WavesWorld 3D Graphics Kit

Library:	/LocalDeveloper/Libraries/libWW3DKit.a
IB Palette:	/LocalDeveloper/Palettes/WW3DKit.palette
Header File Directory:	/LocalDeveloper/Headers/WavesWorld
Import:	WW3DKit.h

Introduction

The WavesWorld 3D Graphics Kit (WW3DKit) is a framework for building animatable 3D media. The framework includes a set of objects which conform to a small number of protocols, as well as a large set of examples. These objects are provided both as an InterfaceBuilder palette and as a linkable library. The WW3D Kit is designed to be used with the WWTCLKit, which allows sophisticated user interfaces to built quickly and easily for any 3D objects built using the WW3DKit.

The current version of WW3DKit is based on top of NeXT's 3DKit, and is therefore not potentially OpenStep compliant. This restriction will be removed in a future release, which will be based on a combination of OpenGL (for on-screen rendering), Pixar's PhotoRealistic RenderMan renderer and Larry Gritz's rendrib radiosity and raytracing renderer (for final output).

WW3D vs. NeXT's 3DKit

In contrast to NeXT's 3DKit, the WW3DKit doesn't require you to subclass any classes to start using it. In NeXT's 3DKit, you are expected to have some (set of) shape(s) that you want to draw on screen. You are expected to subclass N3DShape to draw the particular shape(s). This requires that you write some amount of Objective-C code to implement this N3DShape. In the WW3DKit's development model, you can immediately begin using it by manipulating some of the examples written in the scripting language WW3DKit uses; a simple language called **eve**. By using an ordinary text editor and dropping the eve code into a WW3DWell (an object provided in the WW3DKit; the only object on its IB palette), you can have the code compiled into a manipulatable hierarchical set of shapes. No compiling (of Obj-C code) required! Because of this, development proceeds much faster. Also, by using IB and the WWTCLKit palette, you can create specialized interfaces for a given 3D model by dropping a nib containing tcl UI objects directly into the WW3DWell, where they will be loaded and attached to the current model.

Once you have developed a reusable object in the scripting language, you can then recode the object as an Objective-C class, and lose none of the generality of the scripted version (although it will most likely be significantly faster to manipulate).

In sharp contrast to NeXT's 3DKit, the WW3DKit is built for animation, and it is built to take full advantage of the RenderMan interface. Developers of objects don't need to concern themselves with this; they get it for free by using the kit.

Models, Scenes and Shots

There are three terms that you need to understand in order to begin to grok what the WW3DKit is all about: *models*, *scenes*, and *shots*. We'll start off with brief high-level explanations, and then move directly into examples.

Models

In WavesWorld, a **model** refers to a description of the shape, shading and state of some character. There are two kinds of models; static and dynamic. Static models describe characters that never change their structure. They might change their position in the world, but their internal structure stays exactly the same for all points in time. Static models aren't very interesting, but you can build them in WavesWorld.

A dynamic model is one which has the *potential* to change over time. A dynamic model can say varying amounts about its potential for change. It can constrain this potential in a variety of ways, or it can leave itself open for a huge amount of internal change, or highly constrain what about itself is dynamic to a very narrow range. We'll see examples of both as we go on. Dynamic models are much more interesting, and used much more widely.

Scenes

Remember that a model just has, at best, the *potential* to change over time. A scene is a recording of a model expressing that potential in some way over the course of some amount of time. Another way to say this is that a scene is a sampling of components of the model over time. The sampling might be regular (every component every 1/30 of a second, say) or irregular (some parts once a second, some parts every 1/60, etc.). Some parts of the model might be sampled at a higher rate than others; there is no restriction on the granularity or regularity of the sampling. From a computer graphics point of view, a scene is completely abstract; it corresponds to the raw, sampled data of the simulation. If we want to see the data in a scene, i.e. images, we need to *shoot* it.

