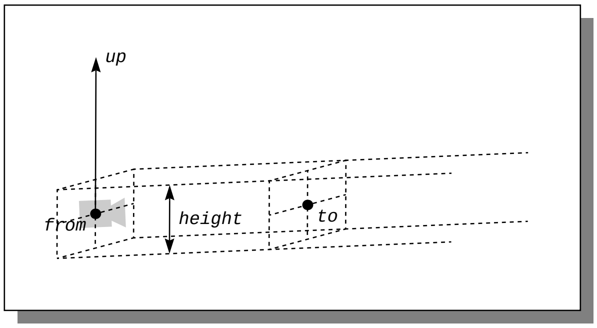


Figure 22: Parameters of an Orthographic Camera

3.17 The OrthoCameraGO Module

```
- help OrthoCameraGO;
  OrthoCameraGO_New(from to up: PointVal, height: RealVal): OrthoCameraGO
  OrthoCameraGO_Height: RealPropName
  OrthoCameraGO_SetHeight(go: GO, height: RealVal): Ok
WHERE
  OrthoCameraGO <: CameraGO
  PointVal = PointPropVal + Point3
  RealVal = RealPropVal + Real + Int
```

An `OrthoCameraGO` is a graphical object describing an *orthographic camera*. An orthographic camera uses an *orthographic projection* (or *parallel projection*) to map the scene onto the viewing area. Lines that are parallel in the scene appear parallel in the image. The volume of space observed by such a camera forms a parallelopiped. The geometry of the parallelopiped is determined by the camera's position ("from"), its viewing direction (specified through the target point "to"), its up-vector ("up"), the parallelopiped's height ("height"), and its width (determined by the height of the parallelopiped, and the aspect ratio of the camera). Figure 22 illustrates the role of these parameters.

`OrthoCameraGO` is a subclass of `CameraGO`. Orthographic camera objects do not provide any additional methods. They observe all the properties defined for their supertypes `GO` and `CameraGO`. In addition, they provide a real property named `OrthoCameraGO_Height`, which controls the height of the volume of space observed by the camera. By default, orthographic cameras are 10 units high.

`OrthoCameraGO_SetHeight` is an overloaded convenience procedure for attaching the property `OrthoCameraGO_Height` to graphical objects; it is similar to `GO_SetTransform` (see Section 3.8).

`OrthoCameraGO_New(fr,to,up,ht)` is an overloaded function for creating new orthographic cameras. `fr`, `to`, and `up` can be `PointPropVal`'s or `Point3`'s; `ht` can be a `RealPropVal`, a `Real`, or an `Int`. The new camera object has `CameraGO_From`, `CameraGO_To`, `CameraGO_Up`, and `OrthoCameraGO_Height` properties attached to it. One can imagine `OrthoCameraGO_New` to be defined as:

```
proc(fr,to,up,ht)
     let c = a new OrthoCameraGO;
     CameraGO_SetFrom(c,fr);
     CameraGO_SetTo(c,to);
     CameraGO_SetUp(c,up);
     OrthoCameraGO_SetHeight(c,ht);
     c
end
```

3.18 The PerspCameraGO Module

```
- help PerspCameraGO;
  PerspCameraGO_New(from to up: PointVal, fovy: RealVal): PerspCameraGO
  PerspCameraGO_Fovy: RealPropName
  PerspCameraGO_SetFovy(go: GO, fovy: RealVal): Ok
WHERE
  PerspCameraGO <: CameraGO
  PointVal = PointPropVal + Point3
  RealVal = RealPropVal + Real + Int
```

A `PerspCameraGO` is a graphical object describing a *perspective camera*. A perspective camera behaves like a real-world camera; it uses a *perspective projection* to map the scene onto the viewing area. The volume of space observed by such a camera forms an (infinitely high) pyramid. The geometry of the pyramid is determined by the camera's position ("from"), its viewing direction (specified through the target point "to"), its up-vector ("up"), the *field-of-vision* ("fovy"), which is the angle between the "up" and the "down" wall of the pyramid, and the width of the pyramid, which is determined by its height and the aspect ratio of the camera. Figure 23 illustrates the role of these parameters.

`PerspCameraGO` is a subclass of `CameraGO`. Perspective camera objects do not provide any additional methods. They observe all the properties defined for their supertypes `GO` and `CameraGO`. In addition, they provide a real property named `PerspCameraGO_Fovy`, which controls the field of vision of the pyramidal area of space observed by the camera. By default, perspective cameras have a field of vision of 0.1 degrees radian.

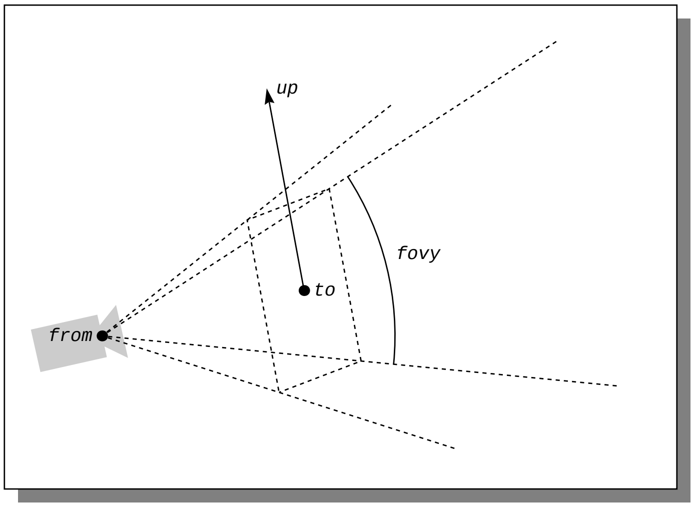


Figure 23: Parameters of a Perspective Camera

`PerspCameraGO_SetFovy` is an overloaded convenience procedure for attaching the property `PerspCameraGO_Fovy` to graphical objects; it is similar to `GO_SetTransform` (see Section 3.8).

`PerspCameraGO_New(fr,to,up,fv)` is an overloaded function for creating new orthographic cameras. `fr`, `to`, and `up` can be `PointPropVal`'s or `Point3`'s; `fv` can be a `RealPropVal`, a `Real`, or an `Int`. The new camera object has `CameraGO_From`, `CameraGO_To`, `CameraGO_Up`, and `PerspCameraGO_Fovy` properties attached to it. One can imagine `PerspCameraGO_New` to be defined as:

```
proc(fr,to,up,fv)
     let c = a new PerspCameraGO;
     CameraGO_SetFrom(c,fr);
     CameraGO_SetTo(c,to);
     CameraGO_SetUp(c,up);
     PerspCameraGO_SetFovy(c,fv);
     c
end
```

3.19 The LineGO Module

```
- help LineGO;
  LineGO_New(p1 p2: PointVal): LineGO
  LineGO_Color: ColorPropName
  LineGO_Width: RealPropName
  LineGO_Type: LineTypePropName
  LineGO_Point1: PointPropName
  LineGO_Point2: PointPropName
  LineGO_SetColor(o: GO, c: ColorVal): Ok
  LineGO_SetWidth(o: GO, r: RealVal): Ok
  LineGO_SetType(o: GO, t: LineType): Ok
  LineGO_SetPoint1(o: GO, p: PointVal): Ok
  LineGO_SetPoint2(o: GO, p: PointVal): Ok
WHERE
  LineGO <: GO
  PointVal = PointPropVal + Point3
  RealVal = RealPropVal + Real + Int
  ColorVal = ColorPropVal + Color + Text
  LineTypeVal = LineTypePropVal + LineType
```

A `LineGO` object is a graphical object describing a line. It is a subtype of `GO`, and does not add any extra fields or methods.

A line is a graphical object, and as such, it is affected by the `GO_Transform` property. The two endpoints of a line are controlled by two point properties named `LineGO_Point1` and `LineGO_Point2`. If these two properties are undefined, the line goes from point $(0,0,0)$ to point $(1,0,0)$ with respect to the local coordinate system. The line color is controlled by a color property named `LineGO_Color`; by default, lines are white. The line width is controlled by a real property named `LineGO_Width`; by default, lines are one pixel wide. Finally, the type of the line (that is, whether it is solid, dashed, dotted, or dashed and dotted) is controlled by a line type property named `LineGO_Type`; by default, lines are solid.

`LineGO_SetPoint1`, `LineGO_SetPoint2`, `LineGO_SetColor`, `LineGO_SetWidth`, and `LineGO_SetType` are overloaded convenience procedures for attaching the line-specific properties to graphical objects; they are similar to `GO_SetTransform` (see Section 3.8).

`LineGO_New(p1,p2)` creates a new line object and returns it. In the process, it attaches `p1` as property `LineGO_Point1` and `p2` as property `LineGO_Point2` to the new line object, thereby defining its two endpoints.

3.20 The MarkerGO Module

```
- help MarkerGO;
  MarkerGO_New(point: PointVal): MarkerGO
  MarkerGO_Center: PointPropName
  MarkerGO_Color: ColorPropName
  MarkerGO_Scale: RealPropName
  MarkerGO_Type: MarkerTypePropName
  MarkerGO_SetCenter(o: GO, p: PointVal): Ok
  MarkerGO_SetColor(o: GO, c: ColorVal): Ok
  MarkerGO_SetScale(o: GO, r: RealVal): Ok
  MarkerGO_SetType(o: GO, t: MarkerTypeVal): Ok
WHERE
  MarkerGO <: GO
  PointVal = PointPropVal + Point3
  RealVal = RealPropVal + Real + Int
  ColorVal = ColorPropVal + Color + Text
  MarkerTypeVal = MarkerTypePropVal + MarkerType
```

A `MarkerGO` object is a graphical object describing a marker, that is, a graphical representation of a single point. It is a subtype of `GO`, and does not add any extra fields or methods.

A marker is a graphical object, and as such, it is affected by the `GO_Transform` property. The location of a marker is controlled by a point property named `MarkerGO_Point`; if this property is undefined, the marker is located at the origin of the local coordinate system. The marker color is controlled by a color property named `MarkerGO_Color`; by default, markers are white. The scale of the marker is controlled by a real property named `MarkerGO_Scale`; the default scale factor is 1. Finally, the type of the marker (that is, whether it looks like a dot, a circle, a cross, an asterisk, or an "X") is controlled by a marker type property named `MarkerGO_Type`; by default, markers are shown as asterisks.

`MarkerGO_SetPoint`, `MarkerGO_SetColor`, `MarkerGO_SetScale`, and `MarkerGO_SetType` are overloaded convenience procedures for attaching the marker-specific properties to graphical objects; they are similar to `GO_SetTransform` (see Section 3.8).

`MarkerGO_New(p)` creates a new marker object and returns it. In the process, it attaches `p` as property `MarkerGO_Point` to the new marker object, thereby defining its center.

