3 Modeling in WavesWorld

Introduction

> *At the beginning of our lesson I told Tortsov, the Director of our school and theatre, that I could comprehend with my mind the process of planting and training with myself the elements necessary to create character, but that it was still unclear to me how to achieve the building of that character in physical terms. Because, if you do not use your body, your voice, a manner of speaking, walking, moving, if you do not find a form of characterization which corresponds to the image, you probably cannot convey to others its inner, living spirit.*
>
> *"Yes", agreed Tortsov, "without an external form neither your inner characterization nor the spirit of your image will reach the public. The external characterization explains and illustrates and thereby conveys to your spectators the inner pattern of your part."*
>
> *"That's it!" Paul and I exclaimed.*
>
> *"But how do we achieve that external, physical characterization?" I asked.*
> **(Stanislavski49)**

In the beginning of Chapter 2, we saw how important the actor's own body, costume and props is to the successful creation of a character. In the digital domain, though, *everything* must be constructed, including the actor's body. We clearly need to consider the representation of which our character's form will be composed before we address the issue of its behavior, i.e. how it changes over time.

This chapter describes the approach to modeling that has been developed for this thesis. It is an object-oriented approach that allows modeling three-dimensional character parts (amenable to photorealistic rendering) and a set of tools for facilitating their debugging and encapsulation for later reuse. We have implemented this as two large groups of object classes (over 150 classes in the WWTCLKit & WW3DKit), that I collectively refer to as the **eve object framework**.

This framework addresses a myriad of interconnected issues that must be resolved if we are going to have photorealistic three-dimensional animated characters, composed of cooperating and competing agents, situated in a dynamic environment. I know of no other published approach that addresses all of these issues together, and I consider this explication of the issues and our representative implementation as one of the central

contributions of this thesis work.

I refer to our solution as an *object framework*, which may be a confusing term to some readers. In our case, **eve** has several faces: it refers to both a modeling language embedded in a computational language and an extensible set of object classes which conform to a small set of protocols.

I have developed an object compiler which transcodes from the modeling language to a database of compiled objects (written in Objective-C). There is a run-time system that maintains dependencies between variables accessed via the computational language and the underlying objects. In addition, we have developed a set of GUI tools to both visualize and manipulate the objects in the database. This combination of a dynamic language and a fast object compiler, coupled with an interactive GUI toolkit for manipulation and visualization of objects and their interrelationships, allows us an unprecedented level of flexibility and power in building and debugging the parts that compose a character.

Because of the object-oriented nature of the framework, appropriately sampled scenes constitute a powerful recording mechanism for allowing the events of a character's interaction with its environment to be recorded, allowing arbitrary spatial and temporal resolution playback of the experience. These can be played back with no simulation system present; just a modest run-time system and a graphics subsystem for rendering are necessary.

In this chapter I first describe the requirements for the representation and then present several simple examples which build on each other to point out some of the capabilities of our approach. I then describe our approach in some detail; discussing the process of debugging and the issues of reusability (i.e. the “packaging problem”), which is a central issue in using this system for collaborative construction of characters. Finally, I'll touch on some of the interesting and novel capabilities of this framework, using more complex examples for illustration, and finally summarize what we’ve seen thus far.

Representation Requirements

> *The act of the mind, wherein it exerts its power over simple ideas, are chiefly these three:*
>
> *1. Combining several simple ideas into one compound one, and thus all complex ideas are made.*
>
> *2. The second is bringing two ideas, whether simple or complex, together, and setting them by one another so as to take a view of them at once, without uniting them into one, by which it gets all its ideas of relations.*
>
> *3. The third is separating them from all other ideas that accompany them in their real existence: this is called abstraction, and thus all general ideas are made.* **(Locke1690)**

This section briefly discusses the essential requirements for a representation for photorealistic three-dimensional data over time, with special consideration given to our problem of constructing three-dimensional semi-autonomous animated characters. For the larger, general issues involved in describing photorealistic scenes, I would recommend **Upstill89** or **Pixar89**.

Comprehensive CG Vocabulary

Since our representation is intended to be understood by a computer graphics rendering system, it is vital that it contain a comprehensive three-dimensional computer graphics vocabulary. This includes support for:

- hierarchical transformations
- polygons
- curved surfaces
- a materials description language

The first three requirements are obvious, but the fourth may be unfamiliar to some readers.

Encouraging Encapsulation and Abstraction for Materials

In the computer graphics domain, the more sophisticated modeling and rendering systems make a careful distinction between **shape** and **shading**. *Shape* refers to the geometry of objects. This includes what geometric primitives they're composed of (spheres, polygons, patch meshes, etc.), as well as what geometric transformations are acting on them (scale, skew, translate, rotate, etc.). *Shading* refers to the subtle surface qualities of an object due to the material properties of the objects in the scene, local illumination effects, and global illumination effects.

Being able to accurately and expressively describe the properties of light sources and the material properties of objects is one of the gulfs that separate limited CG systems from powerful production systems capable of rendering photorealistic images. From a character constructionist's point of view, a shading (i.e. materials description) language is important because we want to be able to "package" a given material and give it understandable parameters that can be manipulated and measured. For example, if a robotic character is composed of a certain kind of pitted, rusted metal, it would be nice if we were able to describe this in whatever complex detail we needed in order to render this material, and then be able to abstract it as some function with a set of parameters. This abstraction is important to a character builder for several reasons.

First of all, we want to be able to package up and abstract portions of our character so that we're not inundated with details during the iterative construction phases. More importantly, though, is the vital role this will play in making characters and their environment *perceivable* to other characters and to the autonomous behavioral components (i.e. the agents) that comprise the character. All of these need to be able to **measure** and **manipulate** parts of the character and the environment, and only by having descriptions available at the appropriate level of abstraction can this be achieved. Otherwise, we are left with having all of our characters continually forced to work from first principles. We shouldn't *have* to solve the vision problem for the character to know it's on a tile floor, and it should be able to easily determine the size and position of the tiles by actively perceiving that from the representation, and not be reduced to only working from some static database (i.e. tiles are usually one foot square).

Such a materials description language is vital to being able to build up a useful knowledge base of object properties in a given environment. By constructing and experimenting with a variety of ways of describing materials appropriately, we can build up abstractions that will allow us to build up stereotypical descriptions that can be reused among characters and environments.

Continuous over Time

We want our representation to be continuous over time. This means that we don't want our representation to consist of a set of disconnected samples (i.e. frames) over time, nor do we *really* want them to be samples at all. Realistically, though, because of the discrete nature of animation, we will need to deal with sampling issues. We may have

multiple samples of what a portion of our model is at a given point in time, and we also may have some set of samples over time that represent some component of our model. If so, we'll need reasonable mechanisms for reconstructing the underlying signals that the samples represent. We also may have a disparate set of samples at a given point in time that represent some set of opinions of the model. In the same way that we need to be able reconstruct a signal from a set of samples over time, we should have mechanism for reconstructing a single sample at a point in time from some set of weighted samples from different sources at that point in time.

Composable

In the same way that we want to be able to "package up" material properties of character parts and props, we also want to be able to package the shape and state information that comprise the parts of a character. This implies that we have some core set of objects and some rules of composition, i.e. building a new class by composing together other classes. A key issue here is to come up with a core set of classes that can be easily and powerfully composed, whereby newly defined classes have all the flexibility of the core classes. This implies that there are some set of messages that all of these core objects respond to, and that newly composed objects also respond to these messages. Given the fact that we're trying to build a three-dimensional computer graphics representation, these messages would include information for asking the objects to render themselves over some span of time, asking them what their bounding box is over some span of time, whether they push or pop the current transformation stack, etc.

"The nice thing about standards, there's so many to choose from"

Given the amount of work that's gone on in the last twenty year in computer graphics, It clearly makes little sense to try and build such a representation from scratch. But if we're to use and build on an existing standard, which ones are appropriate to choose from? When this work was initially begun in the late 80s, there were a large number of 3D graphics APIs available: GL from SGI, Dore from Ardent, the RenderMan® Interface from Pixar, the in-progress ANSI PHIGS, RenderMatic (an in-house Media Lab rendering library developed primarily by Dave Chen), etc. There were trade-offs involved with each; some scaled well, some ran very fast, some gave high quality results, some were procedural, some were object-oriented, some I could get source code to, etc., but

none met all the criteria I had.

Building on the RenderMan® Interface

Of these, though, the RenderMan® Interface was the closest in spirit; it attempted to define an interface between a modeler and a renderer, where the 3D scene (a 3D model viewed by a camera that had a shutter speed of some duration) was described in great enough detail to facilitate photorealistic rendering. The RenderMan® Interface had several unique things going for it that were especially appealing to me: a very powerful shading language and a set of quadric (sphere, cone, torus, hyperboloid, parabola, disk) and other curved surface primitives (uniform and nonuniform patch meshes, with trim curves). I discussed the need for a shading language earlier in this section, but the significance of having curved surface primitives may not be as obvious.

One of the underlying concerns in WavesWorld is the fact that parts of the testbed may be distributed over a network of computational resources. If so, the size of models built out of our representation is an important factor. Curved surface primitives allow us to very compactly and exactly specify model parts, as opposed to polygons, which are usually a much cruder approximation. Polygons can be useful sometimes (i.e. to represent flat things with flat edges), but there are few (if any) good arguments for having *only* polygonal primitives at the modeling level.

The initial versions of software I wrote on the way to WavesWorld used several of these graphics APIs, with the current (and final) version using the RenderMan® Interface as a basis. The reason for this is that there are only a few downsides to using the RenderMan® Interface. The first problem is that it's just a specification, and if you want to use it as a basis for your modeling representation, you'll need to write a lot of code. There are several high quality RenderMan® compliant renderers available, but there are no freely available modeling system that use it. Because of this, I needed to develop such a system.

The second problem is that the RenderMan® Interface is a procedural interface, it was necessarily to design and implement an appropriate object-oriented layer above it. This was a non-trivial design and engineering task; I spent over two years on this issue. There are a myriad of different ways to build such an object-oriented toolkit atop the RenderMan® Interface; NeXT's 3DKit® is one such framework. NeXT's 3DKit® does not deal with the critical issue of time, which the eve object framework that I developed for this dissertation does.

Why not use OpenInventor® (or OpenGL®)?

One question I'm often asked is why I didn't use OpenGL®, or its corresponding object-oriented toolkit, OpenInventor® (both from SGI). There are several reasons. The simplest is a matter of maturity: both OpenGL® and OpenInventor® are still young software products, with the 2.0 release this past year finally addressing some of the serious extensibility and performance issues of the 1.0 release. The RenderMan® Interface, on the other hand, was developed and released by Pixar in the late 80s, and has stayed stable since its 3.1 release in 1989. RenderMan® Interface compliant renderers have existed since that time, with at least one renderer (and in many cases two or more) available on all major software platforms, from PCs and Macintoshes, through workstations, on up to Crays and CM5s. OpenGL® and OpenInventor® is now becoming available on an equivalent range of machines, but this is a very recent (last year or two) development, with much of the implementations still in beta.