Shots

A shot is a regular sampling of a scene over time from a potentially varying position in space. A shot corresponds to a

rendering of the scene, where the shape, shading and state information of the scene is reduced to a 2D image sequence. Another way to think about the relationship between scenes and shots is that the scene is the simulation of some model, and a shot is a visualization of a given simulation. For a given model, you can have many different enactments of a simulation, and for a given simulation, you can have many different ways of visualizing it.

A Simple Static Example: a cylinder

Let's try and make this a bit more concrete with a simple example. Let's say we want to build a capped cylinder that's rolled a bit toward the viewer. The first thing we would do is create the model file. To do this, we'll create a file called **model.eve** with the following information:

```
set roll 45
set radius 1
set zMin -1
set zMax 2
set thetaMax 360

startShape example1
    Rotate $roll 1 0 0
    Disk $zMin $radius $thetaMax
    Cylinder $radius $zMin $zMax $thetaMax
    Disk $zMax $radius $thetaMax
endShape
```

If we then drop this file into a WW3DWell (either from inside of IB or by running the BuildaDudeII.app), we will create our cylinder:

360483_paste.tiff ¬

So what have we actually created? If we examine the contents of the WW3DWell using it's control panel (click the edge of the

well to bring it up, if you haven't already), we can see that our shape contains 4 objects:

paste.tiff ¬

Another way to visualize this is with the following diagram:

331205_paste.eps ¬

So what does this mean? Well, we have a WW3DShape that has an ordered set of renderable objects inside of it. When the WW3DShape is asked to render itself (say, by a WW3DCamera), it tells each of its renderable objects to draw themselves in turn. But what happened to the variables? In other words, what about $radius, $zMin, $zMax, etc. If we changed those variables, would they affect the model at all? The answer is no, and I'll explain why a bit later.

If we mouse around in the WW3DWell, we can move the cylinder around, but none of that information (the fact that the cylinder is here, then here, then there) is being recorded. Only the last position of the shape is available. If we saved out the scene at this point, the only difference between it and the original model would be that the variables that we used to define it were gone:

```
startShape example1 ;
    Rotate 45.0 1.0 0.0 0.0;
    Disk -1.0 1.0 360.0 ;
    Cylinder 1.0 -1.0 2.0 360.0 ;
    Disk 2.0 1.0 360.0 ;
endShape
```

In a real sense, the WW3DWell *compiles* eve code that you drop into. It doesn't compile it down to machine code, though; it just compiles it down to objects. When the shape hierarchy defined in the **model.eve** file was compiled, all references to the variables were resolved at compile time and assigned to the instance variables of the appropriate RIBRotate, RIBDisk, and RIBCylinder objects.

But what if we wanted the variables to be remembered? What if we wanted to change some variable and have the effects of changing it over time be recorded? How could we express that in eve?

A Simple Dynamic Example: a rolling cylinder

Let's try and make this a bit more concrete by extending our simple example. Let's say we want to build a simple cylinder that can rotate end over end. The first thing we would do is create the model file. To do this, we create a directory called **myCylinder.wwModel**, and put the **model.eve** file inside the directory. Note that the name ("model.eve") is actually very important; when a WW3DWell gets a wwModel file package dropped into it, it looks for a file named *model.eve* in the file package. You can use any prefix for the wwModel file package you like (i.e., foo.wwModel, yabba.wwModel, etc.), but you need to have a model.eve file inside the file wrapper. Anyway, let's make one change to the model.eve file. Wrap the command Rotate with an EveCmd; i.e.:

```
set roll 0
set radius 1
set zMin -1
set zMax 2
set thetaMax 360

startShape example1
    EveCmd {Rotate $roll 1 0 0}
    Disk $zMin $radius $thetaMax
    Cylinder $radius $zMin $zMax $thetaMax
    Disk $zMax $radius $thetaMax
endShape
```

If we then drop the file package **myCylinder.wwModel** into a WW3DWell (either from inside of IB or by running BuildaDudeII.app), we will create our cylinder, which looks exactly like the first example:

728141_paste.tiff ¬

So what have we created? If we want to look at the WW3DKit objects that got created:

51744_paste.eps ¬

So what does this mean? Well, we have a WW3DShape that has an ordered set of renderable objects inside of it. When the WW3DShape is asked to render itself (say, by a WW3DCamera), it tells each of its renderable objects to draw themselves. The difference is that now one of those renderable objects is an EveCommand. EveCommands are special, because they have a notion of time.