3.21 The SurfaceGO Module

```
- help SurfaceGO;
  SurfaceGO_Color: ColorPropName
  SurfaceGO_SetColor(o: GO, color: ColorVal): Ok
  SurfaceGO_RasterMode: RasterModePropName
  SurfaceGO_SetRasterMode(o: GO, t: RasterModeVal): Ok
  SurfaceGO_AmbientReflectionCoeff: RealPropName
  SurfaceGO_SetAmbientReflectionCoeff(o: GO, r: RealVal): Ok
  SurfaceGO_DiffuseReflectionCoeff: RealPropName
  SurfaceGO_SetDiffuseReflectionCoeff(o: GO, r: RealVal): Ok
  SurfaceGO_SpecularReflectionCoeff: RealPropName
  SurfaceGO_SetSpecularReflectionCoeff(o: GO, r: RealVal): Ok
  SurfaceGO_SpecularReflectionConc: RealPropName
  SurfaceGO_SetSpecularReflectionConc(o: GO, r: RealVal): Ok
  SurfaceGO_TransmissionCoeff: RealPropName
  SurfaceGO_SetTransmissionCoeff(o: GO, r: RealVal): Ok
  SurfaceGO_SpecularReflectionColor: ColorPropName
  SurfaceGO_SetSpecularReflectionColor(o: GO, color: ColorVal): Ok
  SurfaceGO_Lighting: BooleanPropName
  SurfaceGO_SetLighting(o: GO, t: BooleanVal): Ok
  SurfaceGO_Shading: ShadingPropName
  SurfaceGO_SetShading(o: GO, sh: ShadingVal): Ok
  SurfaceGO_EdgeVisibility: BooleanPropName
  SurfaceGO_SetEdgeVisibility(o: GO, b: BoolVal): Ok
  SurfaceGO_EdgeColor: ColorPropName
  SurfaceGO_SetEdgeColor(o: GO, color: ColorVal): Ok
  SurfaceGO_EdgeType: LineTypePropName
  SurfaceGO_SetEdgeType(o: GO, lt: LineTypeVal): Ok
  SurfaceGO_EdgeWidth: RealPropName
  SurfaceGO_SetEdgeWidth(o: GO, r: RealVal): Ok
TYPE
  SurfaceGO <: GO
  ColorVal = ColorPropVal + Color + Text
  BooleanVal = BooleanPropVal + Bool
  RealVal = RealPropVal + Real + Int
  LineTypeVal = LineTypePropVal + LineType
  RasterModeVal = RasterModePropVal + RasterMode
  ShadingVal = ShadingPropVal + Shading
```

A *surface* is a graphical object that is made out of polygons. One important difference between surfaces and non-surfaces (such as lines and markers) is that surfaces are affected by light sources, whereas non-surfaces are not.

`SurfaceGO`, the abstract type of surface objects, is a subtype of `GO`. It does not add any extra fields or methods; however, it defines a number of extra property names that affect the appearance of surfaces.

- The color property named `SurfaceGO_Color` controls the color of the surface.
- The raster mode property named `SurfaceGO_RasterMode` controls the rasterization mode used for drawing the surface, that is, whether the interior or only the boundary is drawn, or whether nothing is drawn at all.
- The boolean property named `SurfaceGO_Lighting` controls whether any lighting computations are performed. If the property is false, no lighting computations are performed. In this case, the surface is drawn in its natural color, unaffected by light sources (but still affected by depth cueing).
- The shading property named `SurfaceGO_Shading` controls which surface shading method is used. We support two shading methods: Flat Shading and Gouraud Shading. The *Flat Shading method* computes an intensity value for a single point of the surface, and uses this intensity value for the entire surface. The *Gouraud shading method* computes intensity values for each vertex of the polygons that make up the surface, and then interpolates between these values to determine intensity values for the other points on the surface.
- The real property named `SurfaceGO_TransmissionCoeff` controls the transparency of a surface. A value of 0 means that the surface is completely opaque, while a value of 1 means that is it completely transparent.
- The real property named `SurfaceGO_AmbientReflectionCoeff` controls the amount of ambient light reflected by the surface, as a fraction of the ambient light falling onto the surface. A value of 0 indicates that no ambient light is reflected, while a value of 1 indicates that all ambient light striking the surface is reflected.
- The real property named `SurfaceGO_DiffuseReflectionCoeff` controls the amount of non-ambient light reflected diffusely (in all directions), as a fraction of all the directional light striking the surface. A value of 0 indicates that none of the directional light is reflected diffusely; a value of 1 indicates that all of it is reflected.
- The real property named `SurfaceGO_SpecularReflectionCoeff` controls the brightness of specular highlights. A value of 0 turns off specular reflection entirely; a value of 1 makes the highlights as pronounced as possible.
- The real property named `SurfaceGO_SpecularReflectionConc` controls how focused specular highlights are. A value of 1 creates a very blurry highlight; higher values create increasingly more sharply focused highlights.

- The color property named `SurfaceGO_SpecularReflectionColor` controls the color shift of specularly reflected light.
- The boolean property named `SurfaceGO_EdgeVisibility` controls whether the boundaries of the polygons that make up the surface are visible or not.
- The color property named `SurfaceGO_EdgeColor` controls the color of the boundary lines.
- The color property named `SurfaceGO_EdgeWidth` controls the width of the boundary lines.
- The line type property named `SurfaceGO_EdgeType` controls how the boundary lines are drawn (sold, dashed, dotted, or dashed and dotted).

For each property name defined in the `SurfaceGO` module, there is an overloaded convenience procedure for attaching the property to graphical objects; these procedures are all similar to `GO_SetTransform` (see Section 3.8).

3.22 The PolygonGO Module

```
- help PolygonGO;
  PolygonGO_New(pts: [PointVal]): PolygonGO
  PolygonGO_NewWithShapeHint(pts: [PointVal], s: Shape): PolygonGO
WHERE
  PolygonGO <: SurfaceGO
  PointVal = PointPropVal + Point3
  Shape = Text  (one of "Unknown", "Convex", "NonConvex", "Complex")
```

A `PolygonGO` object is a graphical object describing a polygon. `PolygonGO` is a subtype of `SurfaceGO`, and does not add any extra fields, methods, or properties. A polygon is affected by all the properties defined in the GO and `SurfaceGO` modules.

`PolygonGO_New(pts)` creates a new polygon object and returns it. `pts` is an array of points. Each of these points can be a constant `Point3` value or a time-variant `PointPropVal` value.

`PolygonGO_NewWithShapeHint(pts,s)` is similar to `PolygonGO_New(pts)`, but allows the client program to supply a "shape hint" `s`, indicating the shape of the polygon . Possible shape hints are `"Convex"` (the polygon is convex), `"NonConvex"` (the polygon is concave, but not self-intersecting), `"Complex"` (the edges of the polygon are self-intersecting), and

"Unknown" (the shape is unknown). Supplying a shape hint can increase rendering performance. On the other hand, shape hints must be accurate. In particular, if the shape of a polygon changes (that is, if the vertices defining the polygon change over time), the shape hint must be conservative enough to be accurate for the entire lifetime of the polygon. Polygon_New assumes the shape of the polygon to be unknown; in other words, it makes the safest, most conservative choice.

3.23 The BoxGO Module

```
- help BoxGO;
  BoxGO_New(p1 p2: PointVal): BoxGO
  BoxGO_Corner1: PointPropName
  BoxGO_Corner2: PointPropName
  BoxGO_SetCorner1(o: GO, p: PointVal): Ok
  BoxGO_SetCorner2(o: GO, p: PointVal): Ok
WHERE
  BoxGO <: SurfaceGO
  PointVal = PointPropVal + Point3
```

A BoxGO object is a graphical object describing a box, or more precisely, an axis-aligned parallelopiped. The geometry of such an object can be specified in terms of two opposing corner points.

BoxGO is a subtype of SurfaceGO, and does not add any extra fields or methods. A box is affected by all the properties defined in the GO and SurfaceGO modules. In addition, the two opposing corners of the box are controlled by the point properties named BoxGO_Corner1 and BoxGO_Corner2.

BoxGO_SetCorner1 and BoxGO_SetCorner2 are overloaded convenience procedures for attaching the box-specific properties to graphical objects; they are similar to GO_SetTransform (see Section 3.8).

BoxGO_New(p1,p2) creates a new box object and returns it. In the process, it attaches p1 as property BoxGO_Corner1 and p2 as property BoxGO_Corner2 to the new box object, thereby defining its two corners.

3.24 The DiskGO Module

```
- help DiskGO;
  DiskGO_New(center normal: PointVal, rad: RealVal): DiskGO
  DiskGO_NewWithPrec(center normal: PointVal, rad: RealVal, prec: Int): DiskGO
  DiskGO_Center: PointPropName
  DiskGO_Normal: PointPropName
  DiskGO_Radius: RealPropName
  DiskGO_SetCenter(o: GO, p: PointVal): Ok
  DiskGO_SetNormal(o: GO, p: PointVal3): Ok
  DiskGO_SetRadius(o: GO, r: RealVal): Ok
WHERE
  DiskGO <: SurfaceGO
  PointVal = PointPropVal + Point3
  RealVal = RealPropVal + Real + Int
```

A `DiskGO` object is a graphical object describing a disk. It is a subtype of `SurfaceGO`, and does not add any extra fields or methods.

A disk is affected by all the properties defined in the GO and `SurfaceGO` modules as well as by three additional, disk-specific properties: The center of a disk is controlled by a point property named `DiskGO_Center`, its normal vector is affected by a point property named `DiskGO_Normal`, and its radius is controlled by a real property named `DiskGO_Radius`.

`DiskGO_SetCenter`, `DiskGO_SetNormal`, and `DiskGO_SetRadius` are overloaded convenience procedures for attaching the disk-specific properties to graphical objects; they are similar to `GO_SetTransform` (see Section 3.8).

`DiskGO_New(p,n,r)` creates a new disk object and returns it. In the process, it attaches `p` as property `DiskGO_Center`, `n` as property `DiskGO_Normal`, and `r` as property `DiskGO_Radius` to the new disk object, thereby defining its center, normal vector, and radius. Disks are approximated by polygons; by default, this polygon has 10 sizes.

`DiskGO_NewWithPrec(p,n,r,i)` behaves like `DiskGO_New(p,n,r)`, but allows the client program to specify the number of sides of the polygon. The new disk is approximated by a polygon with `i` sides.

3.25 The SphereGO Module

```
- help SphereGO;
  SphereGO_New(p: PointVal, rad: RealVal): SphereGO
  SphereGO_NewWithPrec(p: PointVal, rad: RealVal, prec: Int): SphereGO
  SphereGO_Center: PointPropName
  SphereGO_Radius: RealPropName
  SphereGO_SetCenter(go: GO, center: PointVal): Ok
  SphereGO_SetRadius(go: GO, radius: RealVal): Ok
WHERE
  SphereGO <: SurfaceGO
  PointVal = PointPropVal + Point3
  RealVal = RealPropVal + Real + Int
```

A `SphereGO` object is a graphical object describing a sphere. It is a subtype of `SurfaceGO`, and does not add any extra fields or methods.

A sphere is affected by all the properties defined in the GO and `SurfaceGO` modules as well as by two additional, sphere-specific properties: The center of a sphere is controlled by a point property named `SphereGO_Center`; its radius is controlled by a real property named `SphereGO_Radius`.

`SphereGO_SetCenter` and `SphereGO_SetRadius` are overloaded convenience procedures for attaching the sphere-specific properties to graphical objects; they are similar to `GO_SetTransform` (see Section 3.8).

`SphereGO_New(p,r)` creates a new sphere object and returns it. In the process, it attaches `p` as property `SphereGO_Center` and `r` as property `SphereGO_Radius` to the new sphere object, thereby defining its center and radius. Spheres are approximated by polyhedra; by default, a sphere consists of 30 strips, each one containing 30 triangles.

`SphereGO_NewWithPrec(p,r,i)` behaves like `SphereGO_New(p,r)`, but allows the client program to specify the precision of the polygonal approximation to a sphere. The new sphere is composed of $\mathtt{i}^2$ triangles.

3.26 The CylinderGO Module

```
- help CylinderGO;
  CylinderGO_New(p1 p2: PointVal, rad: RealVal): CylinderGO
  CylinderGO_NewWithPrec(p1 p2: PointVal, rad: RealVal, prec: Int): CylinderGO
  CylinderGO_Point1: PointPropName
  CylinderGO_Point2: PointPropName
  CylinderGO_Radius: RealPropName
  CylinderGO_SetPoint1(o: GO, p: PointVal): Ok
  CylinderGO_SetPoint2(o: GO, p: PointVal): Ok
  CylinderGO_SetRadius(o: GO, r: RealVal): Ok
WHERE
  CylinderGO <: SurfaceGO
  PointVal = PointPropVal + Point3
  RealVal = RealPropVal + Real + Int
```

A `CylinderGO` object is a graphical object describing a cylinder. It is a subtype of `SurfaceGO`, and does not add any extra fields or methods.