As opposed to the RenderMan® Interface, which has a complete and general set of quadric primitives, OpenInventor® as a somewhat ad-hoc set of curved surface primitives. Both have trimmed NURBS (non-uniform rational b-spline) surfaces, which are a very general but difficult to specify primitive (i.e. a modeling program can generate a NURBS surface, but one rarely writes one by hand). This means a fair amount of work has to go into specifying a higher level set of primitives such as partially swept out spheres, cones, cylinders, hyperboloids, tori, etc. that can be used directly by a modeler. None of which is terribly difficult, but tedious (and time-consuming) nonetheless.

In short, in the areas that the RenderMan® Interface did not deal with (such as time over the course of a scene rather than just a frame, or composability and encapsulation of primitives), neither did OpenGL® or OpenInventor®. In the areas that both OpenGL®/ OpenInventor® and the RenderMan® Interface addressed, the RenderMan® Interface addressed more elegantly (i.e. intuitive curved surface primitives and materials description). Also, the RenderMan® Interface scales well from interactive rendering all the way up to the most complex and demanding photorealistic rendering algorithms, such as radiosity (**Gritz95**).

It's also interesting to note that it's relatively straightforward to implement an OpenGL®-based renderer that uses the RenderMan® Interface. One of the preview renderers I use, **rgl** (part of the **Blue Moon Rendering Tools** written by Larry Gritz) uses OpenGL®'s predecessor, GL, to do all of its rendering. An OpenGL® version, while not

trivial, is reasonably straightforward (**Gritz95B**).

On the other hand, adding the functionality needed by WavesWorld to OpenInventor® would probably not be straightforward, largely due to deficiencies in C++, OpenInventor®'s implementation language (**Bell95**). It would be difficult to extend the base classes in Inventor to respond to the messages need in WavesWorld, which means that the entire class hierarchy would have to be subclassed before work could begin. Also, since C++ has no run-time system to depend on, the run-time requirements of WavesWorld would have to be served by OpenInventor's run-time system, which may or may not suffice.

In short, the only thing that the combination of OpenGL®/OpenInventor® did have going for it was that it was a procedural interface married with an object-oriented toolkit, while the RenderMan® Interface was strictly a procedural interface, and I needed to design and implement the corresponding object-oriented toolkit. Because the superior power, flexibility, and scalability of the RenderMan® Interface, that's what I chose to do.

Three Simple Examples

Let's begin with some simple examples of building models by writing them in **eve**, the modeling language I developed for WavesWorld. I'll quickly introduce many of the issues we're concerned with in practice, and then give a more formal overview and exposition of the framework. This section forward references material later in this chapter; the interested reader is encouraged to revisit this section after reading the rest of the chapter.

Static Example: A Cylinder

Let's say we want to build a dented metal cylinder. The first thing we would do is create a text file containing the following information:

```
set radius 1
set zMin -1
set zMax 2
set thetaMax 360
set color {.858913 .960738 1}
set materialName DentedMetal

startShape example1
   Color $color
   Surface $materialName
   Disk $zMin $radius $thetaMax
   Cylinder $radius $zMin $zMax $thetaMax
   Disk $zMax $radius $thetaMax
endShape
```

The preceding information is written in **eve**, a simple modeling language that we'll shortly compile into objects. As currently implemented, eve is a modeling language embedded in **tcl**; a popular extensible, embeddable scripting language. Eve extends tcl in several ways, most notably by adding a full binding for the RenderMan® scene description protocol. We'll talk about both tcl and RenderMan® more later, but for now the only thing to point out is that tcl refers to the contents of a variable by using the `$` sign, so whenever you see `$foo`, you can read it as "the value in the variable named `foo`."

If we compile the preceding file, it will compile the eve code into an ordered list of **renderable objects**. As you might imagine, a "renderable object" in WavesWorld is one that can be asked to render itself (either to a file, a framebuffer, etc.). If we ask these objects to render themselves to an image, we'd get:

Since these are objects, we can inspect them with tools available in WavesWorld. If we inspect the instantiated objects, we would see:

```
startShape example1
        Surface DentedMetal ;
        Color {0.858913 0.960738 1.0};
        Disk -1.0 1.0 360.0 ;
        Cylinder 1.0 -1.0 2.0 360.0 ;
        Disk 2.0 1.0 360.0 ;
    endShape
```

So what does this mean? Well, we have a container object (the "shape", bounded by the **`startShape`** and **`endShape`**) that has an ordered set of renderable objects inside of it. When the shape is asked to render itself, 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 now, would they affect the model at all? The answer is no, and the reason is that the values of the variables were resolved at the time we went from the text file written in eve and compiled into objects.

In WavesWorld, we would say that this model has "no potential for change" or "it's not animatable". In other words, once it is compiled into objects, everything about it is fixed; nothing is mutable, there are no degrees of freedom left in the model.

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?

Dynamic Example: Rotatable Cylinder

Let's say we want to build a simple cylinder that can rotate end over end. In contrast to the previous example, this model will have one degree of freedom: its rotation in the X axis.

The first thing we would do is create the model file. It will be the same as the previous one, with one important change: since we want to give it a degree of freedom in rotation, we'll mark the Rotate object as "animatable", which means that the expression the object is based on may change over time:

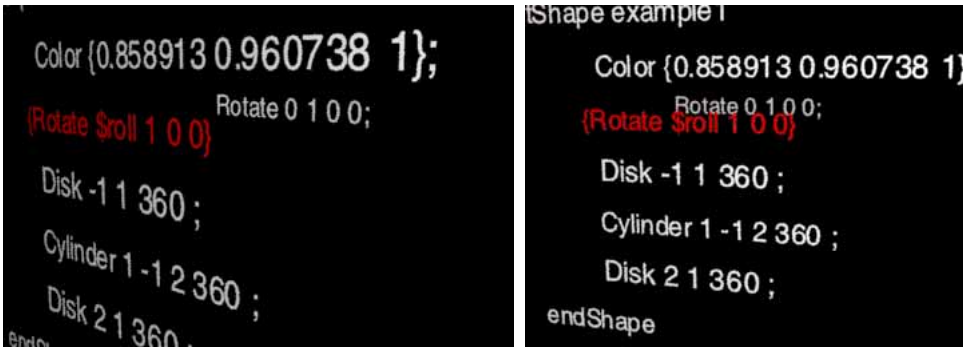
```
set roll 0
set radius 1
set zMin -1
set zMax 2
set thetaMax 360
set color {.858913 .960738 1}
set materialName DentedMetal

startShape example1
   Color $color
   Surface $materialName
   animatable: {Rotate $roll 1 0 0}
   Disk $zMin $radius $thetaMax
   Cylinder $radius $zMin $zMax $thetaMax
   Disk $zMax $radius $thetaMax
endShape
```

If we now compile this eve code, we will again create a dented metal cylinder, which, when we ask the objects to render themselves, looks exactly like the first example:

But what if we inspected the objects—what would we see? Well, using the object inspector in WavesWorld (wherein we visualize the model in 3-space, with Z

corresponding to time) and here are two views of what our instances look like:



So what does this mean? Well, we have a model composed of an ordered set of renderable objects. When the model is asked to render itself by a camera, it tells each of its renderable objects to draw themselves. The difference is that now one of those renderable objects is now also an **animatable** object.

All objects in a model (i.e. all *renderable* objects) know how to render themselves **over some interval of time**, i.e. from time 1.0 seconds to time 1.033 seconds. A basic renderable object, though, has the same way of rendering itself at *any* time. This means that if we asked it to render itself at time 0, time 12.32, or time 543543.12, it would render itself exactly the same way. An *animatable* object, on the other hand, knows that any of its instance variables may change over time, and has several other objects inside itself to help it keep track of this information:

- an object containing its symbolic representation
- an object containing its time-stamped, generator-stamped, sampled representation

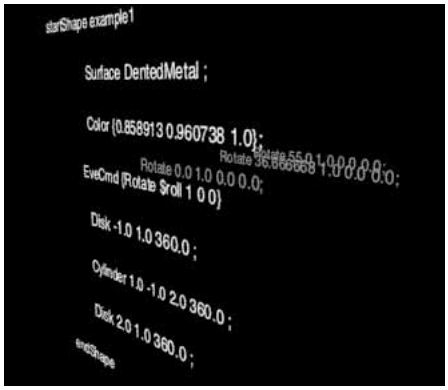
When an animatable object is asked to render itself over a given interval, it asks its list of samples for some representative set of samples over that interval. Usually, this means that it asks for two samples; a sample at the beginning of the time interval and one at the end of the time interval, but it could ask for more if the underlying renderer supported it. Either way, the animatable object then tells that list of representative samples to render themselves. This is an important process, and will be explained in more detail later in this chapter.

For example, if we rendered our cylinder now from time 1.0 to time 2.0, we'd get the

following image:

Note that this is exactly the same as the image we got when we asked the objects to render themselves at time 0.

This is because so far the animatable object has only has one sampled representation of itself—the **`Rotate`** object corresponding to time 0. If at time 1.5, though, the value of `$roll` suddenly changed, the animatable object would get sent a message to **resample** itself. It resamples itself by sending its symbolic representation (the expression "**`Rotate`** `$roll 1 0 0`") to the eve compiler built into the run-time system, which compiles the representation into a new object representing the current sample of the animatable command. Let's say that the value of `roll` changed to 55.



Now suppose we again rendered the scene starting at time 1.0 and ends 1/4 of a

second later. The rendered image looks like this:

Notice that the image is a little smeared and blurry; nowhere near as sharp as the first image. Why is this? Well, when the objects were asked to render themselves for this frame, each instance in turn tries to render itself over the course of the frame. In the current implementation, this means that each object tries to render itself at the beginning and end of the frame.

All the instances except the animatable command have only one representation of themselves, and use that representation at both the beginning (at time 1.0) and end of the frame (at time 1.25). In the case of the animatable object, though, it has a list of samples, containing only two samples. The animatable command treats its samples list as representative of a continuous signal, though, and asks it to provide a list of samples over the span of time bounded by time 1.0 and 1.25.

The samples list looks at the samples that it does have, which are at 0.0 and 1.5. It then tries to manufacture an appropriate intermediate sample at time 1.0 by making a copy of the sample at the last point before the current one and asking it to linearly interpolate itself appropriately with the first sample after the current one. In this case, this means that a new **`Rotate`** object for time 1.0 that has an angle of 36.7. It then generates a new intermediate sample at time 1.25, which is a **`Rotate`** object with an angle of 44.

The animatable object then tells both of these objects to render themselves, which when all the commands are finally evaluated by the renderer, yields the motion blurred image we see above.

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 the run-time system and asks it to perform a *closure* on it. In other words:

1. The eve compiler takes the symbolic description and identifies all the variable expressions in it.
2. 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.
3. The other variables, the global ones, are assumed to be persistent over the model, and for each global variable in the expression the eve compiler 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) the WavesWorld run-time system will insure that the animatable command will get sent a message to resample itself. Each of these variables is called an **articulated variable**.
4. When the animatable command is told to resample itself:
    1. it hands its symbolic expression back to the eve compiler (which is part of the run-time system) which recompiles it into objects in the current context.
    2. it then ask the run-time system what time it is.
    3. it then asks the run-time system the name of the agent generating this sample. This information is used later to blend the various samples together; each agent's name can be mapped to a weight value.
5. It then wraps all three pieces of information up in a `WWSample` object, and stores it in its sample list.