Actually, when I said that the objects were asked to render themselves, I lied. The objects are actually asked to render themselves **over some period of time**, i.e. from time 1.0 seconds to time 1.033 seconds. In the limit, the objects are asked to render themselves at an instant of time, where the interval between the start and the end of the interval is zero. In most cases, the objects only know how to render themselves one way, no matter what time it is. EveCommands (and EveProcs) are special, though. EveCommands know two things: their symbolic representation and their time-stamped sampled (compiled) representation. When asked to render themselves over a given interval, they search their list of samples looking for a sample at the beginning of the time interval and one at the end of the time interval, and then tell each of them to render themselves. For example, if we rendered our cylinder now from time 1.0 to time 2.0, we'd get the following image:

cylinder.tiff ¬

This is because the EveCommand right now only has one sampled representation of itself; the RIBRotate object corresponding to time 0. If at time 1.5, though, the value of $roll suddenly changed, the EveCommand would get sent a message to resample itself. It resamples itself by having sending its symbolic representation to its interp outlet , which compiles the representation into a new object representing the current state of the EveCommand. Let's say that the value of roll changed from 45 to 15:

paste.eps ¬

Now suppose we again rendered the scene starting at time 1.0 and ending at time 2.0.

Now the rendered image looks like this:
motionBlurredCylinder.tiff ¬
If you look closely, you'll see that the image is a little smeared and blurry; nowhere near as sharp as the first image. Why is this? Well, when the shape was asked to render itself for this frame, the EveCommand in the shape had two samples of itself over the interval that is was being asked to render itself, so it gave both of them to the renderer. This is a phenomenon that occurs all the time with real cameras - it's called ***motion blur*****.** What happened was that the shutter was open while some part of the scene was changing. RenderMan allows you to describe such changes, and PhotoRealistic RenderMan and Larry Gritz's rendrib (as opposed to QuickRenderMan used by the Window Server) *can* render such scenes correctly. If we asked the WW3DWell to dump out our scene now, what would we see? Well, if you try it, you'll get a file something like this:

```
set roll {55};
startShape {dynamicExample} ;
    EveCmd {Rotate $roll 1 0 0} {\
                {0.0 { {{} 1.0 {Rotate 45.0 1.0 0.0 0.0;}} }} \
                {1.0 { {{WWSampleList} 1.0 {Rotate 25.0 1.0 0.0 0.0;}} }} \
                {1.500000 { {{} 1.0 {Rotate 15.0 1.0 0.0 0.0;}} }} \
               };
    Disk -1.0 1.0 360.0 ;
    Cylinder 1.0 -1.0 2.0 360.0 ;
    Disk 2.0 1.0 360.0 ;

endShape
```

This is very similar to our original model, with one obvious exception: The EveCmd line has an additional argument; a list containing three elements:

- {0.0 { {{} 1.0 {Rotate 45.0 1.0 0.0 0.0;}} }}
- {1.0 { {{WWSampleList} 1.0 {Rotate 25.0 1.0 0.0 0.0;}} }}
- {1.500000 { {{} 1.0 {Rotate 15.0 1.0 0.0 0.0;}} }}

What are they? Again, remember what the WW3DWell does when you drop a model into it: it compiles it into objects. Most commands (startShape, Disk, Cylinder, etc.) map directly to an underlying representation, and therefore WW3DWell only needs to map the variables in the command's arguments to their current values. Certain commands (EveCommand, EveProc), though, need to maintain both a symbolic representation of themselves as well a compiled representation that has the values of the command's arguments mapped to their current value. The crucial difference is that these commands (The ***animatable*** commands) have a notion of time. They know **when** they compile themselves. But how do they know when to initiate a given compilation? And how do they know what time it is when they compile themselves? The second question is simple to answer Ð they just ask the sceneClock object, of which there is only one in the WW3DWell. We'll talk more about the sceneClock later.