A cylinder is affected by all the properties defined in the GO and `SurfaceGO` modules as well as by three additional, cylinder-specific properties: The two endpoints of a cylinder are controlled by two point properties named `CylinderGO_Point1` and `CylinderGO_Point2`; its radius is controlled by a real property named `CylinderGO_Radius`.

`CylinderGO_SetPoint1`, `CylinderGO_SetPoint2`, and `CylinderGO_SetRadius` are overloaded convenience procedures for attaching the cylinder-specific properties to graphical objects; they are similar to `GO_SetTransform` (see Section 3.8).

`CylinderGO_New(p1,p2,r)` creates a new cylinder object and returns it. In the process, it attaches `p1` as property `CylinderGO_Point1`, `p2` as property `CylinderGO_Point2`, and `r` as property `CylinderGO_Radius` to the new cylinder object, thereby defining its endpoints and radius. Cylinders are approximated by polyhedra; by default, a cylinder consists of 30 rectangles.

`CylinderGO_NewWithPrec(p1,p2,r,i)` behaves like `CylinderGO_New(p1,p2,r)`, but allows the client program to specify the precision of the cylinder. The new cylinder is composed of `i` rectangles.

3.27 The ConeGO Module

```
- help ConeGO;
  ConeGO_New(base tip: PointVal, rad: RealVal): ConeGO
  ConeGO_NewWithPrec(base tip: PointVal, rad: RealVal, prec: Int): ConeGO
  ConeGO_Base: PointPropName
  ConeGO_Tip: PointPropName
  ConeGO_Radius: RealPropName
  ConeGO_SetBase(o: GO, p: PointVal): Ok
  ConeGO_SetTip(o: GO, p: PointVal): Ok
  ConeGO_SetRadius(o: GO, r: RealVal): Ok
WHERE
  ConeGO <: SurfaceGO
  PointVal = PointPropVal + Point3
  RealVal = RealPropVal + Real + Int
```

A `ConeGO` object is a graphical object describing a cone. It is a subtype of `SurfaceGO`, and does not add any extra fields or methods.

A cone is affected by all the properties defined in the GO and `SurfaceGO` modules as well as by three additional, cone-specific properties: The base of a cone is controlled by a point property named `ConeGO_Base`, its tip is controlled by a point property named `ConeGO_Tip`, and its radius at the base is controlled by a real property named `ConeGO_Radius`.

`ConeGO_SetBase`, `ConeGO_SetTip`, and `ConeGO_SetRadius` are overloaded convenience procedures for attaching the cone-specific properties to graphical objects; they are similar to `GO_SetTransform` (see Section 3.8).

`ConeGO_New(p1,p2,r)` creates a new cone object and returns it. In the process, it attaches `p1` as property `ConeGO_Base`, `p2` as property `ConeGO_Tip`, and `r` as property `ConeGO_Radius` to the new cone object, thereby defining its endpoints and radius. Cones are approximated by polyhedra; by default, a cone consists of 30 triangles.

`ConeGO_NewWithPrec(p1,p2,r,i)` behaves like `ConeGO_New(p1,p2,r)`, but allows the client program to specify the precision of the cone. The newc one is composed of `i` triangles.

3.28 The TorusGO Module

```
- help TorusGO;
  TorusGO_New(center normal: PointVal, rad1 rad2: RealVal): TorusGO
  TorusGO_NewWithPrec(c n: PointVal, r1 r2: RealVal, prec: Int): TorusGO
  TorusGO_Center: PointPropName
  TorusGO_Normal: PointPropName
  TorusGO_Radius1: RealPropName
  TorusGO_Radius2: RealPropName
WHERE
  TorusGO <: SurfaceGO
  PointVal = PointPropVal + Point3
  RealVal = RealPropVal + Real + Int
```

A `TorusGO` object is a graphical object describing a torus. It is a subtype of `SurfaceGO`, and does not add any extra fields or methods.

A *torus* is an object that is shaped like a doughnut. It can be described as a surface of revolution: if we rotate a circle C around an axis A that does not intersect C, the resulting surface forms a torus. The radius of C is called the *minor radius* of the torus, the perpendicular distance from the center of C to A is called the *major radius* of the torus, the intersection between A and the perpendicular vector from C to A is called the *center* of the torus, and the directional vector of A is called the *normal* of the torus.

Alternatively, we can describe a torus as a cylinder which is bent such that its ends connect. The axis of the cylinder, which was straight before, now forms a circle. The center of this circle is called the *center* of the torus, the normal vector of the circle (in 3D) is called the *normal* of the torus, the radius of the sphere is called the *major radius* of the torus, and the radius of the cylinder is called the *minor radius* of the torus.

A torus is affected by all the properties defined in the GO and `SurfaceGO` modules as well as by three additional, torus-specific properties: the center of the torus is controlled by a point property named `TorusGO_Center`, its normal vector is controlled by a point property named `TorusGO_Normal`, its major radius is controlled by a real property named `TorusGO_Radius1`, and its minor radius is controlled by a real property named `TorusGO_Radius2`.

`TorusGO_SetCenter`, `TorusGO_SetNormal`, `TorusGO_SetRadius1`, and `TorusGO_SetRadius2` are overloaded convenience procedures for attaching the torus-specific properties to graphical objects; they are similar to `GO_SetTransform` (see Section 3.8).

`TorusGO_New(c,n,r1,r2)` creates a new torus object and returns it. In the process, it attaches `c` as property `TorusGO_Center`, `n` as property `TorusGO_Normal`, `r1` as property `TorusGO_Radius1`, and `r2` as property `TorusGO_Radius2` to the new torus object, thereby defining its geometry. Tori are approximated by polyhedra; by default, a torus consists of 30 strips, each one with 30 rectangles.

`TorusGO_NewWithPrec(c,n,r1,r2,i)` behaves like `TorusGO_New(c,n,r1,r2)`, but allows the client program to specify the precision of the torus. The new torus is composed of `i` strips, each one with `i` rectangles.

3.29 The QuadMeshGO Module

```
- help QuadMeshGO;
  QuadMeshGO_BadSize: Exception
  QuadMeshGO_ColorsUndefined: Exception
  QuadMeshGO_New(pts: [[Point3]]): QuadMeshGO
  QuadMeshGO_NewWithShapeHint(pts: [[Point3]], s: Shape): QuadMeshGO
WHERE
  QuadMeshGO <: SurfaceGO &
     { addFacetColors: ([[Col]]) => Ok ! QuadMeshGO_BadSize,
       setColorOfFacet: (i j: Int, c: Col) => Ok ! QuadMeshGO_ColorsUndefined }
  Shape = Text  (one of "Unknown", "Convex", "NonConvex", "Complex")
  Col = Color + Text
```

A `QuadMeshGO` is a graphical object that describes a *quadrilateral mesh*, a grid of four-sided polygons (also called *quadrilaterals*). A quadrilateral mesh is defined by a two-dimensional array of points

$$\begin{pmatrix} p_{0,0} & \cdots & p_{0,n} \\ \vdots & & \vdots \\ p_{m,0} & \cdots & p_{m,n} \end{pmatrix}$$

Neighboring points $p_{i,j}$, $p_{i,j-1}$, $p_{i-1,j-1}$, $p_{i-1,j}$ $(1 \leq i \leq m\,; 1 \leq j \leq n)$ in the array define a quadrilateral; the entire mesh consists of $m \times n$ quadrilaterals.

Assuming `pts` is an $(m+1) \times (n+1)$ array of points, the function `QuadMeshGO_New(pts)` creates and returns a new quadrilateral mesh, consisting of $m \times n$ quadrilaterals.

`QuadMeshGO_NewWithShapeHint(pts,sh)` acts similarly, but it allows the client to supply a "shape hint". Shape hints are described in Section 3.22.

By default, the color of a `QuadMeshGO` is the color defined by the current value of the `SurfaceGO_Color` property. But `QuadMeshGO` objects have two methods (in addition to the methods common to all `SurfaceGO` objects) that allow the client to attach an individual color to each individual quadrilateral. Given a quad-mesh `qm` that consists of $m \times n$ quadrilaterals, `qm.addFacetColors(cols)` takes an $m \times n$ array of colors, and sets the color of the quadrilateral (i, j) to be `col[i][j]` (for all $1 \leq i \leq m\,; 1 \leq j \leq n$). If `cols` has incompatible dimensions, the exception `QuadMeshGO_BadSize` is raised.

`qm.setColorOfFacet(i,j,c)` assigns the color `c` to the quadrilateral (`i`, `j`) of the quad-mesh `qm`. This method may be invoked only if a color-array was previously attached to the quad-mesh (via `addFacetColors`), otherwise, the exception `QuadMeshGO_ColorsUndefined` is raised.

`QuadMeshGO` objects are affected by all the properties defined in the `GO` and `SurfaceGO` modules, and by no others.

3.30 The Prop Module

```
- help Prop;
  Prop_BadMethod: Exception
  Prop_BadInterval: Exception
TYPES
  Prop     <: ProxiedObj
  PropName <: ProxiedObj
  PropVal  <: ProxiedObj
  PropBeh  <: ProxiedObj
  PropRequest <: ProxiedObj & { start: () => Real, dur: () => Real }
```

A *property* is an object that describes the appearance of some particular aspect of a graphical object (its color, size, location, etc). A property consists of two parts: a *property name*, which states what aspect of the object is affected, and a *property value*, which states how it is affected.

Property names are objects of type `PropName`, or a subtype thereof. `PropName` in turn is a subtype of `ProxiedObj`; it does not add any extra fields or methods. There is a fixed set of property names; the client program cannot create new ones at run time.

Property values are objects which describe the value of a property at a given point in time. They are represented by objects of type `PropVal`. `PropVal` is an "abstract class"; it has subtypes that represent particular types of property values (such as real numbers or colors).

We just stated that property values encapsulate time-variant values. So, at a first approximation, they are functions from time to some value (reals, colors, etc). Because property values can be shared by many properties, and be referred to by many other property values, it is desirable to be able to change the function without having to create a new property value. For this reason, property values contain a *behavior*, which describes this function, and which can be changed without having to create a new property value.

Property behaviors are represented by objects of type `PropBeh`. This type is an "abstract class"; its subtypes implement mappings to concrete values like real numbers or colors. These subtypes have in turn several subtypes, for constant, asynchronous, synchronous, and dependent behaviors.

A *constant behavior* is a behavior whose value does not depend on the current time. An *asynchronous behavior* is a behavior whose value changes perpetually and over its entire lifetime, and does not depend on other property values. A *dependent behavior* is a behavior whose value is dependent on other property values. The value of an asynchronous or dependent behavior is computed using a method supplied by the client program. If this method is faulty (e.g. ill-typed), the exception `Prop_BadMethod` is raised.

Finally, a *synchronous behavior* is a behavior that accepts requests to perform changes over some finite period of time, and processes these changes only when a controlling animation handle (see Section 3.7) is signaled. Each synchronous behavior has a few methods that enqueue standard requests (such as "move at constant speed over a straight path") into its request queue. In addition, it allows the client program to enqueue arbitrary requests. Requests are represented by objects of type `PropRequest`, or a subtype thereof. Each request has a well-specified start time (relative to the time at which the controlling animation handle is signaled) and duration, and these values can be accessed by the methods `start` and `dur`.

Start time and duration determine a time interval for each request. By definition, zero-duration intervals are closed; all other intervals are open. Within the request queue of a synchronous behavior, no two requests may have overlapping time intervals. If a request that would cause an overlap is enqueued, the exception `Prop_BadInterval` is raised.