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

```
animatable: {Rotate $roll 1 0 0} {\
             {0   { {{rollAgent} 1 {Rotate 0 1 0 0;}} }} \
             {1.5 { {{rollAgent} 1 {Rotate 55 1 0 0;}} }} \
            };
```

can be read as:

"Define an animatable object which has the symbolic expression of **`Rotate`** `$roll 1 0 0`. This animatable object depends on a single articulated variable, `roll`. We have two samples of this expression already; at time equals 0, it compiled to the command **`Rotate`** `45 1 0 0` which was generated by the agent "rollAgent", and at time 1.5 it compiled to the command **`Rotate`** `55 1 0 0` which was also generated by the agent "rollAgent." In both cases, the blending weight of "rollAgent" is 1. We'll add these two samples to the animatable command's list of samples."

Abstraction Example: Squishy Sphere

In the previous example, we saw how we could express the potential for change over time in a model, and then, by acting on that model's parameters over time, we could animate it. But what if we wanted to "package up" our model, and only manipulate it via a few simple parameters? In other words, what if we wanted to treat our whole model as a single object, and not a complex set of variables and commands? We don't want to give up the facilities we just saw, though, such as the ability of the model to automatically interpolate the samples of itself over time. How could we express this in eve?

Let's say we want to build a model of a sphere that we can squash and stretch. To that end, we might build a model like this:

```
set sphere(v) 1.0

startShape squishySphere
  Color {1 0 0}
  Surface plastic
  animatable: {wwSet sphere(xScale) [expr 1./sqrt($sphere(v))]}
  animatable: {wwSet sphere(yScale) $sphere(v)}
  animatable: {wwSet sphere(zScale) [expr 1./sqrt($sphere(v))]}
  animatable: {Scale $sphere(xScale) $sphere(yScale) $sphere(zScale)}
  Sphere 1 -1 1 360
endShape
```

When we compile this code into a model, we would have dependencies set up on four articulated variables: `sphere(xScale), sphere(yScale), sphere(zScale)`, and `sphere(v)`. Whenever `sphere(v)` changed, each of the animatable objects containing the variable `sphere(v)` would get asked to resample themselves (i.e. the three **`wwSet`** commands, which set the value of their first argument to the value of their second argument). Those would then trigger a resample message to be sent to the animatable **`Scale`** object, which would resample itself with those newly calculated values.

There's nothing particularly bad about building our model this way, but we can see that this level of description can become cumbersome quickly. To help the developer build up abstractions, eve allows a developer to define a new class on the fly and make any instance of the newly defined class "animatable", just as they could for any of the built-in classes.

Using this mechanism, we can do the following:

```
defineClass: squishySphere {squish} {

  Color {1 0 0}
  set xScale [expr 1./sqrt($squish)]
  set yScale $squish
  set zScale [expr 1./sqrt($squish)]
  Scale $xScale $yScale $zScale
  Surface ColoredFilledWeb
  Sphere 1 -1 1 360
}

set sphere(v) 1.0

startShape squishySphere
  animatable: {squishySphere $sphere(v)}
endShape
```

What's especially useful about this approach is that now we suddenly have a new primitive that we can use anywhere. Also, since the new class is defined at the scripting language level, we can easily download the definition of this new class to some process on a network that is running the eve run-time system. This is where the power of eve really shines.

Eve is an object-oriented modeling language which draws heavily on the RenderMan® Interface. Eve allows modelers to build arbitrarily complex, photorealistically renderable models that can change continuously over time. It allows for packaging up arbitrarily complex models into objects which have their own sophisticated multi-modal graphical user interface (**GUI**).

In its current incarnation, eve is based atop **tcl**, the Tool Command Language (**Ousterhout94**), and uses tcl for doing computation on the arguments given to the instance variables of the objects that compose a given model. The choice of tcl is a historical artifact; many languages (Scheme, Dylan, Java, etc.) could serve as a base language for eve (and probably will, in future work). The best way to think about what eve is, is to look at the popular parallel language **Linda** (**Carriero90**). In the same way that Linda is a *coordination* language (**Gelernter90**) that is orthogonal to the *computation* language that it is embedded in (i.e. C, C++, LISP, etc.), eve is a *modeling* language that is orthogonal to its computation language (which is currently tcl).

This section begins with a story of an experience that helped focus the design aesthetic behind the language, then segues into a discussion of the three main constructs in the language: articulated variables, renderable objects, and animatable objects. I then discuss, at a lower, implementation level, the *protocols* and *classes* that make up the WW3DKit that sits below eve, and then what happens when the object database is asked to render itself over time. Finally, I mention some additions I had to make to tcl to make it more amenable for use as a computation language for eve. This discussion should be useful to readers who want to add eve to other dynamic languages.

Making a Model "Animatable"

A few years ago when I was working out in California, my officemate came in to work on a film she was doing some free-lance character animation for. She had a beautiful wax maquette of the head of an elf, one of the main characters in the piece she was working on. She sat down to her SGI and Alias PowerAnimator to work, and a mere two hours later when I looked over to see how she was doing, I was amazed to see the little elf's head, fully realized on the screen; a perfect 3D digital version of the wax analogue next to the monitor.

"Annabella", I exclaimed, "that looks great! You're all done."

"Oh no, wave", Annabella said. "I'm only halfway done."

"But it looks great! What more do you need to do?" I asked.

"I've built the model, yes, but now I have to make it **animatable**", she replied. She then spent the next several hours poring over the model, grouping control points, naming parts, tweaking here and there. When she finished, the model, to my eyes, looked the same, but now, according to Annabella, it was "animatable".

Animatable == has the potential for change

Animatable, in this context, refers to the *potential for change* in a given model. If you pull on this part, what happens? If you push here, prod there, what happens? A model is an immutable thing; static and stiff. An animated model is one which is changing in some particular way over time. An animatable model, on the other hand, is one that has the possibility of changing over time, and has appropriate controls for engendering this activity.

This vital distinction is one that is usually lost on everyone but animators that work in three dimensions (in either the analog or digital domain). As a CG person (but not really an animator), I knew that it was essential to build your model carefully so that you could manipulate it easier, but I'd not really "gotten it" until Annabella so succinctly summed it up for me that day.

In building a model and then immediately animating it, we can design it with the trade-offs for that particular animation in mind. If we need to do simple animation, perhaps just affine transformations on a whole hierarchy of objects, it's fine if the hierarchy is actually all jumbled up, with coordinate frames at arbitrary locations with respect to the various sub-objects.

If, on the other hand, we'll be doing some inverse kinematics animation on a hierarchical chain of joints, we need to be certain we've modeled them in a way that is amenable to manipulation by the inverse kinematic routines. If we plan to use physically based techniques for animating the model, in addition to such kinematic parameters, we also need to make sure that the specification of the various parameters to be used in the physical simulation (weight, inertia, plasticity, etc.) all make sense.

Clearly, a framework that purports to address modeling must give equal attention to allowing the construction of "animatable" models, where **animatable** is loosely defined as having *degrees of freedom* (that we call **articulated variables** (**Reeves90**)) that can be

manipulated over time in a straightforward manner, where changing the value of a given variable will change the model in some well understood manner.

Articulated Variables: A Model's Namespace

Looking at a model from the outside, the only thing that is visible are the variables (or slots, for the readers who prefer LISP) in the model. Some of the variables are read only, some are both readable and writable. Some change over time, some don't. They can be of a wide variety of types (bounded or unbounded floats and integers, colors, normalized vectors, enumerated strings, lists, etc.).

Once a model has been compiled, the only way for a process outside of the model to manipulate the model is to write a value to one of the model's variables. In order to write to a variable, a process must first **attach** to a variable. Once they've successfully attached to a variable, a process can write new values to it. When they are done writing to the variable, they must **detach** from it. Whenever a process attaches to or detaches from a variable, all the objects in the model that depend on that variable are notified.

We'll talk more about this in the next chapter (since this is the responsibility of the processes manipulating the variables, not the modeling language), but for now it's only important to point out that the only variables that can be attached or detached are ones that can change over time, and these variables are normally referred to as **articulated** variables, where the term comes from Reeves et.al's (**Reeves90**) use of it.

The Model == Renderable Objects (some of which are Animatable)

Once we zoom into the compiled model, we see that these articulated variables are directly wired to an ordered list of **renderable** objects. These renderable objects all conform to the *WWRenderable protocol*, which is a set of messages that all of the objects respond to. These messages include messages to ask the object to render itself over the course of some amount of time to a variety of output formats, as well as other messages, such as what the bounding box of the object is over some span of time.

In the same way that some model variables can change over time, some of the renderable objects in the model can too. Any object which can change over time is called **animatable**. It should be noted that an animatable object is, by definition, a renderable object.

There is a core set of classes provided with the base eve implementation that

conform to the WWRenderable protocol. They essentially map directly to calls to the RenderMan® Interface, with two vital additional ones: **RIBCommandList** and **WWSet**.

`RIBCommandList` is a class which contains an ordered list of renderable objects (any of which may be instances of **`RIBCommandList`**). By using instances of this class, arbitrary new classes can be easily composed; either in eve or in the low-level implementation language of the object framework (currently Objective-C). Instances of these new classes can be used in models in exactly the same way as any instance of the core classes, since they all eventually reduce to lists of instances of objects from the core classes.

WWSet is a class that facilitates simple constraint chaining (like the *abstraction example* above), by allowing the value of an articulated variable to be calculated automatically from the values of other articulated variables. Instances of this class should be used sparingly, and almost always for articulated variables that are read-only outside of the model.

Sometimes you want to implement a new class directly in a low level language, because you need to get at resources that aren't available from the computation language hosting eve. This is usually only a problem in languages like tcl, which are intended to be extended in their implementation language (namely C).

For example, one especially useful new class that was implemented directly in the low-level implementation language is a 3D text object (originally written for me by Ian Wilkinson at Canon UK Research). Given a font name, point size, justification (left, right, centered) and some piece of text, this object will create a list of corresponding renderable objects that will scale, translate, and draw the corresponding polygons. This was implemented in Objective-C since getting the geometric information for the fonts involved talking to the Postscript WindowServer to get that info, and tcl had no facilities for doing this. It's important to note, though, that for all of the models you'll see in this dissertation (and in the entire WavesWorld examples suite), this is the only class that had to be implemented this way. Everything else you see in WavesWorld other than 3D text was built through simple composition of other eve objects.

The Eve Commands

Here is a list of the current core set of eve commands that are accessible from the eve language level; you should check the documentation and large examples set that accompanies WavesWorld (and also the RenderMan® Interface documentation (**Pixar89**))

for details of the arguments these objects take and how to use them. The following is intended to be a brief summary of the eve commands available in the current implementation. Since eve sits atop the RenderMan® Interface, this section borrows heavily from (**Pixar89**).

animatable: *eveEpression* [*samplesList*]
Allow the objects defined by eveExpression to change over time. If *sampleList* is present, it is evaluated and added to the list of time-stamped, generator-stamped samples of the objects' representation.

defineClass: *className instanceVariables classDefinition*
Defines a new eve command named *className* that takes the arguments *instanceVariables,* defined by the code in *eveExpression.*

wwSet *varName varValue*
Sets the global variable *varName* to *varValue.*

startShape *shapeName* [*4x4TranformationMatrix*]
This maps to a list of objects containing an **`AttributeBegin`** and an **`Attribute`** object, where the **`Attribute`** object has "identifier", "name", `$shapeName` as its instance variables. Also, a 4x4 transformation matrix can optionally be added after *shapeName*, at which point there is a **`ConcatMatrix`** object put on the end of the list containing that 4x4 transformation matrix.

endShape
Pops the current set of attributes, which includes the current transformation, same as **`AttributeEnd`**.