But how do they know when to resample and recompile themselves? That's a bit more complex. When the animatable command is first instantiated, it hands its symbolic representation to its eveParser and asks it to perform a ***closure*** on it. The eveParser takes the symbolic description and identifies all the variable expressions in it. It then determines which of those variables are local in scope; i.e. they might disappear immediately after this command in the model. It evaluates each of these variables to their current expression and replaces that in the symbolic description. The other variables, the global ones, are assumed to be persistent over the model. For each global variable in the expression, the eveParser sets up a variable trace on it. Each time any of these variables has a value written to it (even if it's the same value as the previous one) this animatable command will get sent a message to ***resample*** itself. When the animatable command resamples itself, it recompiles itself, ask the sceneClock what time it is, wraps both bits of information up in a WWSample object, and stores it in its sample list. Whenever the EveCommand resamples itself, there is, somewhere, an object that generated the events that caused it to resample itself. Sometimes, it might be a particular agent, sometimes it might the "hand of god" (i.e. a user in direct manipulation), perhaps it might be an ongoing physical simulation in the virtual environment, or it might even be the underlying sample list itself.

So now let's look at that EveCmd expression again that got saved out when we dumped out our scene file. We should now be able to see that the expression:

```
EveCmd {Rotate $roll 1 0 0} {\
        {0.0 { {{} 1.0 {Rotate 45.0 1.0 0.0 0.0;}} }} \
        {1.0 { {{WWSampleList} 1.0 {Rotate 25.0 1.0 0.0 0.0;}} }} \
        {1.500000 { {{} 1.0 {Rotate 15.0 1.0 0.0 0.0;}} }} \
       };
```

can be read as

"Define an EveCommand object which has the symbolic expression of 'Rotate $roll 1 0 0'. We have three samples of this expression already; at time equals 0, it compiled to the command 'Rotate 45 1 0 0' at the behest of an unnamed signal generator, at time 1.0 it compiled to command 'Rotate 25 1 0 0' at the behest of the WWSampleList, and finally at time 1.5 it compiled to the command 'Rotate 15 1 0 0' at the behest of an unnamed signal generator."

<<more>>...

Up till now though, I haven't mentioned how the variable $roll was changed, or how time happened to move forward from 0 to 1, through 1.5 and then to 2, and I haven't told you how to make a picture. All in good time, but if you were wondering about these things, you're thinking about the right issues...

Manipulating variables

One way to manipulate a variable attached to a model in a scene in a WW3DWell is by typing eve code directly at the tcl interp inside the WW3DWell. You do this by using the WW3DWell's Control Panel, which can be brought up by clicking the edge of the WW3DWell, much like you click the edge of a NXColorWell to bring up its control panel:

359248_paste.tiff ¬
Using this technique, we can send any valid cmd we want to the tcl interp. But this seems rather clumsy. What we'd rather be able to do is to attach (say) a slider to the variable, which we could then use to both manipulate the variable $roll directly, as well as visualize the value and thus be able to monitor if it changes (say due to some other process manipulating it). This is where the WWTCLKit IB palette comes in. All we have to do is fire up IB and load in the WWTCLKit palette.