3.31 The BooleanProp Module

```
- help BooleanProp;
  BooleanProp_NewConst(b: Bool): BooleanPropVal
  BooleanProp_NewSync(ah: AnimHandle, b: Bool): BooleanPropVal
  BooleanProp_NewAsync(beh: BooleanPropAsyncBeh): BooleanPropVal
  BooleanProp_NewDep(beh: BooleanPropDepBeh): BooleanPropVal
  BooleanProp_NewConstBeh(b: Bool): BooleanPropConstBeh
  BooleanProp_NewSyncBeh(ah: AnimHandle, b: Bool): BooleanPropSyncBeh
  BooleanProp_NewAsyncBeh(compute: M1):BooleanPropAsyncBeh
  BooleanProp_NewDepBeh(compute: M2):BooleanPropDepBeh
  BooleanProp_NewRequest(start dur: Num, value: M3): BooleanPropRequest
WHERE
  BooleanPropName <: PropName & { bind: (v: BooleanPropVal) => Prop }
  BooleanPropVal <: PropVal & { getBeh: () => BooleanPropBeh,
                                setBeh: (BooleanPropBeh) => Ok,
                                get: () => Bool,
                                value: (Num) => Bool }
  BooleanPropBeh <: PropBeh
  BooleanPropConstBeh <: BooleanPropBeh & { set: (Bool) => Ok }
  BooleanPropSyncBeh <: BooleanPropBeh &
           { addRequest: (BooleanPropRequest) => Ok ! Prop_BadInterval,
             change: (Bool,Num) => Ok ! Prop_BadInterval }
  BooleanPropAsyncBeh <: BooleanPropBeh & { compute: M1 }
  BooleanPropDepBeh <: BooleanPropBeh & { compute: M2 }
  BooleanPropRequest <: PropRequest & { value: M3 }
  M1 = Self (X <: BooleanPropAsyncBeh) (Real) => Bool
  M2 = Self (X <: BooleanPropDepBeh) (Real) => Bool
  M3 = Self (X <: BooleanPropRequest) (Bool,Real) => Bool
  Num = Real + Int
```

A `BooleanPropName` is a property name that associates with a boolean property value. `BooleanPropName` is a subtype of `PropName` (see Section 3.30). It has one extra method, `bind`, which is for internal use only.

A `BooleanPropVal` is an object representing a boolean property value. `BooleanPropVal` is a subtype of `PropVal`. In addition to the methods common to all `PropVal` and `ProxiedObj` objects, a boolean property value `v` has four additional methods:

- `v.get()` returns the current value of `v`.
- `v.value(t)` returns the value of `v` at time `t`. [9]

[9] Assuming that `v` has the same behavior at time `t` as it has now

- `v.getBeh()` returns the behavior of `v`.
- `v.setBeh(beh)` changes the behavior of `v` to `beh`.

A `BooleanPropBeh` represents the behavior of a boolean property value. It has four subtypes:

1. `BooleanPropConstBeh` objects represent constant boolean behaviors. The function `BooleanProp_NewConstBeh(b)` takes a boolean `b` and returns a new constant behavior object whose initial value is `b`. The function `BooleanProp_NewConst(b)` returns a new boolean property value `v` whose behavior is `BooleanProp_NewConstBeh(b)`. In other words, `v.get()` evaluates to `b`.

   `BooleanPropConstBeh` objects understand one extra message: `beh.set(b)` changes the value of `beh` to `b`.

2. `BooleanPropAsyncBeh` represents asynchronous boolean behaviors. Objects of this type have one extra method, `compute`. This method is supplied by the client program, but meant to be called only by the animation server. The method defines the value of `beh` over time; `beh.compute(t)` returns the value of `beh` at time `t`.

   The function `BooleanProp_NewAsyncBeh(m)` takes a method `m` and returns a new asynchronous behavior `beh`, such that `beh.compute` is set to `m`.

   The function `BooleanProp_NewAsync(m)` returns a new boolean property value `v` whose behavior is `BooleanProp_NewAsyncBeh(m)`.

3. `BooleanPropDepBeh` objects represent dependent boolean behaviors, that is, boolean behaviors whose value depends on other property values. Dependent boolean behaviors have one extra method `compute`, just like asynchronous boolean behaviors; they also have similar creation functions. The main difference between the two types is that dependent boolean behaviors ensure that the dependency relationships are acyclic. Cyclic dependencies cause the exception `Prop_BadMethod` to be raised.

4. `BooleanPropSyncBeh` objects represent synchronous boolean behaviors. The function `BooleanProp_NewSyncBeh(ah,b)` takes an animation handle `ah` and a boolean `b`, and returns a new synchronous boolean behavior that is controlled by `ah` and whose initial value is `b`. `BooleanProp_NewSync(ah,b)` returns a new boolean property value whose behavior is `BooleanProp_NewSyncBeh(ah,b)`.

   Synchronous boolean behaviors have two extra methods: `beh.change(b,t)` puts a request into `beh`'s request queue that asks `beh` to change its current value to `b`, `t` seconds after the controlling animation handle `ah` is signaled. `beh.addRequest(r)` puts a client-defined request object `r` into `beh`'s request queue.

Recall that all request objects have start times and durations, and methods `start` and `dur` to access them (see Section 3.30). Boolean request objects have one extra method, `value`, which is supplied by the client program, but meant to be called only by the animation server. Assume that the boolean request `r` has a start time of `s` and a duration of `d`, and that it is enqueued in the request queue of a behavior `beh` whose controlling animation handle `ah` was signaled at time t_0. Then `r.value(`b_0`,t)` returns the value `beh` should have at time t_0`+t`, assuming that its value was b_0 at time `t+s`, just before `r` became active. `t` can range between `s` and `s+d`.

`BooleanProp_NewRequest(s,d,m)` returns a new boolean request object whose start time is `s`, whose duration is `d`, and whose `value` method is set to `m`.

3.32 The RealProp Module

```
- help RealProp;
  RealProp_NewConst(r: Num): RealPropVal
  RealProp_NewSync(ah: AnimHandle, r: Num): RealPropVal
  RealProp_NewAsync(beh: RealPropAsyncBeh): RealPropVal
  RealProp_NewDep(beh: RealPropDepBeh): RealPropVal
  RealProp_NewConstBeh(r: Num): RealPropConstBeh
  RealProp_NewSyncBeh(ah: AnimHandle, r: Num): RealPropSyncBeh
  RealProp_NewAsyncBeh(compute: M1):RealPropAsyncBeh
  RealProp_NewDepBeh(compute: M2):RealPropDepBeh
  RealProp_NewRequest(start dur: Num, value: M3): RealPropRequest
WHERE
  RealPropName <: PropName & { bind: (v: RealPropVal) => Prop }
  RealPropVal <: PropVal & { getBeh: () => RealPropBeh,
                             setBeh: (RealPropBeh) => Ok,
                             get: () => Real,
                             value: (Num) => Real }
  RealPropBeh <: PropBeh
  RealPropConstBeh <: RealPropBeh & { set: (Num) => Ok }
  RealPropSyncBeh <: RealPropBeh &
           { addRequest: (RealPropRequest) => Ok ! Prop_BadInterval,
             linChangeTo: (Num,Num,Num) => Ok ! Prop_BadInterval,
             linChangeBy: (Num,Num,Num) => Ok ! Prop_BadInterval }
  RealPropAsyncBeh <: RealPropBeh & { compute: M1 }
  RealPropDepBeh <: RealPropBeh & { compute: M2 }
  RealPropRequest <: PropRequest & { value: M3 }
  M1 = Self (X <: RealPropAsyncBeh) (Real) => Real
  M2 = Self (X <: RealPropDepBeh) (Real) => Real
  M3 = Self (X <: RealPropRequest) (Real,Real) => Real
  Num = Real + Int
```

A `RealPropName` is a property name that associates with a real property value. A `RealPropVal` is an object representing a real property value. A `RealPropBeh` represents the behavior of a real property value. It has four subtypes: `RealPropConstBeh`, which represents constant real behaviors; `RealPropAsyncBeh`, which represents asynchronous real behaviors; `RealPropDepBeh`, which represents dependent real behaviors; and `RealPropSyncBeh`, which represents synchronous real behaviors. Finally, a `RealPropRequest` is an object that represents a request to a synchronous real behavior.

The methods and creation functions associated with these types are completely analogous to those provided by the `BooleanProp` module, except for the methods of `RealPropSyncBeh` objects.

A synchronous real behavior is a `RealPropBeh` object that has three extra methods: `beh.linChangeTo(r,s,d)` puts a request into `beh`'s request queue that asks `beh` to change its current value to `r`. The change starts `s` seconds after the controlling animation handle `ah` is signaled, and completes `d` seconds later. The interpolation is linear, that is, the intermediate values move at constant speed through $\mathcal{R}$. The method call `beh.linChangeBy(r,s,d)` is similar, but changes the value of `beh` by `r`. Finally, `beh.addRequest(r)` puts a client-defined request object `r` into `beh`'s request queue.

3.33 The PointProp Module

```
- help PointProp;
  PointProp_NewConst(r: Point3): PointPropVal
  PointProp_NewSync(ah: AnimHandle, r: Point3): PointPropVal
  PointProp_NewAsync(beh: PointPropAsyncBeh): PointPropVal
  PointProp_NewDep(beh: PointPropDepBeh): PointPropVal
  PointProp_NewConstBeh(r: Point3): PointPropConstBeh
  PointProp_NewSyncBeh(ah: AnimHandle, r: Point3): PointPropSyncBeh
  PointProp_NewAsyncBeh(compute: M1):PointPropAsyncBeh
  PointProp_NewDepBeh(compute: M2):PointPropDepBeh
  PointProp_NewRequest(start dur: Num, value: M3): PointPropRequest
WHERE
  PointPropName <: PropName & { bind: (v: PointPropVal) => Prop }
  PointPropVal <: PropVal & { getBeh: () => PointPropBeh,
                              setBeh: (PointPropBeh) => Ok,
                              get: () => Point3,
                              value: (Num) => Point3 }
  PointPropBeh <: PropBeh
  PointPropConstBeh <: PointPropBeh & { set: (Point3) => Ok }
  PointPropSyncBeh <: PointPropBeh &
           { addRequest: (PointPropRequest) => Ok ! Prop_BadInterval,
             linMoveTo: (Point3,Num,Num) => Ok ! Prop_BadInterval,
             linMoveBy: (Point3,Num,Num) => Ok ! Prop_BadInterval }
  PointPropAsyncBeh <: PointPropBeh & { compute: M1 }
  PointPropDepBeh <: PointPropBeh & { compute: M2 }
  PointPropRequest <: PropRequest & { value: M3 }
  M1 = Self (X <: PointPropAsyncBeh) (Real) => Point3
  M2 = Self (X <: PointPropDepBeh) (Real) => Point3
  M3 = Self (X <: PointPropRequest) (Point3,Real) => Point3
  Num = Real + Int
```

A `PointPropName` is a property name that associates with a point property value. A `PointPropVal` is an object representing a point property value. A `PointPropBeh` represents the behavior of a point property value. It has four subtypes: `PointPropConstBeh`, which represents constant point behaviors; `PointPropAsyncBeh`, which represents asynchronous point behaviors; `PointPropDepBeh`, which represents dependent point behaviors; and `PointPropSyncBeh`, which represents synchronous point behaviors. Finally, a `PointPropRequest` is an object that represents a request to a synchronous point behavior.

The methods and creation functions associated with these types are completely analogous to those provided by the `BooleanProp` module, except for the methods of `PointPropSyncBeh` objects.

A synchronous point behavior is a `PointPropBeh` that has three extra methods: `beh.linMoveTo(p,s,d)` puts a request into `beh`'s request queue that asks `beh` to change its current value to `p`. The change starts `s` seconds after the controlling animation handle `ah` is signaled, and completes `d` seconds later. The interpolation is linear, that is, the intermediate values move at constant speed through $\mathcal{R}^3$. The method call `beh.linMoveBy(p,s,d)` is similar, but changes the value of `beh` by `p`. Finally, `beh.addRequest(r)` puts a client-defined request object `r` into `beh`'s request queue.