ArchiveRecord comment|structure *string1 ... stringN*
Writes a user data record that is the concatenation of the *stringXX* arguments with a new-line at the end. It's an error to try and embed a new-line in the *stringXX* args.

AreaLightSource *name lightHandle* [*parameterList*]
Adds the area light source using the shader named *name* to the current light list.

Atmosphere *shaderName* [*parameterList*]
Sets the current atmosphere shader to *shaderName.*

Attribute *name parameterList*
Sets the parameters of the attribute *name,* using the values specified in the token-value list *parameterList.*

AttributeBegin
Pushes the current set of attributes, which includes the current transformation.

AttributeEnd
Pops the current set of attributes, which includes the current transformation.

Basis *uName|uBasis uStep vName|vBasis vStep*
Sets the current u-basis to either *uBasis* or *uName*, and the current v-basis to either *vBasis* or *vName*, where *uName* and can be either of the following enumerated strings: **bezier**, **b-spline**, **catmull-rom**, **hermite**, **power**. The variables *ustep* and *vstep* specify the number of control points that should be skipped in the u and v directions, respectively. The default basis is **bezier** in both directions.

Color *color*
Sets the current color. In the current implementation, only rgb is supported, so *color* should be a list of 3 elements, normalized between 0 and 1, respectively

corresponding to the red, green and blue intensities.

ConcatTransform *4x4TransformationMatrix*
Concatenate the transformation matrix *4x4TransformationMatrix* by premultiplying it with the current transformation matrix.

Cone *height radius thetaMax* [*parameterList*]
Defines a cone *height* high with a radius of *radius,* swept *thetaMax* degrees about Z.

CoordinateSystem *space*
Names the current coordinate system as *space.*

Cylinder *radius zMin zMax thetaMax* [*parameterList*]
Defines a cylinder from *zMin* to z*Max* with a radius of *radius,* swept *thetaMax* degrees about Z.

Declare *name declaration*
Declares the variable *name* using *declaration,* where it is composed of [*class*] [*type*] [[*n*]], where *class* is one of **int, float, color, point, char,** and *type* is either **varying** or **uniform**, and *n* is some positive, non-zero integer.

Disk *height radius thetaMax* [*parameterList*]
Defines a disk *height* high with a radius of *radius,* swept *thetaMax* degrees about Z.

Displacement *shaderName* [*parameterList*]
Sets the current displacement shader to *shaderName.*

Exterior *shaderName* [*parameterList*]
Sets the current exterior volume shader to *shaderName.*

GeneralPolygon *nVertices parameterList*
Defines a general polygon of *nVertices* using the information in *parameterList*.

Hyperboloid *point1 point2 thetaMax* [*parameterList*]
Defines a hyperboloid where a line defined by the endpoints *point1* and *point2* swept *thetaMax* degrees about Z.

Identity
Sets the current transformation matrix to the identity.

Illuminate *lightHandle flag*
If *flag* is true (i.e. non-zero), the light source corresponding to *lightHandle* is turned on, otherwise it's turned off.

Interior *shaderName* [*parameterList*]
Sets the current interior volume shader to *shaderName.*

LightSource *name lightHandle* [*parameterList*]
Adds the non-area light source using the shader named *name* to the current light list.

MakeBump *pictureName textureName sWrap tWrap filter sWdith tWidth* [*parameterList*]
Makes a bump map.

MakeCubeFaceEnvironment *pX nX pY nY pZ nZ textureName fov filter sWidth tWidth* [*parameterList*]
Makes an environment map from six images.

MakeLatLongEnvironment *pictureName textureName filter sWdith tWidth* [*parameterList*]
Makes an environment map from a single latitude-longitude image.

MakeShadow *pictureName textureName* [*parameterList*]
Makes a shadow map from *pictureName*.

MakeTexture *pictureName textureName* [*parameterList*]
Makes a texture map from *pictureName*.

NuPatch *nU uOrder uKnot uMin uMax nV vOrder vKnot vMin vMax* [*parameterList*]
Defines a non-uniform rational b-spline surface.

Opacity *color*
Sets the current opacity. In the current implementation, only rgb is supported, so *color* should be a list of 3 elements, normalized between 0 (completely transparent) and 1 (completely opaque), respectively corresponding to the red, green and blue opacities.

Paraboloid *rMax zMin zMax thetaMax* [*parameterList*]
Defines a paraboloid from *zMin* to z*Max* with a maximum radius of *rMax,* swept *thetaMax* degrees about Z.

Patch *type parameterList*
Defines a uniform patch of type *type*, using the information in *parameterList*.

PatchMesh *type nU uWrap nV vWrap parameterList*
Defines a uniform patch mesh of type *type*, using the information in *parameterList*.

PointsGeneralPolygons *nLoops nVertices vertices parameterList*
Defines a set of general polygons.

PointsPolygons *nVertices vertices parameterList*
Defines a set of convex polygons.

Polygon *parameterList*
Defines a convex polygon.

Rotate *angle dX dY dZ*
Concatenate a rotation of *angle* degrees about the given axis onto the current transformation.

Scale *sX sY sZ*
Concatenate a scaling onto the current transformation.

ShadingRate *size*
Sets the current shading rate to *siz*e.

Sides *sides*
If *sides* is **2**, subsequent surfaces are considered two-sided, and both the inside and the outside of the surface will be visible. If *sides* is **1**, subsequent surfaces are considered one-sided and only the outside of the surface will be visible.

Skew *angle dX1 dY1 dZ1 dX2 dY2 dZ2*
Concatenate a skew onto the current transformation.

SolidBegin *operation*
Begins the definition of a solid. *operation* may be one of the following: **primitive, intersection, union, difference.**

SolidEnd
Terminates the definition of a solid.

Sphere *radius zMin zMax thetaMax* [*parameterList*]
Defines a sphere from *zMin* to z*Max* with a radius of *radius,* swept *thetaMax* degrees about Z.

Surface *shaderName* [*parameterList*]
Sets the current surface shader to *shaderName.*

TextureCoordinates *s1 t1 s2 t2 s3 t3 s4 t4*
Sets the current set of texture coordinates to these values, where (s1,t1) maps to (0,0) in **uv** space, (s2,t2) maps to (1,0), (s3,t3) maps to (0,1), and (s4,t4) maps to (1,1).

Torus *majorRadius minorRadius phiMin phiMax thetaMax* [*parameterList*]
Defines a torus with a major radius of *majorRadius* and a minor radius of *minorRadius*, where it is swept in XY from *phiMin* to *phiMax,* swept *thetaMax* degrees about Z.

Transform *4x4Matrix*
Sets the current transformation matrix to *4x4Matrix*.

TransformBegin
Pushes the current transformation.

TransformEnd
Pops the current transformation.

Translate *dX dY dZ*
Concatenate a translation onto the current transformation.

TrimCurve *nLoops nCurves order knot min max n u v w*
Sets the current trim curve, which is applied to NuPatch objects.

Composing New Classes

In eve, you can define a new class by simply declaring its name, arguments, and the ordered renderable objects that comprise it. For example, as we saw in the last section, we could define a "squishy sphere" class with a single instance variable with the following eve code:

```
defineClass: squishySphere {squish} {

  Color {1 0 0}
  set xScale [expr 1./sqrt($squish)]
  set yScale $squish
  set zScale [expr 1./sqrt($squish)]
  Scale $xScale $yScale $zScale
  Surface ColoredFilledWeb
  Sphere 1 -1 1 360
}
```

This would compile into a **RIBCommandList** object that had 4 objects inside of it: **Color**, **Scale**, **Surface**, and **Sphere**. New instances of this class could now be instantiated just like any of the core classes.

Building up the Scene

In WavesWorld, a model is dropped into an on-screen view called a **WW3DWell**. The **WW3DWell** serves as the gateway to WavesWorld; every object that is in WavesWorld passes through the **WW3DWell**. When a new model is dropped into the **WW3DWell**, the

time in the scene is reset to zero and the model is compiled into an ordered list of renderable objects, some of which are also animatable. As we mentioned earlier, when an animatable command is first instantiated, it hands its symbolic representation to the run-time system and asks it to perform a *closure* on it. In other words:

1. The eve compiler takes the symbolic description and identifies all the variable expressions in it.
2. 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.
3. The other variables, the global ones, are assumed to be persistent over the model, and for each global variable in the expression the eve compiler 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) the WavesWorld run-time system will insure that the animatable command will get sent a message to resample itself. Each of these variables is called an **articulated variable**.
4. When the animatable command is told to resample itself:
    1. it hands its symbolic expression back to the eve compiler (which is part of the run-time system) which recompiles it into objects in the current context.
    2. it then ask the run-time system what time it is.
    3. it then asks the run-time system the name of the agent generating this sample. This information is used later to blend the various samples together; each agent's name can be mapped to a weight value.
5. It then wraps all three pieces of information up in a **`WWSample`** object, and stores it in its sample list.

Once the model has been compiled, time begins moving forward. It may move forward at pace with wall clock time, or it may move forward much slower or much faster. One important feature of WavesWorld is the lack of an enforced lock-step time increment. Time may move forward in increments of seconds, or it may move forward in 1/100th of a second. In WavesWorld, everything is discussed with reference to "scene time."

Either way, outside processes begin contacting the run-time system and asking to **attach to** a given set of **articulated variables**. Each process (referred to as an **agent** in WavesWorld) provides the run-time system with several pieces of information: its name, how long (in the scene's time, not wall clock time) the agent plans to stay attached to the variable ("an indefinite length of time" is a valid response), and the kind of interpolation the run-time system should use to interpolate the agents' sampled signal with respect to a given articulated variable (right now, only linear is supported). For each agent, the run-time system has a weight that it associates with a given named agent, which it will use to blend this agent's contribution to the articulated variable's value.

If the agent gave a finite amount of time that it would stay attached to the articulated variable, the run-time system ensures that the connection is closed after that much time has elapsed in the scene. If, on the other hand, the agent gave “indefinite” as the length of time it would stay attached, it needs to send an explicit “detachFrom” message to the run-time system when it is done. Either way, if a process’ connection to the run-time system is broken before its allotted time, the run-time system will automatically clean up and detach it from all its associated articulated variables.

With respect to the run-time system, agents can do four things: attach to it, set a value of a variable managed by it, get a value of a variable managed by it, detach from it. We’ve already seen how an agent can attach, set, and detach from the run-time system, but what happens when an agent tries to get the value of a variable? If the value is not articulated, the current value is simply returned. If, on the other hand, the variable is articulated, the run-time system returns the value of the variable “a moment ago”, where this is dependent on how time is currently moving forward in WavesWorld. Note that returning this value is potentially a very complex operation, especially if there are several agents that are contributing to the value of this variable. The run-time system, using the time-stamped, sampled representations of the variable’s value from the agents that are currently contributing to its value are blended together using the weights associated with those agents, and that value is returned.