- Open up a new nib file that has a window in it and save the nib file to disk.
- Make the File's Owner of the nib file be a WWTCLInterp and connect the Window up to the **controlPanel** outlet of the File's Owner.
- Make the Window "visible at launch time".
- Drag a WWSlider off the WWTCLKitWWTCLKit palette and place it on the Window.
  If we then inspect the WWSlider, we can do the following:

102239_paste.tiff ¬ we then alt-click: 288114_paste.tiff ¬

If we now drop that nib file directly into the WW3DWell (we could also load it from inside a model.eve file by using the "loadControlPanel" command), we can control the value of $roll by manipulating the slider. Unfortunately, we can't accurately set the value, since we don't have an easy way of seeing exactly what the number is. If we go back into IB, though, and open up the nib file, we can drag out a WWTextField and set its values the same as we did for the WWSlider. We then drop that nib file in, and voila! we can now manipulate and visualize the variable in our dynamic model. One thing to note here is the fact that all the information we need (after we hooked the Window up to the controlPanel outlet) is available on a given object's inspector - unlike traditional NEXTSTEP programming, there is no need to attach each additional UI element as the outlet of the File's owner - the WWTCLInterp takes care of that automagically when the nib file is opened.

Manipulating time I

Well, we now have some notion of how to manipulate the dynamic components of a model, but we still don't know how time operates. What time was it when we were doing all of our former activity? Was time standing still? Was it rushing forward at some multiple of real-time, was it being triggered by some external event, or was it moving at some fraction of wall time? Well, the answer in our previous example is that it was standing still, but it's possible to make time behave in any of those other ways.

If you haven't already, bring up the Scene Clock Controls.

679084_paste.tiff ¬704701_paste.tiff ¬

There is just one scene clock in a given WW3DWell. Whenever any part of an expression that an EveCommand depends on changes, the EveCommand resamples itself with respect to the current timestamp it receives from the scene clock. You can change the scene clock by manipulating the "Scene Clock Controls".

To make our motion blurred cylinder, we move the current time to be 1.5 seconds, and then change the value of $roll to 15. We can change the value of $roll by manipulating the slider, typing in the text field, or by typing directly to the Tcl Interp by using the WW3DWell Control Panel.

If we want to take a still picture, we switch over to the Camera panel and bring up the PhotoRealistic RenderMan controls.

628473_paste.tiff ¬

Since we want the image to be shot starting at time 1 second and lasting until 2 seconds, we make sure the currentTime on the Scene Clock Controls says 1 second:

122422_paste.tiff ¬

and that the exposure length on the prman controls says 1 second, and then we snap the picture by pressing the "render it in the foreground" button.

Manipulating time II

Well, we now have some notion of how to manipulate time manually by manipulating the scene clock by using the Scene Clock

Controls, but how could we do programmatically? Let's write a little tcl proc that will rotate the cylinder end over end over the course of 1 second.

```
proc rotateIt {} {

    global   roll scene;

    set rollIncr [expr {180./$scene(ticksPerSecond)}]
    for {set i 0} {$i <= $scene(ticksPerSecond)} {incr i} {
        set roll [expr {$i * $rollIncr}];
        synchWithScene
    }
}
```

Save this out in our model file and then add a button from the WWTCLKit palette to tell it to run the *rotateIt* proc. If we now drop this in the WW3DWell and press the button, we'll see several things start happening at once. First of all, the cylinder starts rotating. As it rotates, we notice that the text field and the slider both change value. Also, and more interestingly, the current time on the Scene Clock Controls moves forward. How did that happen? Well, the "synchWithScene" command blocks until the scene clock increments its next step. In the full-blown WavesWorld system, this actually means something (where we have multiple processes on multiple machines, all manipulating the scene), but here, it means that "synchWithScene" increments the sceneClock by one sample.

The other thing to note is the use of the global variable "scene(ticksPerSecond)". This variable tells the eve code how coarsely or finely time is currently being sampled. Since "synchWithScene" will block until the next sample has passed, it doesn't make any sense to provide more samples of the model over the course of a second than "$scene(ticksPerSecond)".

Now that we've got a scene of our model, we can scan around through time using the scene clock controls. Use the rewind, fast forward, play, and step buttons to see what I mean.

If we wanted, we could dump out the "scene" file by using the 328851_paste.tiff ¬ menu. Do that now, and take a look at the file.