3.34 The ColorProp Module

```
- help ColorProp;
  ColorProp_NewConst(r: Col): ColorPropVal
  ColorProp_NewSync(ah: AnimHandle, r: Col): ColorPropVal
  ColorProp_NewAsync(beh: ColorPropAsyncBeh): ColorPropVal
  ColorProp_NewDep(beh: ColorPropDepBeh): ColorPropVal
  ColorProp_NewConstBeh(r: Col): ColorPropConstBeh
  ColorProp_NewSyncBeh(ah: AnimHandle, r: Col): ColorPropSyncBeh
  ColorProp_NewAsyncBeh(compute: M1):ColorPropAsyncBeh
  ColorProp_NewDepBeh(compute: M2):ColorPropDepBeh
  ColorProp_NewRequest(start dur: Num, value: M3): ColorPropRequest
WHERE
  ColorPropName <: PropName & { bind: (v: ColorPropVal) => Prop }
  ColorPropVal <: PropVal & { getBeh: () => ColorPropBeh,
                              setBeh: (ColorPropBeh) => Ok,
                              get: () => Color,
                              value: (Num) => Color }
  ColorPropBeh <: PropBeh
  ColorPropConstBeh <: ColorPropBeh & { set: (Col) => Ok }
  ColorPropSyncBeh <: ColorPropBeh &
     { addRequest: (ColorPropRequest) => Ok ! Prop_BadInterval,
       rgbLinChangeTo: (Col,Num,Num) => Ok ! Prop_BadInterval }
  ColorPropAsyncBeh <: ColorPropBeh & { compute: M1 }
  ColorPropDepBeh <: ColorPropBeh & { compute: M2 }
  ColorPropRequest <: PropRequest & { value: M3 }
  M1 = Self (X <: ColorPropAsyncBeh) (Real) => Color
  M2 = Self (X <: ColorPropDepBeh) (Real) => Color
  M3 = Self (X <: ColorPropRequest) (Color,Real) => Color
  Col = Color + Text
  Num = Real + Int
```

A `ColorPropName` is a property name that associates with a color property value. A `ColorPropVal` is an object representing a color property value. A `ColorPropBeh` represents the behavior of a color property value. It has four subtypes: `ColorPropConstBeh`, which represents constant color behaviors; `ColorPropAsyncBeh`, which represents asynchronous color behaviors; `ColorPropDepBeh`, which represents dependent color behaviors; and `ColorPropSyncBeh`, which represents synchronous color behaviors. Finally, a `ColorPropRequest` is an object that represents a request to a synchronous color behavior.

The creation functions for these types are similar to those provided by the `BooleanProp` module, except that they are overloaded: where the `BooleanProp` creation functions accepted a boolean as an argument, these creation functions accept either a `Color` or a `Text` describing a color. For example, the expression `ColorProp_NewConst("red")` is equivalent to `ColorProp_NewConst(color_named("red"))`.

The methods associated with these types are also analogous to those provided by the `BooleanProp` module, except for the methods of `ColorPropSyncBeh` objects.

Synchronous color behaviors are `ColorPropBeh` objects that have two extra methods: `beh.addRequest(r)` puts a client-defined request object `r` into `beh`'s request queue. `beh.rgbLinChangeTo(c,s,d)` puts a predefined request object into `beh`'s request queue that asks `beh` to change its current value *to* `c`. The change starts `s` seconds after the controlling animation handle `ah` is signaled, and completes `d` seconds later. The interpolation is linear through RGB space, that is, the intermediate colors move on a straight path and at a constant speed through the RGB cube.

3.35 The TransformProp Module

```
- help TransformProp;
  TransformProp_NewConst(m: Matrix4): TransformPropVal
  TransformProp_NewSync(ah: AnimHandle, m: Matrix4): TransformPropVal
  TransformProp_NewAsync(beh: TransformPropAsyncBeh): TransformPropVal
  TransformProp_NewDep(beh: TransformPropDepBeh): TransformPropVal
  TransformProp_NewConstBeh(m: Matrix4): TransformPropConstBeh
  TransformProp_NewSyncBeh(ah: AnimHandle, m: Matrix4): TransformPropSyncBeh
  TransformProp_NewAsyncBeh(compute: M1):TransformPropAsyncBeh
  TransformProp_NewDepBeh(compute: M2):TransformPropDepBeh
  TransformProp_NewRequest(start dur: Num, value: M3): TransformPropRequest
WHERE
  TransformPropName <: PropName & { bind: (v: TransformPropVal) => Prop }
  TransformPropVal <: PropVal & { getBeh: () => TransformPropBeh,
                                  setBeh: (TransformPropBeh) => Ok,
                                  get: () => Matrix4,
                                  value: (Num) => Matrix4 }
  TransformPropBeh <: PropBeh
  TransformPropConstBeh <: TransformPropBeh &
                                { set: (Matrix4) => Ok,
                                  compose: (Matrix4) => Ok,
                                  reset: () => Ok,
                                  translate: (Num,Num,Num) => Ok,
                                  scale: (Num,Num,Num) => Ok,
                                  rotateX: (Num) => Ok,
                                  rotateY: (Num) => Ok,
                                  rotateZ: (Num) => Ok }
  TransformPropSyncBeh <: TransformPropBeh &
       { addRequest: (TransformPropRequest) => Ok ! Prop_BadInterval,
         reset: (Num) => Ok ! Prop_BadInterval,
         changeTo: (Matrix4,Num,Num) => Ok ! Prop_BadInterval,
         translate: (Num,Num,Num,Num,Num) => Ok ! Prop_BadInterval,
         scale: (Num,Num,Num,Num,Num) => Ok ! Prop_BadInterval,
         rotateX: (Num,Num,Num) => Ok ! Prop_BadInterval,
         rotateY: (Num,Num,Num) => Ok ! Prop_BadInterval,
         rotateZ: (Num,Num,Num) => Ok ! Prop_BadInterval }
  TransformPropAsyncBeh <: TransformPropBeh & { compute: M1 }
  TransformPropDepBeh <: TransformPropBeh & { compute: M2 }
  TransformPropRequest <: PropRequest & { value: M3 }
  M1 = Self (X <: TransformPropAsyncBeh) (Real) => Matrix4
  M2 = Self (X <: TransformPropDepBeh) (Real) => Matrix4
  M3 = Self (X <: TransformPropRequest) (Matrix4,Real) => Matrix4
  Num = Real = Int
```

A TransformPropName is a property name that associates with a transformation property value. A TransformPropVal is an object representing a transformation property value. A TransformPropBeh represents the behavior of a transformation property value. It has four subtypes: TransformPropConstBeh represents constant transformation behaviors; TransformPropAsyncBeh represents asynchronous transformation behaviors; TransformPropDepBeh represents dependent transformation behaviors; and TransformPropSyncBeh represents synchronous transformation behaviors. Finally, a TransformPropRequest is an object that represents a request to a synchronous transformation behavior.

The methods and creation functions associated with these types are completely analogous to those provided by the BooleanProp module, except for the methods of PointPropConstBeh and PointPropSyncBeh objects.

TransformPropConstBeh is a subtype of TransformPropBeh that adds 8 extra methods. In the following, assume that n denotes the current value (a Matrix4; see Section 3.2) of the constant transformation behavior beh:

- beh.set(m) changes the value of beh to m.
- beh.reset() changes the value of beh to Matrix4_Id.
- beh.compose(m) changes the value of beh to Matrix4_Multiply(m,n).
- beh.translate(x,y,z) changes the value of beh to Matrix4_Translate(n,x,y,z).
- beh.scale(x,y,z) changes the value of beh to Matrix4_Scale(n,x,y,z).
- beh.rotateX(a) changes the value of beh to Matrix4_RotateX(n,a).
- beh.rotateY(a) changes the value of beh to Matrix4_RotateY(n,a).
- beh.rotateZ(a) changes the value of beh to Matrix4_RotateZ(n,a).

TransformPropSyncBeh is another subtype of TransformPropBeh that also adds 8 extra methods. The first method, beh.addRequest(r), puts a client-defined request object r into beh's request queue. The remaining seven methods all put predefined request objects into the behavior's request queue, asking it to change its current value (call it n) to a new value. The methods take arguments s and d, indicating that the requested change should start s seconds after the controlling animation handle is signaled, and should be completed d seconds later.

- beh.reset(s) enqueues a request to change to the value Matrix4_Id. This method has no argument d, so there is no interpolation between n and Matrix4_Id.

- `beh.changeTo(m,s,d)` enqueues a request to change to the value `m`. The interpolation between the two matrices is done so that it leads to a smooth-looking motion of the scene. Both `m` and `n` must be matrices that can be constructed using only translation, rotation, and *uniform* scaling operations.

- `beh.translate(x,y,z,s,d)` enqueues a request to change to the value `Matrix4_Translate(n,x,y,z)`.

- `beh.scale(x,y,z,s,d)` enqueues a request to change to the value `Matrix4_Scale(n,x,y,z)`.

- `beh.rotateX(a,s,d)` enqueues a request to change to the value `Matrix4_RotateX(n,a)`.

- `beh.rotateY(a,s,d)` enqueues a request to change to the value `Matrix4_RotateY(n,a)`.

- `beh.rotateZ(a,s,d)` enqueues a request to change to the value `Matrix4_RotateZ(n,a)`.

3.36 The LineTypeProp Module

```
- help LineTypeProp;
  LineTypeProp_NewConst(lt: LineType): LineTypePropVal
  LineTypeProp_NewSync(ah: AnimHandle, lt: LineType): LineTypePropVal
  LineTypeProp_NewAsync(beh: LineTypePropAsyncBeh): LineTypePropVal
  LineTypeProp_NewDep(beh: LineTypePropDepBeh): LineTypePropVal
  LineTypeProp_NewConstBeh(lt: LineType): LineTypePropConstBeh
  LineTypeProp_NewSyncBeh(ah: AnimHandle, lt: LineType): LineTypePropSyncBeh
  LineTypeProp_NewAsyncBeh(compute: M1):LineTypePropAsyncBeh
  LineTypeProp_NewDepBeh(compute: M2):LineTypePropDepBeh
  LineTypeProp_NewRequest(start dur: Num, value: M3): LineTypePropRequest
WHERE
  LineTypePropName <: PropName & { bind: (v: LineTypePropVal) => Prop }
  LineTypePropVal <: PropVal & { getBeh: () => LineTypePropBeh,
                                 setBeh: (LineTypePropBeh) => Ok,
                                 get: () => LineType,
                                 value: (Num) => LineType }
  LineTypePropBeh <: PropBeh
  LineTypePropConstBeh <: LineTypePropBeh & { set: (LineType) => Ok }
  LineTypePropSyncBeh <: LineTypePropBeh &
          { addRequest: (LineTypePropRequest) => Ok ! Prop_BadInterval,
            change: (LineType,Num) => Ok ! Prop_BadInterval }
  LineTypePropAsyncBeh <: LineTypePropBeh & { compute: M1 }
  LineTypePropDepBeh <: LineTypePropBeh & { compute: M2 }
  LineTypePropRequest <: PropRequest & { value: M3 }
  M1 = Self (X <: LineTypePropAsyncBeh) (Real) => LineType
  M2 = Self (X <: LineTypePropDepBeh) (Real) => LineType
  M3 = Self (X <: LineTypePropRequest) (LineType,Real) => LineType
  LineType = Text  (one of "Solid", "Dashed", "Dotted", "DashDot")
  Num = Real + Int
```

There are two types of graphical objects that use lines: `LineGO` objects, which represent lines themselves, and `SurfaceGO` objects (including spheres, cones, etc). The latter are composed of individual polygons, and the outline of a polygon can be surrounded by a line.

Obliq-3D supports four different line styles: solid, dashed, dotted, and alternatingly dashed and dotted lines. We identify these line types by the strings `"Solid"`, `"Dashed"`, `"Dotted"`, and `"DashDot"`, and refer to the set of these four strings as the type `LineType`.