Shooting the Scene

Once we've built a model by writing eve code and compiling it into objects, and then constructed a scene wherein our model changed over the course of some amount of time, we can think of our ordered list of renderable objects as an *object database*. Each object in the database is either a single renderable object or a list of renderable objects, where this continues recursively until each object reduces to a single atomic (i.e. not an instance of **`RIBCommandList`**) object. All objects know how to render themselves at any given positive point in time in the scene (in WavesWorld, time always starts at 0 and is never negative). In order to render the objects (to a file, the screen, etc.), we ask the first object to render itself starting at some point in time, and ending at some later point. In other words, we give it some span of time over which we want it to render itself (i.e. 1.1 to 1.25). It's useful to point out that these two values could be the same, but the second will always be equal to or greater than the second. Each object renders itself in turn and sends

the message on to its descendant in the list.

Note that it's very straightforward to have objects that are *not* animatable render themselves at any given point in time, since none of their instance variables ever change.

Objects that *are* animatable, though, are more complex, since it is assumed that their instance variables can be changing continuously over time. Unfortunately, this continuous signal is most likely represented as a series of samples, where interpolation needs to be done. Each animatable object has a `samplesList` containing time-stamped, agent-stamped (i.e. which agent generated this sample) instances of this object. For each agent that impacts this object, the object has a notion of over what spans of times each of the sample generators were active (i.e. that they were **attached** to the **articulated variables** that this animatable object depends on).

When asked to render itself at time over some span of time, the animatable object asks itself (which in turn looks at its samples list object) for a set of samples spanning that time. In the simplest case (which we'll restrict this discussion to), this would yield a list of two samples: one at the beginning and one at the end of the span.

The samples list then tries to provide a sample at the given points in time. It does this the same way the run-time system provides a value to an agent; by blending the various signals, weighted by the weights corresponding to the agents that generated them. In other words, it might have one signal from agent `foo` and another signal from one agent `bar`. There might be a weight of 0.5 associated with `foo` and 1.0 associated with `bar`, so it would return a signal that was interpolated using both `foo` and `bar`, but twice as much credence would be given to `bar`'s signal.

Once it has the two samples of the animatable object (one for the start time of the frame, one for the end time of the frame it is rendering), the animatable object sends itself a `renderCompoundCommand::` message with the two samples as arguments.

At that point, the two samples are asked a series of questions by being sent messages in the *WWRenderable protocol*, which all renderable objects conform to.

First the two samples are asked if they are instances of the same class. If they aren't, the first sample is asked if it is not *moot* (i.e. does it actually affect the rendering environment—a relative translation of (0, 0, 0) *would* be considered moot) the sample is asked to render itself and the method returns.

If they are the same class, the first sample is asked if it is motion blurrable or not. If it

isn't, the first sample is asked if it is not *moot* (i.e. does it actually affect the rendering environment—a relative translation of (0, 0, 0) *would* be considered moot) the sample is asked to render itself and the method returns.

Now, each of these samples might be instances of either one of the core, atomic objects or a `RIBCommandList` object (also referred to as a "compound command"). At this point the samples are asked if they are compound commands. If they are, it asks for the first renderable object from each, and recursively calls `renderCompoundCommand::` with those two samples as arguments.

Eventually, `renderCompoundCommand::` is called with two samples that are *not* compound commands.

The two samples are then asked if they are the same (i.e. equivalent values of all of their instance variables). If they are, the first sample is asked if it is not *moot* (i.e. does it actually affect the rendering environment—a relative translation of (0, 0, 0) *would* be considered moot) the sample is then asked to render itself and the method returns.

If the two samples aren't the same, the animatable object being rendered (of which these are two samples of it) generates a `MotionBegin/End` block (**Pixar89**) and asks both samples to render themselves within it.

The powerful thing to note about this approach is that assuming the original database accurately sampled the model over time, it can be rendered accurately at arbitrary spatial and temporal resolution, taking full advantage of the RenderMan® Interface and be able to produce photorealistic images of 3D scenes.

Tcl Additions

As I mentioned at the beginning of this section, the fact that eve uses tcl is a historical artifact, and eve could be added to any sufficiently dynamic language. We've had to extend tcl in several ways, though, and this section discusses these issues, in the hope that it will illuminate these topics for others interesting in gaining a better understanding of eve in general, or adding eve to *their* favorite language.

Types

Tcl is completely string based; that is its only type. This is useful, especially for debugging purposes and shipping code around a network of disparate computing resources, but can, for obvious reasons, become a performance bottleneck. In practice, since most of the modeling we've done with WavesWorld has not involved generating

complex curved surfaces on the fly and animating their control points, this has *not* been an issue (which, frankly, surprised me). It's clear, though, that this should be addressed so that we can do such things at a reasonable rate on either slower machines or for more complex models.

Because tcl makes it exceptionally easy to trace reads and writes on variables, it is easy to add a rich set of types, especially when you are adding your own routines (i.e. a RenderMan® binding) which can be built to use these types. The ones we've found useful to define for eve include: **strings**, **enumerated strings** (specified explicitly or implicitly with a regular expression), **booleans**, **integers**, **bounded integers** (i.e. inclusively or exclusively bounded by a min, a max, or both a min & a max), **floats**, **bounded floats**, **multi-dimensional arrays** of any of those types (although mainly ints and floats), and **normalized multi-dimensional arrays** of floats.

All of these are straightforward to implement in tcl, although there is an obvious performance penalty, since all writes must be monitored by the run-time system. Also, for the bounded variables, there's the question of exception/error handling; if someone tries to set a variable to an invalid value, should it return an error, log the event, or silently cast to the appropriate type or clamp the value within the acceptable range? Clearly, you may want any of the three in a given situation, so this needs to be a settable (and gettable) parameter in the run-time system.

Attaching to and Detaching from Variables

One of the key ideas in eve is that a model is defined at time zero, at which point dependencies are set up between a set of articulated variables and the objects comprising the model. As time moves forward, whenever these variables change, objects that have registered a dependency on that variable resample themselves. One key point is that at over a given interval of time, there might be several processes (**agents**, see next chapter) manipulating a given articulated variable. In order for eve's run-time system to keep track of this, agents must be able to **attachTo** and **detachFrom** articulated variables. While the generation of these calls lie in the realm of the behavior generating processes and will not be visible to the person using eve as a modeling language, these facilities must nonetheless be there. I dealt with this issue in two ways: appcom and active objects. AppCom is a portable library for “application communication”, which allows easy, efficient typed messages to be sent between various UNIX boxes. Active objects are a way

of writing a process that has a core set of computation and communication functionality and has its own main loop. Active objects act as a "computation substrate" that agents and receptors (see next chapter) are embedded in.

Efficient Math on Arrays

The next issue is doing efficient and appropriate math on arrays. One reasonable solution to this is what Dave Chen at the MIT Media Lab did for his *3d* system (**Chen93**), wherein he used his array handling routines. A more flexible approach, and the one currently being integrated into WavesWorld, is the narray package by Sam Shen at Lawrence Berkeley Labs (**Shen95**). His package, like Chen's, allows for the creation of arrays which are referenced by name, but while Chen's approach was to create a large set of new tcl commands which took these arrays references as arguments, Shen provides a single routine, **`narray`**, like tcl's **`expr`**, for doing matlab-like operations on the arrays. In addition, because this package allows expressions to be carried out on elements of arrays, this allows points in arrays to be grouped together and manipulated easily and concisely (by, for example, assigning their references in a tcl variable which is then passed into the narray routine). This kind of point grouping are very useful and are absolutely vital when you are manipulating various portions of a single large patch mesh, something which comes up frequently in high-end animation production (**Serra93**).

More Math Functions

In addition to array operations and types, I found it useful to add all of the built-in functions from RenderMan®'s Shading Language. Since I was trying to do this in the most appropriate for the host language (i.e. tcl), I added these by extending tcl's **`expr`** routine, which unfortunately restricted me to scalar types. I can also add these to **`narray`**, though, which will give me equivalent functionality for arrays. Currently, these math functions include:

pi
returns the exact value (to 32 bit IEEE precision) value of PI

radians *angleInDegrees*
returns *angleInDegrees* converted to units of radians.

degrees *angleInRadians*
returns *angleInRadians* converted to units of degrees.

sign *arg*
returns -1 if *arg* is negative, 0 if zero, and +1 is *arg* is positive

min *arg1 arg2*
returns the minimum of *arg1* and *arg2*

max *arg1 arg2*
returns the maximum of *arg1* and *arg2*

clamp *val min max*
returns *val* clamped between *min* and *max*

step *min val*
returns 0 if *val* is less than *min*, otherwise it returns 1

spline *u pt1 pt2 pt3 pt4*
returns the point along the spline specified by the 4 control points at *u* (0<=*u*<=1) on the curve

smoothstep *min max value*
if *value* is less than *min*, it returns *min*, if it's above *max*, it returns *max*, otherwise it smoothly interpolates between them

lerpDown *u min max*
if *u* is less than or equal 0, it returns *max*, if it's above or equal 1, it returns *min*, otherwise it smoothly interpolates between them, downwards

lerpUp *u min max*
if *u* is less than or equal 0, it returns *min*, if it's above or equal 1, it returns *max*, otherwise it smoothly interpolates between them, upwards

noise *x y z*
it returns some value between 0 and 1 which is a pseudorandom function of its argument using Perlin's noise function (see Ch 2 of **Ebert94**)

gvnoise *x y z*
it returns some value between 0 and 1 which is a pseudorandom function of its argument using a gradient value noise function (see Ch 2 of **Ebert94**)

scnoise *x y z*
it returns some value between 0 and 1 which is a pseudorandom function of its argument using a sparse convolution noise function (see Ch 2 of **Ebert94**)

vcnoise *x y z*
it returns some value between 0 and 1 which is a pseudorandom function of its argument using another noise function (see Ch 2 of **Ebert94**)

In addition to these new math functions, I added a new base command, **`spline`**, that takes an arbitrary rank list of numbers of arbitrary length (at least 4 long), along with a *u* value from 0 to 1 and returns the point on that curve. I needed to do this because you can't have variable length argument lists to tcl's **`expr`** command.

Building and Debugging Models

One of the most important aspects of any development environment is the ease with which code under development can be debugged. From the outset, WavesWorld has been driven by the idea of trying to understand how to facilitate the construction of **debuggable** characters. As we've seen in this chapter, one of the central activities in building characters is the construction of an animatable model. But how are these models constructed? How does a model builder test the "animatability" of their model? How do they inspect other model parts they've gotten from collaborators? How do they debug their mental model of how these parts work and interrelate?

As with any programming language environment, models written in eve are written iteratively: starting with old code, modifying it, running it, tweaking, repeat. To truly understand the material in this section, you should get a demonstration of WavesWorld in action, but through the use of screen snapshots and explanatory text, I'll do my best to convey some of the power of this system for debugging models.

Dealing with a Model as a Database

One powerful way to think of a model is as a database: some set of typed fields that can be read and written. There are different levels of access to the fields; some processes can read and write values with impunity, some can only read, some can only write a range of values. The fields themselves may be strongly or weakly typed; given an inappropriate value they may choose to ignore it or modify it to fall within the range of values they accept. In the case of models in WavesWorld, these are more complex than many traditional databases, since most of the data has a notion of itself over time, and also has a notion of "parallel opinions" of the value of a given field, where the number and owners of opinions can also vary over time. Given this view, we want to be able to **manipulate** and **visualize** fields in the database. Let's see how this works in WavesWorld.