But now let's make a little movie. Since we have a scene, we can shoot it. To do that, turn to the movie camera section of the WW3DWell Control Panel:

211271_paste.tiff ¬
Using the Dump menu, we dump out the shot. Note that we said that the shot starts at time 0, goes to time 1, is shot at 30 frames/second, and that we want the shot to be similar to a film camera, i.e. the shutter is open for half the possible time. A shot is dumped out as a single RIB file containing all the frames of the shot. To render it, just bring up a Terminal window and type "prman foo.rib", or whatever the name you gave the rib file. Usually I move the foo.rib file into a directory called "foo.anim" where I then cd to and type "prman foo.rib", since foo.rib will generate a numbered set of frames called foo.0.tiff, foo.1.tiff, etc. A given "foo.anim" directory can be dropped onto the ControlPanel of a WWSimpleMovieView and viewed, or looked at with Movie.app or TiffanyII.app.

Manipulating time III

Well, we now have some notion of how to manipulate time by writing some tcl code and looping over it, but that's really a rather unfriendly way of doing things. While that loop is running, we can't do anything; we can change our location, we can't examine the model, we have to wait until the loop is over. So how can we get around this?

Once again, we turn to the WWTCLKit palette, this time to look at one of the non-UI object on the palette: WWTTTimer:
WWTTTimer.tiff ¬

Open up our nib file that has the slider and the text field again and drop a WWTTTimer into it. Notice that the WWTTTimer is a subclass of Object, not View, so you have to drop it into the File's Window, not onto a Window.

So remember what the task at hand is. We'd like to get our model to change its roll variable over time, but do it in such a way

that we could be interacting with it at the same; changing our viewpoint, etc.

After we drop our WWTTimer object into the File's Window, we need to connect up our Timer's interp outlet to the File's Owner: TimerIBScreenShot1.tiff ¬

When we inspect the Timer, we set its period to be the minimum (type in 0, it will set it to whatever the smallest useful value is) and we make it a "conditional lifespan" timer.

489606_paste.tiff ¬

We now have to inspect the tcl code associated with the timer. Press the "show tcl code" button to inspect it.

NOTE: the UI for this currently sucks, and is totally against all UI guidelines for IB, but...

Remember that we're trying to mimic the functionality of the rotateIt proc we wrote before:

```
proc rotateIt {} {

    global   roll scene;

    set rollIncr [expr {180./$scene(ticksPerSecond)}]
    for {set i 0} {$i <= $scene(ticksPerSecond)} {incr i} {
        set roll [expr {$i * $rollIncr}];
        synchWithScene
    }
}
```

For the timer to work, we'll need to unroll the loop we made and think of it more like a distributed while loop: set up some initial

state, define the end condition for the loop, define the activity to be done each time through, and define the clean up activity after the loop finishes.

For our initialization, we need to set up the amount we'll increment the roll value and our initial Roll value:

```
set rollIncr [expr {180./$scene(ticksPerSecond)}]
set i    0
```

Our condition for killing the timer will be when the value of i is greater than the number of ticks/second in the scene:

```
$i <= $scene(ticksPerSecond)
```

As for what we want to do each time the Timer executes, we do the body of the original loop:

```
set roll [expr {$i * $rollIncr}]
synchWithScene
incr i
```

Note: because the UI is so lame, you need to be clicking "OK" a lot (don't forget to save too, sigh)....

One way to start up the timer is to just select "auto-start"; when the nib is loaded the Timer will automatically start up. If we want more control than that, though, we can drag out a few buttons for play, pause, and start. I use the ones Keith did for the CDPlayer.app. One neat thing about WWTTTimers is that they have outlets for these three buttons and will take care of automatically reflect the state of the timer. Make sure you hook up the action msgs of the three buttons to send the appropriate one to the Timer, as well as hooking them up to the appropriate outlet of the WWTTTimer. See the example nib sliderTextTimer.nib in the directory with this document.