The `LineTypeProp` module supplies line type property names, values, behaviors, and request objects. All of the types and functions exported by the module are completely analogous to those exported by the `BooleanProp` module, with all occurrences of `Bool` replaced by `LineType`.

3.37 The MarkerTypeProp Module

```
- help MarkerTypeProp;
  MarkerTypeProp_NewConst(lt: MarkerType): MarkerTypePropVal
  MarkerTypeProp_NewSync(ah: AnimHandle, lt: MarkerType): MarkerTypePropVal
  MarkerTypeProp_NewAsync(beh: MarkerTypePropAsyncBeh): MarkerTypePropVal
  MarkerTypeProp_NewDep(beh: MarkerTypePropDepBeh): MarkerTypePropVal
  MarkerTypeProp_NewConstBeh(lt: MarkerType): MarkerTypePropConstBeh
  MarkerTypeProp_NewSyncBeh(ah: AnimHandle,
                            t: MarkerType): MarkerTypePropSyncBeh
  MarkerTypeProp_NewAsyncBeh(compute: M1):MarkerTypePropAsyncBeh
  MarkerTypeProp_NewDepBeh(compute: M2):MarkerTypePropDepBeh
  MarkerTypeProp_NewRequest(start dur: Num, value: M3): MarkerTypePropRequest
WHERE
  MarkerTypePropName <: PropName & { bind: (v: MarkerTypePropVal) => Prop }
  MarkerTypePropVal <: PropVal & { getBeh: () => MarkerTypePropBeh,
                                   setBeh: (MarkerTypePropBeh) => Ok,
                                   get: () => MarkerType,
                                   value: (Num) => MarkerType }
  MarkerTypePropBeh <: PropBeh
  MarkerTypePropConstBeh <: MarkerTypePropBeh & { set: (MarkerType) => Ok }
  MarkerTypePropSyncBeh <: MarkerTypePropBeh &
         { addRequest: (MarkerTypePropRequest) => Ok ! Prop_BadInterval,
           change: (MarkerType,Num) => Ok ! Prop_BadInterval }
  MarkerTypePropAsyncBeh <: MarkerTypePropBeh & { compute: M1 }
  MarkerTypePropDepBeh <: MarkerTypePropBeh & { compute: M2 }
  MarkerTypePropRequest <: PropRequest & { value: M3 }
  M1 = Self (X <: MarkerTypePropAsyncBeh) (Real) => MarkerType
  M2 = Self (X <: MarkerTypePropDepBeh) (Real) => MarkerType
  M3 = Self (X <: MarkerTypePropRequest) (MarkerType,Real) => MarkerType
  MarkerType = Text  (one of "Dot", "Circle", "Cross", "Asterisk", "X")
  Num = Real + Int
```

`MarkerGO` objects represent markers, that is, objects that mark a single point. Obliq-3D supports five different marker styles: $\cdot$, $\circ$, $+$, $*$, and $\times$. We identify these marker types by the strings `"Dot"`, `"Circle"`, `"Cross"`, `"Asterisk"`, and `"X"`, and refer to the set of these five strings as the type `MarkerType`.

The `MarkerTypeProp` module supplies marker type property names, values, behaviors, and request objects. All of the types and functions exported by the module are completely analogous to those exported by the `BooleanProp` module, with all occurrences of `Bool` replaced by `MarkerType`.

3.38 The RasterModeProp Module

```
- help RasterModeProp;
  RasterModeProp_NewConst(lt: RasterMode): RasterModePropVal
  RasterModeProp_NewSync(ah: AnimHandle, lt: RasterMode): RasterModePropVal
  RasterModeProp_NewAsync(beh: RasterModePropAsyncBeh): RasterModePropVal
  RasterModeProp_NewDep(beh: RasterModePropDepBeh): RasterModePropVal
  RasterModeProp_NewConstBeh(lt: RasterMode): RasterModePropConstBeh
  RasterModeProp_NewSyncBeh(ah: AnimHandle,
                           lt: RasterMode): RasterModePropSyncBeh
  RasterModeProp_NewAsyncBeh(compute: M1):RasterModePropAsyncBeh
  RasterModeProp_NewDepBeh(compute: M2):RasterModePropDepBeh
  RasterModeProp_NewRequest(start dur: Num, value: M3): RasterModePropRequest
WHERE
  RasterModePropName <: PropName &
      { bind: (v: RasterModePropVal) => Prop }
  RasterModePropVal <: PropVal & { getBeh: () => RasterModePropBeh,
                                   setBeh: (RasterModePropBeh) => Ok,
                                   get: () => RasterMode,
                                   value: (Num) => RasterMode }
  RasterModePropBeh <: PropBeh
  RasterModePropConstBeh <: RasterModePropBeh &
      { set: (RasterMode) => Ok }
  RasterModePropSyncBeh <: RasterModePropBeh &
      { addRequest: (RasterModePropRequest) => Ok ! Prop_BadInterval,
        change: (RasterMode,Num) => Ok ! Prop_BadInterval }
  RasterModePropAsyncBeh <: RasterModePropBeh & { compute: M1 }
  RasterModePropDepBeh <: RasterModePropBeh & { compute: M2 }
  RasterModePropRequest <: PropRequest & { value: M3 }
  M1 = Self (X <: RasterModePropAsyncBeh) (Real) => RasterMode
  M2 = Self (X <: RasterModePropDepBeh) (Real) => RasterMode
  M3 = Self (X <: RasterModePropRequest) (RasterMode,Real) => RasterMode
  RasterMode = Text  (one of "Hollow", "Solid", "Empty")
  Num = Real + Int
```

`SurfaceGO` objects represent surfaces, and these surfaces are composed of polygons. Obliq-3D supports three different rendering styles for polygons: they can be solid, meaning that the entire surface is shown, they can be hollow, meaning that only the edges are shown, or they can be empty, meaning that neither interior nor edges are shown. We identify these styles by the strings `"Solid"`, `"Hollow"`, and `"Empty"`, and refer to the set of these three strings as the type `RasterMode`.

The `RasterModeProp` module supplies raster mode property names, values, behaviors, and request objects. All of the types and functions exported by the module are completely analogous to those exported by the `BooleanProp` module, with all occurrences of `Bool` replaced by `RasterMode`.

3.39 The ShadingProp Module

```
- help ShadingProp;
  ShadingProp_NewConst(lt: Shading): ShadingPropVal
  ShadingProp_NewSync(ah: AnimHandle, lt: Shading): ShadingPropVal
  ShadingProp_NewAsync(beh: ShadingPropAsyncBeh): ShadingPropVal
  ShadingProp_NewDep(beh: ShadingPropDepBeh): ShadingPropVal
  ShadingProp_NewConstBeh(lt: Shading): ShadingPropConstBeh
  ShadingProp_NewSyncBeh(ah: AnimHandle, lt: Shading): ShadingPropSyncBeh
  ShadingProp_NewAsyncBeh(compute: M1):ShadingPropAsyncBeh
  ShadingProp_NewDepBeh(compute: M2):ShadingPropDepBeh
  ShadingProp_NewRequest(start dur: Num, value: M3): ShadingPropRequest
WHERE
  ShadingPropName <: PropName & { bind: (v: ShadingPropVal) => Prop }
  ShadingPropVal <: PropVal & { getBeh: () => ShadingPropBeh,
                                setBeh: (ShadingPropBeh) => Ok,
                                get: () => Shading,
                                value: (Num) => Shading }
  ShadingPropBeh <: PropBeh
  ShadingPropConstBeh <: ShadingPropBeh & { set: (Shading) => Ok }
  ShadingPropSyncBeh <: ShadingPropBeh &
      { addRequest: (ShadingPropRequest) => Ok ! Prop_BadInterval,
        change: (Shading,Num) => Ok ! Prop_BadInterval }
  ShadingPropAsyncBeh <: ShadingPropBeh & { compute : M1 }
  ShadingPropDepBeh <: ShadingPropBeh & { compute: M2 }
  ShadingPropRequest <: PropRequest & { value: M3 }
  M1 = Self (X <: ShadingPropAsyncBeh) (Real) => Shading
  M2 = Self (X <: ShadingPropDepBeh) (Real) => Shading
  M3 = Self (X <: ShadingPropRequest) (Shading,Real) => Shading
  Shading = Text  ("Flat" or "Gouraud")
  Num = Real + Int
```

Obliq-3D supports two shading methods for polygonal surfaces: Flat shading and Gouraud shading. We identify these styles by the strings `"Flat"` and `"Gouraud"`, and refer to the set of these two strings as the type `Shading`.

The PEX 5.0 standard provides for two other shading methods, namely "dot-product" shading and Phong shading. However, since most implementations of PEX (including the ones available to us) do not support these two shading methods, and since OpenGL does not provide them at all, we did not include them in Obliq-3D.

The `ShadingProp` module supplies shading property names, values, behaviors, and request objects. All of the types and functions exported by the module are completely analogous to those exported by the `BooleanProp` module, with all occurrences of `Bool` replaced by `Shading`.

3.40 The MouseCB Module

```
- help MouseCB;
  MouseCB_New(invoke: M): MouseCB
WHERE
  MouseCB <: ProxiedObj & { invoke: M }
  M = Self (X <: MouseCB) (MouseRec) => Ok
  MouseRec = { pos: Point2, change: Button,
               modifiers: [Modifier], clickType: ClickType }
  Point2 = [2*Int]
  Button = Text      (one of "Left", "Middle", "Right")
  Modifier = Text    (a Button or one of "Shift", "Lock", "Control", "Option")
  ClickType = Text   (one of "FirstDown", "OtherDown", "OtherUp", "LastUp")
```

Section 3.8 described Obliq-3D's basic model of how to specify interactive behavior. User actions (such as key strokes or mouse actions) that occur within the scope of an animation window cause *events* to be forwarded to the root object associated with the window. Each graphical object (including the root object) contains a stack of callback objects for each event type. Callback objects are used to encapsulate reactive behavior. The topmost callback object on each stack determines how the graphical object reacts to an event of the corresponding type. One possible response is to forward the event to the children of the graphical object.

A `MouseCB`, or *mouse callback object*, describes the reactive behavior of a graphical object in response to a mouse button transition. `MouseCB` is a subtype of `ProxiedObj`; it adds one extra method, `invoke`, which is supplied by the client, and called by the animation server in response to a mouse button transition. `invoke` takes two arguments, the mouse callback object `self` that is receiving the message, and a mouse event record `mr`.

A *mouse event record* is an object with four fields, `pos`, `change`, `modifiers`, and `clickType`. Note that `MouseRec` has no supertype and does not contain any methods; in other words, its Modula-3 equivalent would be a record rather than an object.

The `pos` field contains a `Point2`, an array of two integers. `pos` describes the position of the mouse pointer when the button transition occurred.

The `change` field contains one of the following strings: `"Left"`, `"Middle"`, or `"Right"`. It describes which mouse button was pressed or released.

The `modifiers` field contains an array of strings; each string can have one of the following values: `"Left"`, `"Middle"`, or `"Right"`, `"Shift"`, `"Lock"`, `"Control"`, or `"Option"`.

The `modifiers` field describes which other mouse buttons and keyboard modifier keys were depressed while the mouse button transition occurred.

Finally, the `clickType` field contains one of the following strings: `"FirstDown"`, `"OtherDown"`, `"OtherUp"`, or `"LastUp"`. It describes whether the mouse button went up or down, and whether any other mouse buttons were pressed at that time.

`MouseCB_New(m)` creates a new mouse callback object, sets its `invoke` method to `m`, and returns it.