Manipulating and Visualizing Model Parameters

An important concept in building and debugging models is that the GUI does not have to map directly to the articulated variables of a model. This is useful when we want to (say) map several model variables to a single control (i.e. a "mood" control that maps to several variables), or perhaps we want to look at some component of a single articulated variable (i.e. the red component of a color). This is where having the power of

a computation language available to the GUI environment is vital.

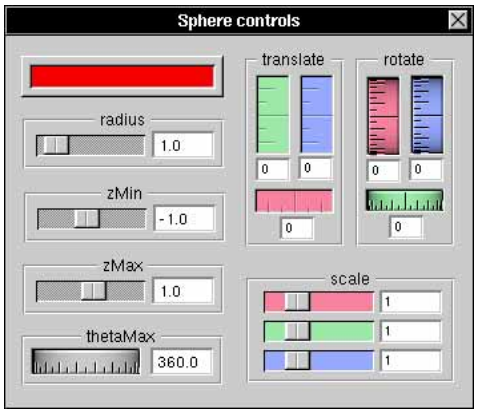
In general, though, we simply want to be able to manipulate fields in the database (i.e. the articulated variables of the model). For example, take the model:

```
startShape aSphere
  animatable: {Color $s(color)}
  animatable: {Scale $s(xScale) $s(yScale) $s(zScale)}
  animatable: {Translate $s(xT) $s(yT) $s(zT)}
  animatable: {Rotate $s(xRotate) 1 0 0 }
  animatable: {Rotate $s(yRotate) 0 1 0 }
  animatable: {Rotate $s(zRotate) 0 0 1 }
  animatable: {Sphere $s(r) $s(zMin) $s(zMax) $s(thetaMax)}
endShape
```

that has the following initial values:

```
set s(color) {1.0 0.0 0.0}
set s(r) 1.0
animatable: {wwSet s(zMin) [expr {-1 * $s(radius)}]}
animatable: {wwSet s(zMax) $s(radius)}
set s(thetaMax) 360.0
set s(xScale) 1
set s(yScale) 1
set s(zScale) 1
set s(xT) 0
set s(yT) 0
set s(zT) 0
set s(xRotate) 0
set s(yRotate) 0
set s(zRotate) 0
```

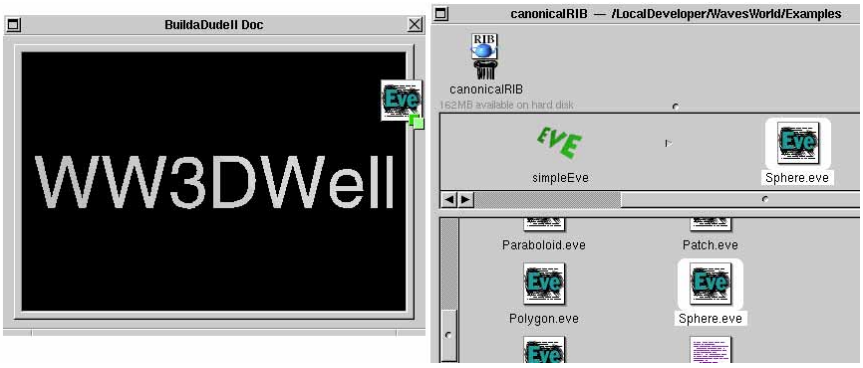
Now let's say we wanted to be able to manipulate these variables, say by dragging a slider or typing a value in a text box. For example, something like this:



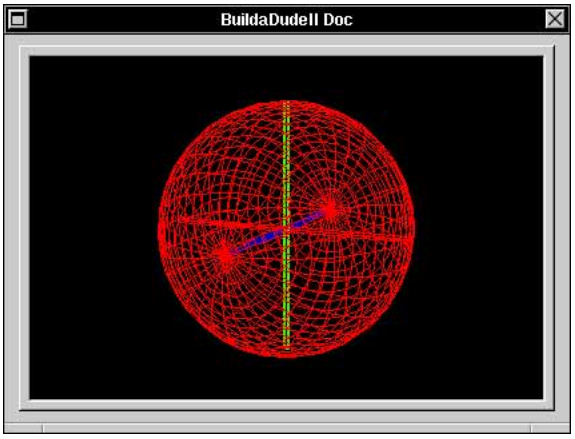
One nice side effect of this is that all of these GUI controls for manipulation also serve equally well as visualization aids. When we manipulate the slider that in turn manipulates the radius of the sphere, both the slider itself and the text field that is also tracking that variable are both automatically updated with the new value.

How Does it Work?

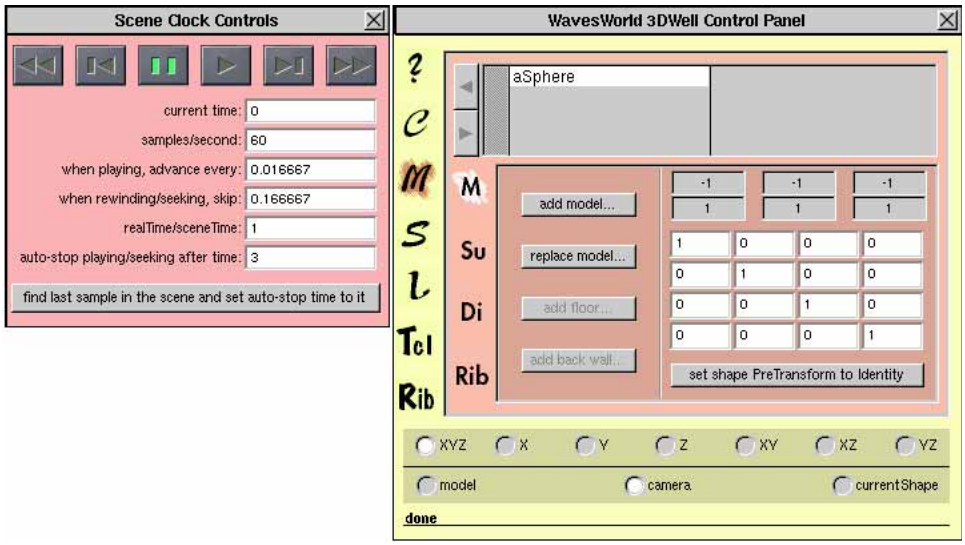
In WavesWorld, the eve compiler and run-time system is embodied in a **`WW3DWell`**, an on-screen object that you can drop models into:



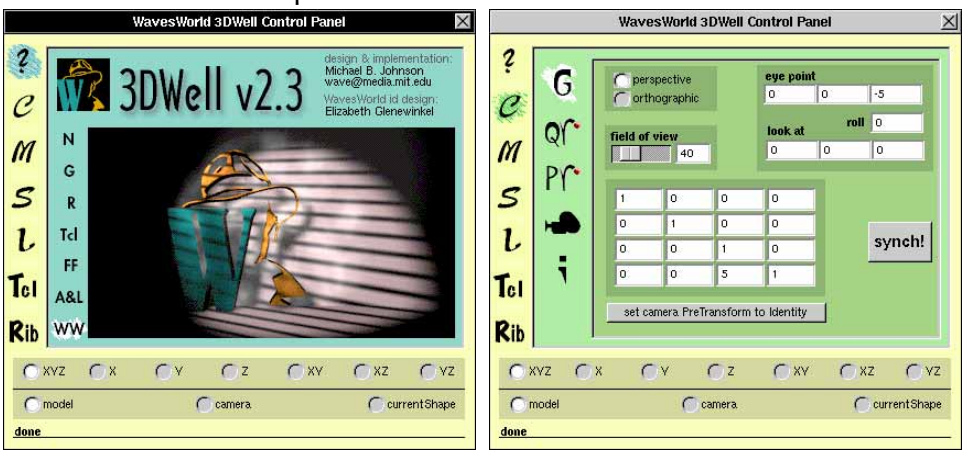
Once dropped into the WW3DWell, the eve code comprising the model is compiled and displayed:

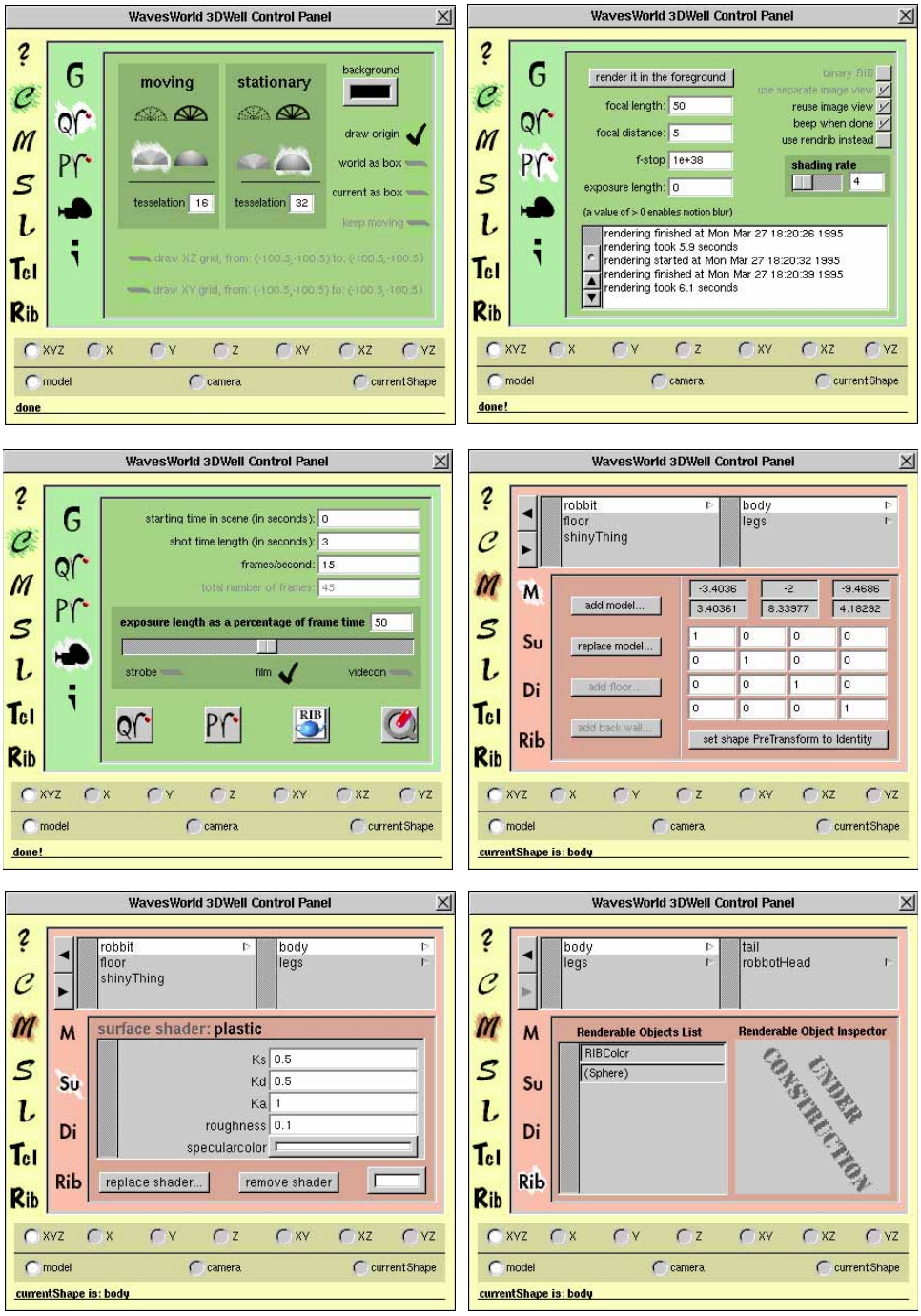


The WW3DWell has a "control panel" that can be accessed by clicking the edge of the WWW3DWell (similar to clicking a NXColorWell to bring up its control panel). For example, here's what a user sees the first time they bring click the edge of the WW3DWell:



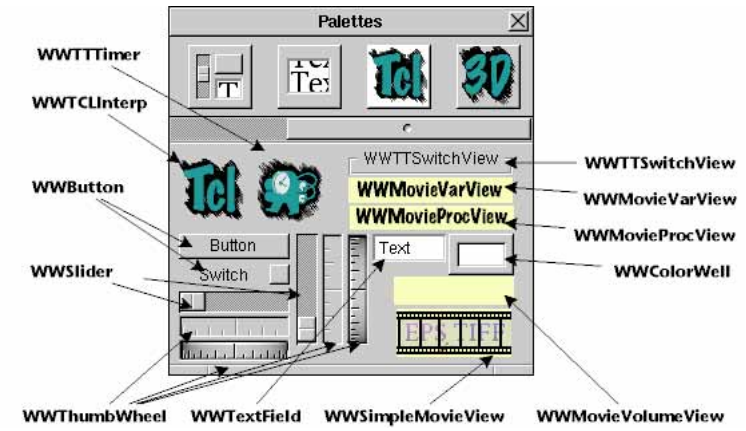
The WW3DWell's control panel has many sub-panels, some of which can be pulled off and made separate windows (like the "Scene Clock Controls" window in the illustration). The user can directly inspect all information in the database, and can freely move back and forth in time. Here are some of the sub-panels you would see if you were able to click on the control panel now:




WavesWorld 3DWell Control Panel
moving
stationary
background
draw origin
world as box
current as box
keep moving
tesselation 16
tesselation 32
draw XZ grid, from: (-100.5,-100.5) to: (-100.5,-100.5)
draw XY grid, from: (-100.5,-100.5) to: (-100.5,-100.5)
XYZ X Y Z XY XZ YZ
model camera currentShape
done
WavesWorld 3DWell Control Panel
render it in the foreground
binary RIB
use separate image view
reuse image view
beep when done
use rendrib instead
focal length: 50
focal distance: 5
f-stop 1e+38
exposure length: 0
(a value of > 0 enables motion blur)
shading rate 4
rendering finished at Mon Mar 27 18:20:26 1995
rendering took 5.9 seconds
rendering started at Mon Mar 27 18:20:32 1995
rendering finished at Mon Mar 27 18:20:39 1995
rendering took 6.1 seconds
XYZ X Y Z XY XZ YZ
model camera currentShape
done!
WavesWorld 3DWell Control Panel
starting time in scene (in seconds): 0
shot time length (in seconds): 3
frames/second: 15
total number of frames: 45
exposure length as a percentage of frame time 50
strobe film videcon
XYZ X Y Z XY XZ YZ
model camera currentShape
done!
WavesWorld 3DWell Control Panel
robbit
floor
shinyThing
body
legs
add model...
replace model...
add floor...
add back wall...
-3.4036 -2 -9.4686
3.40361 8.33977 4.18292
set shape PreTransform to Identity
XYZ X Y Z XY XZ YZ
model camera currentShape
currentShape is: body
WavesWorld 3DWell Control Panel
robbit
floor
shinyThing
body
legs
surface shader: plastic
Ks 0.5
Kd 0.5
Ka 1
roughness 0.1
specularcolor
replace shader...
remove shader
XYZ X Y Z XY XZ YZ
model camera currentShape
currentShape is: body
WavesWorld 3DWell Control Panel
body
legs
tail
robbotHead
Renderable Objects List
Renderable Object Inspector
RIBColor
(Sphere)
UNDER CONSTRUCTION
XYZ X Y Z XY XZ YZ
model camera currentShape
currentShape is: body


WavesWorld 3DWell Control Panel
Scene Clock Controls
current time: 0
samples/second: 60
when playing, advance every: 0.016667
when rewinding/seeking, skip: 0.166667
realTime/sceneTime: 1
auto-stop playing/seeking after time: 3
find last sample in the scene and set auto-stop time to it
XYZ X Y Z XY XZ YZ
model camera currentShape
currentShape is: body
WavesWorld 3DWell Control Panel
Type directly to the Model's Tcl Interpreter:
info vars
Keys
bounceMotorProgram
treatTransformBeginLikeAttributeBegin
RiOrientation string
--> info vars
<-- ambientLight leg foot toe orient constrict tail
squishyJoint lowerHead dilate val2 trackLight
scene moveHim2 wag sit moveAndOrientHim i
rotIncr glow env PI ticksPerSecond { head} {
robbit} stand twirl head robbit orientHead val
legTop bounce moveHim sideToSide torus
shinyThing orientBoth raiseHead modelPath
XYZ X Y Z XY XZ YZ
model camera currentShape
currentShape is: body
WavesWorld 3DWell Control Panel
ignore ShadingRate
ignore color
ignore lights
firm transforms
ignore all shaders
treat AttributeBegin like startShape
treat TransformBegin like AttributeBegin
apply shaders directly to current shape
parse .rib file...
parse .eve file...
XYZ X Y Z XY XZ YZ
model camera currentShape
currentShape is: body

What we're more interested in, though, is in building custom user interfaces for manipulating and visualizing the model we're developing and debugging. So how do we do that? Since WavesWorld runs atop NEXTSTEP, I've extended NeXT's InterfaceBuilder application to allow me to construct user interfaces out of a set of custom objects I designed and implemented (collectively referred to as the **WWTCLKit**). Within InterfaceBuilder, a user can just drag a user interface object from the WWTCLKit palette (each of these will be explained shortly):

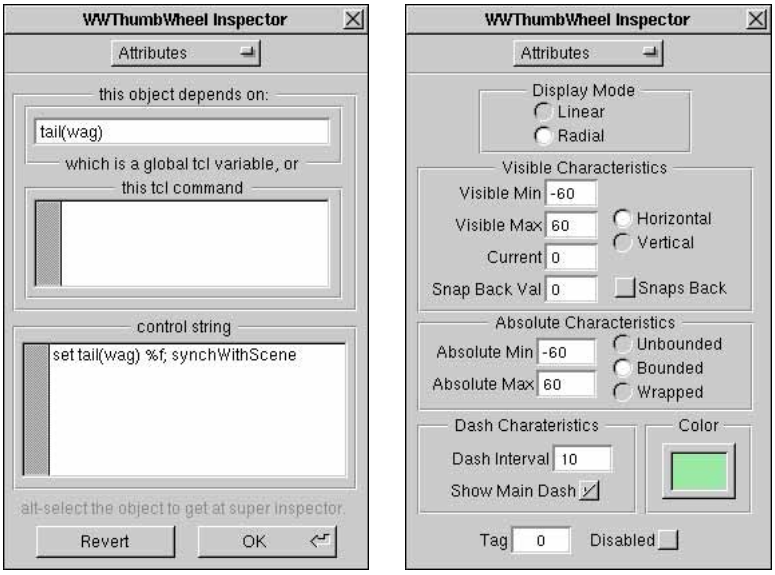


Once dragged from the palette, a GUI object can be dropped onto a Window or Panel and then inspected. Some of the GUI objects can be used for manipulation, some can be used for visualization, and most can be used for both. A slider, for example, can both manipulate some variable's value, and reflect it directly in its on-screen representation. Each object has several inspectors, and one of them allows the user to enter how they want to map this GUI object's on-screen representation to a variable or an expression, and also how this object's value will be mapped onto the model. For example, here's the inspector for a WWThumbwheel object which is attached to the variable in Robbit (see the Examples section later in this chapter) that manages the angle of his tail.

This GUI object, along with other like it, is put on a Window and saved out as a **nib** file (**N**eXT **I**nterface**B**uilder's file format), which is "freeze-dried" version of the GUI objects. This nib file can then be dragged-and-dropped directly into a WW3DWell.

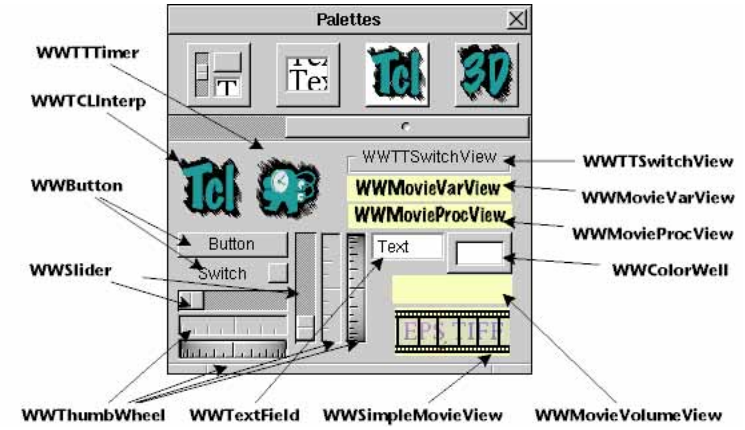
Drag and Drop Debugging

When the nib file is dropped into the WW3DWell, all the objects in the file are "unfrozen" and examined. Each object is asked if it *conforms* to the WWTCLUI protocol, which is a set of messages that all GUI objects in WavesWorld respond to. If it does, the object is asked what expression it is based on. This expression is then analyzed, and for



each variable in the expression, a “variable trace” is instantiated. Each time that variable is updated, this GUI object will be sent a message to resample itself. This is the exact same mechanism that is used when a WW3DKit object is made “animatable”. In addition, for each Window or Panel that has any GUI object on it that conforms to the WWTCLUI protocol, the WW3DWell's run-time system makes sure that if that window/panel is closed, all the variable traces are removed. Also, if the model is removed from the well (by perhaps dropping a new model in), the run-time system can ensure that all the windows/panels are closed when that model is freed. This allows the user complete freedom to close windows/panels themselves, to drag a new model in, etc., while ensuring that “resample” messages don't get sent to objects that have already been freed.

Useful GUI Classes



WWSimpleMovieView
The WWSimpleMovieView allows the display of an arbitrary color and opacity background, a color image (with transparency) and a "movie", i.e. an image sequence running (forward or looping) at some frame rate at some time (always, when clicked, or when the mouse enters). The image can be composited over the movie or the movie can be composited over the image. This can be useful for building general GUIs, or when subclassed (see the WWMovieProcView and the WWMovieVarView below).

WWTTSwitchView
The WWTTSwitchView allows any number of views to be "swapped in" to a given position. This allows things like inspector panels to be easily built, where different facets or parts can be put on different views and switched in as appropriate.