If we drop this nib into a WW3DWell containing our dynamic model, we'll see that (if we made it an autostart WWTTTimer) the

cylinder immediately starts rotating, and we see this reflected in the well, in the fact that the play button for the WWTTTimer is now selected, and that the slider and WWTextField are reflecting the value of the roll variable.

Feel free to pause or stop the timer. Feel free to mouse around in the WW3DWell, or manipulate the controls in the WW3DWell's Control Panel. Cool, huh?

<<More to come>>

Building more sophisticated models

One thing that you might want to do with our rotating cylinder is to make it seem solid, and not just a surface representation. One way we could do this is to draw a pair of Patches on the inner part of the open cylinder, like this:

226759_paste.tiff ¬cappedAndClosedCylinder.tiff ¬
Well, this looks find when the cylinder is open, but it doesn't make much sense to draw those two patches when the cylinder is completely closed. Could we define our model in such a way that when it was closed (i.e. thetaMax == 360) or when it was fully open (i.e. thetaMax == 0) it wouldn't draw the patches, but when it was partially open it would draw them? Well, of course we can, or I wouldn't have mentioned it...

What we want to do is draw two bilinear patches (we could use polygons, or NURBS, or a patch mesh, but patches are easy) in the open section to close it off. We only want to draw them when thetaMax is not evenly divisible by 360.

Up until now, whenever we wanted something in a model to be dynamic and vary over time, we placed the single command that we wanted to vary in an EveCommand. Unfortunately, we now want several commands to vary, and we don't want to just vary their parameters; we want to be able to not render the command at all sometimes. Currently, WavesWorld provides a way to wrap up a proc inside a new command called "EveProc". An EveProc functions exactly the same as an EveCmd (performs a closure on its arguments, sets up traces on it, gets sent resample msgs whenever any of them changes, etc.) except that

instead of taking an atomic Renderable command, it takes a somewhat arbitrary tcl proc. The only restrictions (some of which may be removed in the future) is that there can be no EveCmd, EveProc, or startShape/endShape cmds inside the proc. Also, all global variables that we want the EveProc to be reevaluated ifthey change should be at the top level; i.e. declared as arguments to the proc. If you think about what's going on, it should be obvious why these restrictions are in place, at least for the first pass.

```
proc drawInside {thetaMax zMin zMax radius innerX innerY} {
   if {int($thetaMax) && [expr int($thetaMax)] % 360} \
   {   Patch bilinear P "0 0 $zMax   \
                                        $radius 0 $zMax   \
                                        0 0 $zMin            \
                                       $radius 0 $zMin   \
                                     ";
       Patch bilinear P "0 0 $zMax   \
                                       $innerX $innerY $zMax   \
                                       0 0 $zMin            \
                                       $innerX $innerY $zMin   \
                                    ";
   } {}
}
```

This proc will draw the inside patches in exactly the fashion I've described. Now let's flesh out the rest of the model.

```
set cylinder(radius) 1.0
set cylinder(zMin) [expr {-1 * $cylinder(radius)}]
set cylinder(zMax) $cylinder(radius)
set cylinder(thetaMax) 360.0
```

```
set cylinder(roll) 45.

EveCmd {set cylinder(innerX) [expr {$cylinder(radius) * cos(radians($cylinder(thetaMax)))}]}
EveCmd {set cylinder(innerY) [expr {$cylinder(radius) * sin(radians($cylinder(thetaMax)))}]}

startShape aCylinder
   EveCmd {Rotate $cylinder(roll) 1 0   0 }
   EveCmd {Disk $cylinder(zMin) $cylinder(radius) $cylinder(thetaMax)}
   EveProc {drawInside $cylinder(thetaMax) $cylinder(zMin) $cylinder(zMax) $cylinder(radius) $cylinder(innerX)
$cylinder(innerY)}
   EveCmd {Cylinder $cylinder(radius) $cylinder(zMin) $cylinder(zMax) $cylinder(thetaMax)}
   EveCmd {Disk $cylinder(zMax) $cylinder(radius) $cylinder(thetaMax)}
endShape
```

<<more to come>>...