3.41 The PositionCB Module

```
- help PositionCB;
  PositionCB_New(invoke: M): PositionCB
WHERE
  PositionCB <: ProxiedObj & { invoke: M }
  M = Self (X <: PositionCB) (PositionRec) => Ok
  PositionRec = { pos: Point2, modifiers: [Modifier] }
  Point2 = [2*Int]
  Modifier = Text (one of "Left", "Middle", "Right",
                   "Shift", "Lock", "Control", "Option")
```

A `PositionCB`, or *position callback object*, describes the reactive behavior of a graphical object in response to a change in mouse position. `PositionCB` is a subtype of `ProxiedObj`; it adds one extra method, `invoke`, which is supplied by the client, and called by the animation server in response to a position change. `invoke` takes two arguments, the position callback object `self` that is receiving the message, and a position event record `pr`.

A *position event record* is an object (really a record) with two fields, `pos` and `modifiers`. `pos` describes the new position of the mouse pointer; `modifiers` describes which other mouse buttons and keyboard modifier keys were depressed while the position change occurred. The two fields have the same types as the corresponding `MouseRec` fields (see Section 3.40).

`PositionCB_New(m)` creates a new position callback object, sets its `invoke` method to `m`, and returns it.

3.42 The KeyCB Module

```
- help KeyCB;
  KeyCB_New(invoke: M): KeyCB
WHERE
  KeyCB <: ProxiedObj & { invoke: M }
  M = Self (X <: KeyCB) (KeyRec) => Ok
  KeyRec = { change: Text, wentDown: Bool, modifiers: [Modifier] }
  Modifier = Text (one of "Left", "Middle", "Right",
                    "Shift", "Lock", "Control", "Option")
```

A `KeyCB`, or *key callback object*, describes the reactive behavior of a graphical object in response to a keyboard key transition. `KeyCB` is a subtype of `ProxiedObj`; it adds one extra method, `invoke`, which is supplied by the client, and called by the animation server in response to a key transition. `invoke` takes two arguments, the key callback object `self` that is receiving the message, and a key event record `kr`.

A *key event record* is an object (really a record) with three fields, `change`, `wentDown`, and `modifiers`. The `change` field contains a string indicating which key was pressed or released. The `wentDown` field contains a boolean indicating whether the key went down or up. Finally, the `modifiers` field describes which other mouse buttons and keyboard modifier keys were depressed while the key transition occurred. Its type is the same as the type of the `modifiers` field of `MouseRec` (see Section 3.40).

`KeyCB_New(m)` creates a new key callback object, sets its `invoke` method to `m`, and returns it.

Acknowledgments

Anim3D and Obliq-3D were designed jointly with Marc Brown. The final design of the system owes much to Marc's taste for solutions that are both simple and elegant. Stephen Harrison made some important contributions to the early design of the system. Luca Cardelli designed and implemented Obliq, the interpreted language used by our animation system. Luca also helped in embedding the Obliq interpreter into Obliq-3D. Lyle Ramshaw explained the relationship between transformation matrices and quaternions to us. All drawings in this report were produced using Juno-II, a constraint-based drawing editor developed by Allan Heydon and Greg Nelson. Marc, Allan, and Luca read various drafts of this report; their comments improved both technical accuracy and clarity of the exposition. Finally, thanks to Cynthia Hibbard for helping to improve the presentation, and for explaining the subtleties of the defining relative in the process.

References

[1] American National Standards Institute. **American National Standard for Information Processing Systems — Programmer's Hierarchical Interactive Graphics System (PHIGS) Functional Description, Archive File Format, Clear-Text Encoding of Archive File**, X3.144-1988, American National Standards Institute, New York, NY, 1988.

[2] Marc H. Brown and Marc A. Najork. **Algorithm Animation using 3D Interactive Graphics**. In *ACM Symposium on User Interface Software and Technology*, 93 – 100 (November 1993). Also appeared as Research Report 110a, Digital Equipment Corp., Systems Research Center, Palo Alto, CA (September 1993). There is an accompanying video tape, SRC Research Report 110b.

[3] Luca Cardelli. **Obliq — A Language with Distributed Scope**. Research Report 122, Digital Equipment Corp., Systems Research Center, Palo Alto, CA (June 1994).

[4] Matthew Conway: Randy Pausch, Rich Gossweiler, and Tommy Burnette. **Alice: A Rapid Prototyping System for Building Virtual Environments**. In *Conference Companion, CHI'94*, 295–296 (April 1994).

[5] Robert DeLine. **Alice: A Rapid Prototyping System for Three-Dimensional Interactive Graphical Environments**. University of Virginia, Dept. of Computer Science (May 1993).

[6] Parris Egbert and William Kubitz. **Application Graphics Modelling Support Through Object Orientation**. *IEEE Computer*, 84–91 (October 1992).

[7] Conal Elliott, Greg Schechter, Ricky Yeung, Salim Abi-Ezzi. **TBAG: A High Level Framework for Interactive, Animated 3D Graphics Applications**. *Computer Graphics Proceedings, Annual Conference Series (SIGGRAPH '94 Proceedings)*, 421–434 (July 1994).

[8] Tom Gaskins. **PEXlib Programming Manual**. O'Reilly & Associates, Inc., 1992.

[9] Graphics Standards Planning Committee. **Status Report of the Graphics Standards Planning Committee**. *Computer Graphics*, **13**(3) (August 1979).

[10] Philip M. Hubbard, Matthias M. Wloka, and Robert C. Zeleznik. **UGA: A Unified Graphics Architecture**. Technical Report CS-91-30, Brown University, Dept. of Computer Science, Providence, RI (June 1991).

[11] International Standards Organization. **International Standard Information Processing Systems — Computer Graphics — Graphical Kernel System for Three Dimensions**

(GKS–3D) Functional Description, ISO Document Number 8805:1988(E), American National Standards Institute, New York, NY, 1988.

[12] Larry Koved and Wayne Wooten. **GROOP: An object-oriented toolkit for animated 3D graphics**. *ACM SIGPLAN Notices (OOPSLA '93 Proceedings)* **28**(10):309–325 (October 1993).

[13] **Doré Programmer's Guide, Release 5.0**, Kubota Pacific Computer Inc., Santa Clara, CA, 1991.

[14] Mark S. Manasse and Greg Nelson. **Trestle Reference Manual**. Research Report 68, Digital Equipment Corp., Systems Research Center, Palo Alto, CA (December 1991).

[15] Mark S. Manasse and Greg Nelson. **Trestle Tutorial**. Research Report 69, Digital Equipment Corp., Systems Research Center, Palo Alto, CA (May 1992).

[16] Marc A. Najork and Marc H. Brown. **A Library for Visualizing Combinatorial Structures**. In *IEEE Visualization '94* (1994). Also appeared as Research Report 128a, Digital Equipment Corp., Systems Research Center, Palo Alto, CA (September 1994). There is an accompanying video tape, SRC Research Report 128b.

[17] Greg Nelson (Editor). **Systems Programming with Modula-3**. Prentice-Hall, 1991.

[18] OpenGL Architecture Review Board. **OpenGL Reference Manual**. Addison-Wesley, 1992.

[19] PHIGS+ Committee, Andies van Dam, chair. **PHIGS+ Functional Description, Revision 3.0**. *Computer Graphics*, 22(3):125–218 (July 1988).

[20] Greg Schechter, Conal Elliott, Ricky Yeung, and Salim Abi-Ezzi. **Functional 3D Graphics in C++ — with an Object-Oriented, Multiple Dispatching Implementation**. In *Proc.4th Eurographics Workshop on Object-Oriented Graphics*, 1994.

[21] Robert W. Scheifler and James Gettys. **X Window System**. Third Edition. Digital Press, 1992.

[22] **Graphics Library Programming Guide**. Silicon Graphics Inc., Mountain View, CA, 1991.

[23] Paul S. Strauss. **BAGS: The Brown Animation Generation System**. Technical Report No. CS-88-22, Brown University, Dept. of Computer Science, Providence, RI (May 1988).

[24] Paul S. Strauss. **IRIS Inventor, a 3D Graphics Toolkit**. *ACM SIGPLAN Notices (OOPSLA '93 Proceedings)* **28**(10):192–200 (October 1993).

[25] Paul S. Strauss and Rikk Carey. **An Object-Oriented 3D Graphics Toolkit**. *ACM Computer Graphics (SIGGRAPH '92)* **26**(2):341–349 (July 1992).

[26] Josie Wernecke. **The Inventor Mentor**. Addison-Wesley, 1994.

[27] Gary Wiegand and Bob Bovey. **HOOPS Reference Manual, Version 3.0**, Ithaca Software, 1991.

[28] Robert C. Zeleznik, D. Brookshire Conner, Matthias M. Wloka, Daniel G. Aliaga, Nathan T. Huang, Philip M. Hubbard, Brian Knep, Henry Kaufman, John F. Hughes and Andries van Dam. An Object-Oriented Framework for the Integration of Interactive Animation Techniques. **Computer Graphics (SIGGRAPH '91 Proceedings) 25**(4):105–112 (July 1991).

Index

`add` method, 5, *54*
`addFacetColors` method, 79
`addRequest` method
  of `BooleanPropSyncBeh` type, *83*
  of `ColorPropSyncBeh` type, *89*
  of `LineTypePropSyncBeh` type, *93*
  of `MarkerTypePropSyncBeh` type, *94*
  of `PointPropSyncBeh` type, *87*
  of `RasterModePropSyncBeh` type, *95*
  of `RealPropSyncBeh` type, *86*
  of `ShadingPropSyncBeh` type, *96*
  of `TransformPropSyncBeh` type, *91*
`AmbientLightGO` module, *57*
`AmbientLightGO` type, *57*
`AmimHandle` module, *48*
`AmimHandle` type, *48*
Anim3D animation library, 1
  connection to Obliq-3D, 26–28
`Anim3D` module, 21–24, *46*
`animate` method, *48*
animation handle, 18, *48*, 81
attenuation coefficient
  constant, *60*
  linear, *60*
  quadratic, *60*
attenuation factor, *59*
`awaitDelete` method, *47*

behavior, 16, *80*
  asynchronous, 17, *81*
  constant, 16, *81*
  dependent, 20, *81*
  synchronous, 18, 29, 48, *81*
  within property value
    accessing, 17, *83*
    updating, 17, *83*

`bind` method
  of `BooleanPropName` type, *82*
  of `ColorPropName` type, *88*
  of `LineTypePropName` type, *93*
  of `MarkerTypePropName` type, *94*
  of `PointPropName` type, *86*
  of `RasterModePropName` type, *95*
  of `RealPropName` type, *85*
  of `ShadingPropName` type, *96*
  of `TransformPropName` type, *91*
`BooleanProp` module, *82*
`BooleanPropAsyncBeh` type, *83*
`BooleanPropBeh` type, *83*
`BooleanPropConstBeh` type, *83*
`BooleanPropDepBeh` type, *83*
`BooleanPropName` type, *82*
`BooleanPropRequest` type, *84*
`BooleanPropSyncBeh` type, *83*
`BooleanPropVal` type, *82*
`BooleanVal` type, 43
box, *73*
`BoxGO` module, *73*
`BoxGO` type, *73*
`Button` type, *97*

callback object, 4, 52
  key, *99*
  mouse, *97*
  position, *98*
callback stack, 25
camera, 5, *63*
  orthographic, *65*
  perspective, *66*
`CameraGO` module, *63*
`CameraGO` type, *63*
`change` field

  of `KeyRec` type, *99*
  of `MouseRec` type, *97*
`change` method
  of `BooleanPropSyncBeh` type, *83*
  of `LineTypePropSyncBeh` type, *93*
  of `MarkerTypePropSyncBeh` type, *94*
  of `RasterModePropSyncBeh` type, *95*
  of `ShadingPropSyncBeh` type, *96*
`changeCamera` method, *56*
`changeTitle` method, *47*
`changeTo` method
  of `TransformPropSyncBeh` type, *92*
`clickType` field
  of `MouseRec` type, *98*
`Col` type, 43
`ColorProp` module, *88*
`ColorPropAsyncBeh` type, *88*
`ColorPropBeh` type, *88*
`ColorPropConstBeh` type, *88*
`ColorPropDepBeh` type, *88*
`ColorPropName` type, *88*
`ColorPropRequest` type, *88*
`ColorPropSyncBeh` type, *88*
`ColorPropVal` type, *88*
`ColorVal` type, 43
`compose` method
  of `TransformPropConstBeh` type, *91*
`compute` method
  of `BooleanPropAsyncBeh` type, *83*
  of `BooleanPropDepBeh` type, *83*
  of `ColorPropAsyncBeh` type, *88*
  of `ColorPropDepBeh` type, *88*
  of `LineTypePropAsyncBeh` type, *93*
  of `LineTypePropDepBeh` type, *93*
  of `MarkerTypePropAsyncBeh` type, *94*
  of `MarkerTypePropDepBeh` type, *94*
  of `PointPropAsyncBeh` type, *86*
  of `PointPropDepBeh` type, *86*
  of `RasterModePropAsyncBeh` type, *95*
  of `RasterModePropDepBeh` type, *95*
  of `RealPropAsyncBeh` type, *85*
  of `RealPropDepBeh` type, *85*
  of `ShadingPropAsyncBeh` type, *96*
  of `ShadingPropDepBeh` type, *96*
  of `TransformPropAsyncBeh` type, *91*
  of `TransformPropDepBeh` type, *91*
cone, 6, *77*
`ConeGO` module, *77*
`ConeGO` type, *77*
`content` method, *54*
cylinder, *76*
`CylinderGO` module, *76*
`CylinderGO` type, *76*

`default-ambient-light` name, 28, 40
`default-camera` name, 28, 40
`default-vector-light` name, 28, 40
`destroy` method, *47*
disk, *74*
`DiskGO` module, *74*
`DiskGO` type, *74*
`dur` method, *81*

event, 52, *97*
  key, 24, 52
  mouse, 24, 52
  position, 24, 52
event record, 52
  key, *99*
  mouse, 24, 52, *97*
  position, *98*
`extend` method, *46*

`findName` method, *52*
`flush` method, *54*

`get` method

of `BooleanPropVal` type, *82*
of `ColorPropVal` type, *88*
of `LineTypePropVal` type, *93*
of `MarkerTypePropVal` type, *94*
of `PointPropVal` type, *86*
of `RasterModePropVal` type, *95*
of `RealPropVal` type, *85*
of `ShadingPropVal` type, *96*
of `TransformPropVal` type, *91*
`getBeh` method
of `BooleanPropVal` type, *83*
of `ColorPropVal` type, *88*
of `LineTypePropVal` type, *93*
of `MarkerTypePropVal` type, *94*
of `PointPropVal` type, *86*
of `RasterModePropVal` type, *95*
of `RealPropVal` type, *85*
of `ShadingPropVal` type, *96*
of `TransformPropVal` type, *91*
`getName` method, *52*
`getProp` method, *52*
`GO` module, *49*
`GO` type, *49*
graphical object, 4, *49*
`GraphicsBase` module, *47*
`GraphicsBase` type, *47*
group, 4, *54*
`GroupGO` module, *54*
`GroupGO` type, *54*

`invoke` method
of `KeyCB` type, *99*
of `MouseCB` type, *97*
of `PositionCB` type, *98*
`invokeKeyCB` method, *53*
`invokeMouseCB` message, 24
`invokeMouseCB` method, *53*
`invokePositionCB` method, *53*

`KeyCB` module, *99*
`KeyCB` type, *99*
`KeyRec` type, *99*

light source, 5, *57*
ambient, *57*
point, *59*
spot, *61*
vector, *58*
`LightGO` module, *57*
`LightGO` type, *57*
`linChangeBy` method
of `RealPropSyncBeh` type, *86*
`linChangeTo` method
of `RealPropSyncBeh` type, *86*
line, *68*
`LineGO` module, *68*
`LineGO` type, *68*, 93
`LineType` type, *93*
`LineTypeProp` module, *93*
`LineTypePropAsyncBeh` type, *93*
`LineTypePropBeh` type, *93*
`LineTypePropConstBeh` type, *93*
`LineTypePropDepBeh` type, *93*
`LineTypePropName` type, *93*
`LineTypePropRequest` type, *93*
`LineTypePropSyncBeh` type, *93*
`LineTypePropVal` type, *93*
`LineTypeVal` type, 43
`linMoveBy` method
of `PointPropSyncBeh` type, *87*
`linMoveTo` method
of `PointPropSyncBeh` type, *87*

marker, *69*
`MarkerGO` module, *69*
`MarkerGO` type, *69*, 94
`MarkerType` type, *94*
`MarkerTypeProp` module, *94*

MarkerTypePropAsyncBeh type, *94*
MarkerTypePropBeh type, *94*
MarkerTypePropConstBeh type, *94*
MarkerTypePropDepBeh type, *94*
MarkerTypePropName type, *94*
MarkerTypePropRequest type, *94*
MarkerTypePropSyncBeh type, *94*
MarkerTypePropVal type, *94*
MarkerTypeVal type, 43
Matrix4 module, *45*
Matrix4 type, 43, *45*, 53, 91
Modifier type, *97*
modifiers field
  of KeyRec type, *99*
  of MouseRec type, *97*
  of PositionRec type, *98*
MouseCB module, *97*
MouseCB type, *97*
MouseRec type, *97*

Num type, 43

OrthoCameraGO module, *65*
OrthoCameraGO type, *65*

parallelopiped, *see* box
PerspCameraGO module, *66*
PerspCameraGO type, *66*
point
  constant value, 4
  type, *see* Point3
Point2 type, *97*
Point3 module, *44*
Point3 type, 43, *44*
PointLightGO module, *59*
PointLightGO type, *59*
PointProp module, *86*
PointPropAsyncBeh type, *87*
PointPropBeh type, *86*
PointPropConstBeh type, *87*
PointPropDepBeh type, *87*
PointPropName type, *86*
PointPropRequest type, *87*
PointPropSyncBeh type, *87*
PointPropVal type, *86*
PointVal type, 43
polygon, *72*
PolygonGO module, *72*
PolygonGO type, *72*
popKeyCB method, *53*
popMouseCB method, *53*
popPositionCB method, *53*
pos field
  of MouseRec type, *97*
  of PositionRec type, *98*
PositionCB module, *98*
PositionCB type, *98*
PositionRec type, *98*
projection
  orthographic, *65*
  perspective, *66*
Prop module, *80*
PropBeh type, *80*
property, 4, 7, 50, *80*
property mapping, 7
property name, 4, 7, 50, *80*
property value, 4, 7, 50, *80*
PropName type, *80*
PropRequest type, *81*
PropVal type, *80*
proxied object, 26, *46*
ProxiedObj module, *46*
ProxiedObj type, 26, *46*
pushKeyCB method, *53*
pushMouseCB method, *53*
pushPositionCB method, *53*

QuadMeshGO module, *79*
QuadMeshGO type, *79*

quadrilateral mesh, *79*

RasterMode type, *95*
RasterModeProp module, *95*
RasterModePropAsyncBeh type, *95*
RasterModePropBeh type, *95*
RasterModePropConstBeh type, *95*
RasterModePropDepBeh type, *95*
RasterModePropName type, *95*
RasterModePropRequest type, *95*
RasterModePropSyncBeh type, *95*
RasterModePropVal type, *95*
RasterModeVal type, 43
raw field, *46*
RealProp module, *85*
RealPropAsyncBeh type, *85*
RealPropBeh type, *85*
RealPropConstBeh type, *85*
RealPropDepBeh type, *85*
RealPropName type, *85*
RealPropRequest type, *85*
RealPropSyncBeh type, *85*
RealPropVal type, *85*
RealVal type, 43
remove method, *54*
removeKeyCB method, *53*
removeMouseCB method, *53*
removePositionCB method, *53*
request, 18–20, 29–32, 48, *81*, 83
    time interval of, 19–20, *81*
reset method
    of TransformPropConstBeh type, *91*
    of TransformPropSyncBeh type, *91*
rgbLinChangeTo method
    of ColorPropSyncBeh type, *89*
root, 4, *55*
RootGO module, *55*
RootGO type, *55*
rotateX method
    of TransformPropConstBeh type, *91*
    of TransformPropSyncBeh type, *92*
rotateY method
    of TransformPropConstBeh type, *91*
    of TransformPropSyncBeh type, *92*
rotateZ method
    of TransformPropConstBeh type, *91*
    of TransformPropSyncBeh type, *92*
rotation, *see* transformation, rotation

scale method
    of TransformPropConstBeh type, *91*
    of TransformPropSyncBeh type, *92*
scaling, *see* transformation, scaling
scene graph, 5, *50*
set method
    of BooleanPropConstBeh type, *83*
    of ColorPropConstBeh type, *88*
    of LineTypePropConstBeh type, *93*
    of MarkerTypePropConstBeh type, *94*
    of PointPropConstBeh type, *86*
    of RasterModePropConstBeh type, *95*
    of RealPropConstBeh type, *85*
    of ShadingPropConstBeh type, *96*
    of TransformPropConstBeh type, *91*
setBeh method, 17
    of BooleanPropVal type, *83*
    of ColorPropVal type, *88*
    of LineTypePropVal type, *93*
    of MarkerTypePropVal type, *94*
    of PointPropVal type, *86*
    of RasterModePropVal type, *95*
    of RealPropVal type, *85*
    of ShadingPropVal type, *96*
    of TransformPropVal type, *91*
setColorOfFacet method, 80
setName method, *52*

setProp method, *52*
Shading type, *96*
ShadingProp module, *96*
ShadingPropAsyncBeh type, *96*
ShadingPropBeh type, *96*
ShadingPropConstBeh type, *96*
ShadingPropDepBeh type, *96*
ShadingPropName type, *96*
ShadingPropRequest type, *96*
ShadingPropSyncBeh type, *96*
ShadingPropVal type, *96*
ShadingVal type, 43
shape hint, *72*, 79
Shape type, *72*, 79
sphere, 5, *75*
  center property of, 10
  radius property of, 10
SphereGO module, *75*
SphereGO type, *75*
SpotLightGO module, *61*
SpotLightGO type, *61*
start method, *81*
surface, 7, *70*
  color property of, 8
SurfaceGO module, *70*
SurfaceGO type, *70*, 93, 95, 96

torus, *78*
TorusGO module, *78*
TorusGO type, *78*
transformation
  rotation, *45*
  scaling, *45*
    uniform, *45*, 92
  translation, *45*
transformation matrix, *see* Matrix4
TransformProp module, *90*
TransformPropAsyncBeh type, *91*
TransformPropBeh type, *91*
TransformPropConstBeh type, *91*
TransformPropDepBeh type, *91*
TransformPropName type, *91*
TransformPropRequest type, *91*
TransformPropSyncBeh type, *91*
TransformPropVal type, *91*
TransformVal type, 43
translate method
  of TransformPropConstBeh type, *91*
  of TransformPropSyncBeh type, *92*
translation, *see* transformation, translation

unsetProp method, *52*

value method
  of BooleanPropRequest type, *84*
  of BooleanPropVal type, *82*
  of ColorPropRequest type, *88*
  of ColorPropVal type, *88*
  of LineTypePropRequest type, *93*
  of LineTypePropVal type, *93*
  of MarkerTypePropRequest type, *94*
  of MarkerTypePropVal type, *94*
  of PointPropRequest type, *86*
  of PointPropVal type, *86*
  of RasterModePropRequest type, *95*
  of RasterModePropVal type, *95*
  of RealPropRequest type, *85*
  of RealPropVal type, *85*
  of ShadingPropRequest type, *96*
  of ShadingPropVal type, *96*
  of TransformPropRequest type, *91*
  of TransformPropVal type, *91*
VectorLightGO module, *58*
VectorLightGO type, *58*

wentDown field, *99*

X'PEX'Base module, *47*
X'PEX'Base type, *47*