Zooming in on time: I

Let's look at our original cylinder again - remember, the one that's scene file looks like this:

```
tartShape dynamicExample ;
    EveCmd {Rotate $roll 1 0 0} {\
                    {0.0 {Rotate 45.0 1.0 0.0 0.0;}} \
                    {1.5 {Rotate 55.0 1.0 0.0 0.0;}} \
```

```
                };
    Disk -1.0 1.0 360.0 ;
    Cylinder 1.0 -1.0 2.0 360.0 ;
    Disk 2.0 1.0 360.0 ;
endShape
```

Again, remember how to read the EveCmd section of the scene:

"Define an EveCommand object which has the symbolic expression of 'Rotate $roll 1 0 0'. We have two samples of this expression already; at time equals 0, it compiled to the command 'Rotate 45 1 0 0', and at time 1.5 it compiled to the command 'Rotate 55 1 0 0'. We'll add these two samples to the animatable command's list of samples."

Let's think about what that might mean. At time 0.0 (all scenes start at 0.0), we had a RIBRotate object that had an angle value of "45" and a vector value of "1 0 0". Then at time 1.5, we had a new RIBRotate object, which had an angle value of 55 and the same vector value of "1 0 0". So here's the question: what is the value of that EveCmd at time .75? In other words, do we assume that the point samples of the object correctly represent the underlying signal that the model generated in this scene? If so, then it's fair to assume that the rotation angle at time .75 would be exactly the same as it was at the last sample: 45.
pointSample.eps ¬

Or should we expect to be able to linearly interpolate values between samples, thereby giving us a value of 50?

lerp.eps ¬

You can make arguments for both sides, and you can even make other arguments, like you should consider the samples as control points on a Catmull-Rom spline and interpolate values that way. For now, though, the WW3DKit just supports point samples and linear interpolation (called "lerping").

<<more here>>

Zooming in on time: II

So that's what things look like when we look at an interval containing some number of samples. But what if we zoom into the space of a single sample? Where did it come from? Was it some combination of a set of samples that all came in at that point in time, or what? Let's review what happens in the WW3DKit at the point 1.5 in time for our Cylinder:

some agent (in this case, the user manipulating the slider) changes the value of the variable "roll" to 66.43. This causes all objects that have asked to be notified about changes in that variable to be sent a "resample" msg. In this case, there were three objects that asked to be notified: the WWSlider that was just used to manipulate the value, the WWTextField sitting next to it, and finally the EveCmd in the model. The WWTCLKit's run-time system sends the msgs to the UI objects and they update themselves and redisplay. The EveCmd gets the message and sends its symbolic expression ("Rotate $roll 1 0 0") to its eveParser to compile in the current context. The WWEveParser compiles this into a single RIBRotate object which has an angle of 66.43 about a vector of "1 0 0". The EveCmd time stamps this new object (creating a WWSample object to hold the new sample data and the timestamp) and hands it to its WWSampleList. The WWSampleList examines the new WWSample, looking at its timestamp. It then examines its own list of WWSamples, and determines that it doesn't have any WWSamples within epsilon of the time on the timestamp, so it places it in order in its list, right after the WWSample time stamped 0.0.

Now the user types "55" in the WWTextField. This again causes the 3 "resample" messages to go out, and things proceed as before, with the exception of when the WWSampleList receives its new samples. This WWSample is also time stamped 1.5, just like the previous one the WWSampleList inserted in itself. So what to do with this new sample?

As before, we have a few options. We could throw away the previous sample and put the new one in it's place. We could linearly interpolate between the two samples, throw both of them away and store the newly interpolated one. We could weight the interpolation, so that the sample that was there first has more weight than the incoming one, or vice versa. We could just store both samples and mix them on the fly. What to do, what to do...