GUI Classes for Manipulation

WWMovieVarView
The WWMovieVarView is a subclass of WWSimpleMovieView. It adds the ability to drag and drop instances on to other instances, where a given instance can be designated as a "source" or "sink" for drag and drop operations. A tcl variable and value can be attached to a given instance. When one WWMovieVarView instance is dropped on another, the receiver sets its variable's value to that of the instance being dropped on it. Drag operations can be restricted so that the type of source and sink must match up.

WWMovieProcView
The WWMovieProcView is a subclass of WWSimpleMovieView. It adds the ability to drag and drop instances on to other instances, where a given instance can be designated as a "source" or "sink" for drag and drop operations. A tcl "proc" name and definition can be attached to a given instance. A tcl proc is essentially a procedure, which maps directly to eve's defineClass: operation. When one WWMovieProcView instance is dropped on another, the receiver sets definition of its proc to that of the instance being dropped on it. Drag operations are restricted so that the name of source and sink's proc must match up.

WWTTTimer
A WWTTimer object asks the run-time system to evaluate some piece of tcl code at some rate for some amount of time. It can also have some code it asks it to evaluate each time it starts up, and each time it quits. The amount of time that it runs can be preset or can be based on a conditional (i.e. the result of some expression that the run-time system evaluates each time the instance wakes up).

WWFishHook
This object attaches to the Physics & Media Group's "fish" sensor, which has (currently) four channels of input. These can be mapped into arbitrary expressions which are sent to the run-time system. This object can run at an arbitrary sampling rate and can be started and stopped easily by attaching buttons to it.

GUI Classes for both Manipulation and Visualization

WWSlider
The WWSlider is a simple slider that can be configured horizontally or vertically. It allows a restricted range of values.

WWTextField
The WWTextField is a simple text entry and display object that can show text of arbitrary color, background, typeface and size.

WWButton
The WWButton is a button which can display text of arbitrary color, background, typeface and size, and two images; one when pressed, and one when depressed.

WWThumbWheel
The WWSlider is a better looking slider that can be configured horizontally or vertically. It allows a restricted range of values, and can be configured to snap back to some predetermined value.

WWColorWell
The WWColorWell allows colors to be dragged and dropped into it, as well as serve as an on-screen representation of an expression that evaluates to a color.

This is by no means a complete list of the GUI objects you would like, but it does allow (especially by using the `WWMovieVarView` and `WWMovieProcView`) very graphically sophisticated GUIs to be built. Some obvious objects that would extend this set nicely would be a suite of curve editing objects that worked in at least one and two dimensions. Because of the rich and powerful development environment in NEXTSTEP, which has been extensively extended by me for WavesWorld, it is straightforward to build new GUI objects.

Reusability: The Packaging Problem

One of the difficult issues in building 3D semi-autonomous animated characters is the problems brought on by collaboration. In almost all cases, a character will be collaboratively designed and constructed; perhaps because there are several people involved from the inception, or perhaps because a character builder is using some set of pre-packaged model parts and behavior parts. One of the big problems in these situations is that of reusability: how can people "package up" models and agents for their collaborators to use? This problem becomes especially difficult when the collaborators have disparate abilities, and are not familiar with the particular tools of the other's domain. This section gives some examples of where these issues might come up, and then discusses mechanisms that WavesWorld has to facilitate collaboration by addressing this question of designing for reuse.

A Quandary in 2D Illustration

Before we how discuss how WavesWorld addresses these problems, let's look at a simpler situation from today's world of desktop illustration.

Two people are working on a piece, an art director and a free-lance illustrator. The art director specs out what she wants the piece to look like, and the illustrator goes off and begins working on the images using his favorite applications, MacroMedia FreeHand® and Adobe PhotoShop®.

The illustrator shows the art director a variety of ideas, from which she picks two. The main element of their favorite illustration is three people seated on a couch. As the art director looks on, the illustrator toys with the facial expressions of the people on the couch, and also plays with their features; first all white, then changing one to Hispanic, one to Asian, one to African, then back again. Some of these changes are chosen from an enumerated set of options that the illustrator skillfully draws ((man, woman), (Hispanic, Anglo, Asian, African)), and some are a range over some extremes (facial expressions going from bemused to concerned, smiling through taciturn to frowning).

It's important to note that this is a two-tier control problem. At the top tier, the art director gives high level direction to the illustrator: "...make her Hispanic. Yea. Make her smile. No, not that sappy. Good. Okay, give him a bemused expression. Yea. Okay. Make him white, maybe a little shorter than her. Good."

The next level of control is the illustrator working directly with the software

application, skillfully using his talents as a creative user to get the application to do what they want. There are no buttons or sliders for “African” or “bemused”; but the illustrator's skills in manipulating colors, blends, splines, lines, perspective, etc. all are successfully brought to bear on the task. If asked, at this point, the illustrator could almost certainly explain the process he went through, although perhaps in a very task-specific way.

Finally, the art director chooses a few options she likes from the many she and the illustrator have just explored, which the illustrator polishes and saves out the two they decided on. She then thanks him and takes delivery of the final set of illustrations, which are both a FreeHand® file and saved out as high quality clip art (i.e. **E**ncapsulated **P**ost**S**cript® files).

She then composes two different versions of the final piece, incorporating the illustration work, and takes them to the client. The client and the art director discuss the piece, and the client makes several suggestions which the art director agrees would make the piece work better in the context the client will be using it. Unfortunately, the changes the client suggested all came up when the illustrator and the art director were discussing the piece, but the particular configuration is not in the Freehand® files that the illustrator left with the art director. Even worse, the free-lance illustrator is now off on another job in another state, and the art director is an Adobe Illustrator® maven, and doesn't really know how to use FreeHand® to make the changes and generate a new piece of EPS clip art to use in the piece.

If only there had been a way to capture the options that they had explored the other day. If only there were a way to encapsulate the variety of constraints (“pull on these splines, change this fill to that, change that blend to this, mask off that, etc.”) the illustrator had skillfully managed. Nothing magical, just some mechanism to allow the two designers—the illustrator and the art director—to “package up” the options and interconnected constraints they had easily explored the other day. No AI-complete solution, no reimplementation of FreeHand®; just some way of treating the static clip art they generated as a more “plastic” material, as a kind of malleable media...

fade to black...

Thinking of Collaborating on A Film Noir

Now let's look at something beyond the scope of the current implementation of WavesWorld—imagine trying to build a character to fit in a scene from a 1940's style film noir motion picture. What components would we need to synthesize such a film?

- script
- actors/characters
- props (costumes, furniture, etc.)
- lighting
- camera work
- direction

Imagine that there are people or programs who are very good at creating these component pieces:

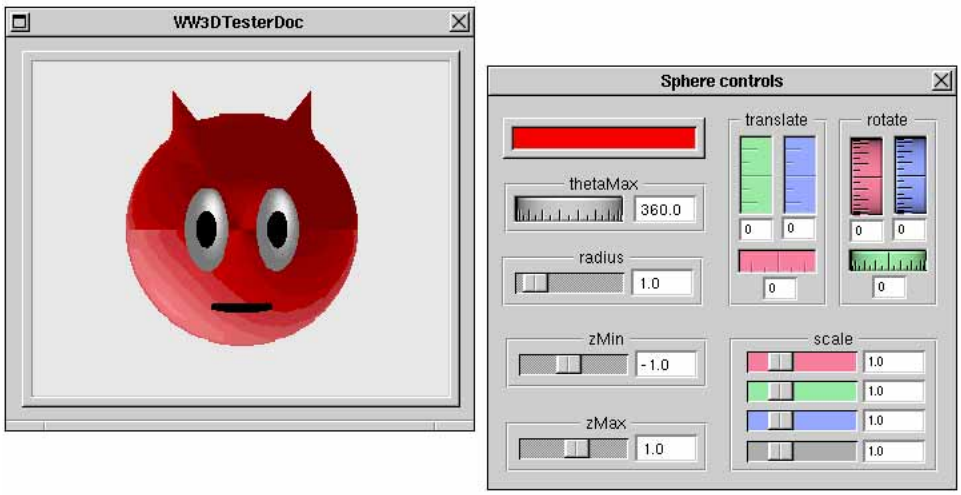
- Would be script-writers, with a gift for hard-boiled detective pulp dialogue.
- People who delight in creating complicated personalities with conflicting desires and goals. Some might enjoy sampling their voice for phrases and words, others might write programs in the spirit of Eliza, trying to model a detective trying to elicit information from a potential client. Others might work on physical simulation of gestures, whether they be facial or hand.
- Modelers who toil endlessly over an exact mathematical model of a 45 caliber Magnum revolver. Others endlessly scan pulp materials and nostalgia kitsch to be used texture maps. Others build fish to swim in aquariums, or finite state machines to mimic pests like flies or cockroaches. Still others spend cycles culling techniques from computational fluid dynamics to simulate (as cheaply as possible) cigarette smoke.
- someone who treats camera motion as an optimization problem, building software for juggling various high level requirements (keep the face in the frame, pan to the right, truck in, etc.).
- someone who enjoys building constraints among some of the constituent parts of the scene, whether in providing physics to the scene, or emotional direction to the various synthetic actors.

How would these people share information? How would they collaborate, especially if they weren't all in the same physical space? How would they collaborate, if some work on the problem early and leave for other projects, while others then need to incorporate their work later on?

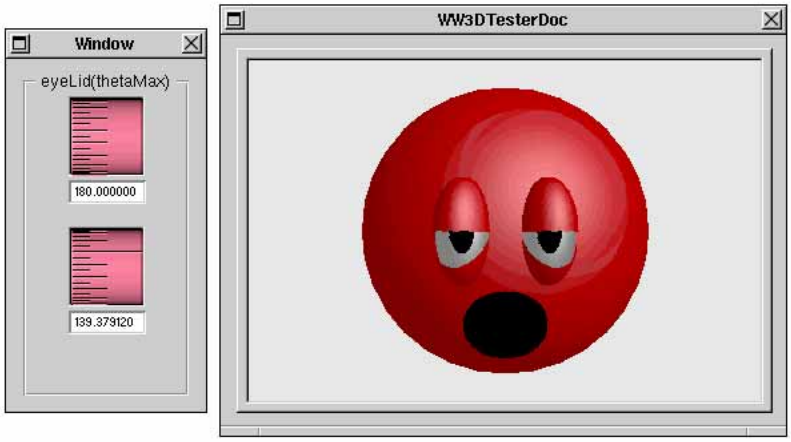
This is exactly the kind of situation that I hope WavesWorld points in the direction of. For today, though, we can only address simpler (but still realistically complex) problems. Let's look at one such situation using the tools in WavesWorld.

Packaging up a Robot: Collaboratively Building a Character in WavesWorld

Now imagine a modeler working on a head model. They would probably start with a simple sphere and its standard GUI controls (included in the large set of Examples that comes with the standard WavesWorld software distribution I make available). They then might add an eye, then duplicate that, and then add a mouth. As we saw in the "Building and Debugging Models" section, it's straightforward to attach sliders and buttons to various variables and procedures in the model. For example, let's say you're working on the head of our robot. You first of all might want to attach sliders, text fields, and a color well to the various parts of the head to manipulate it.

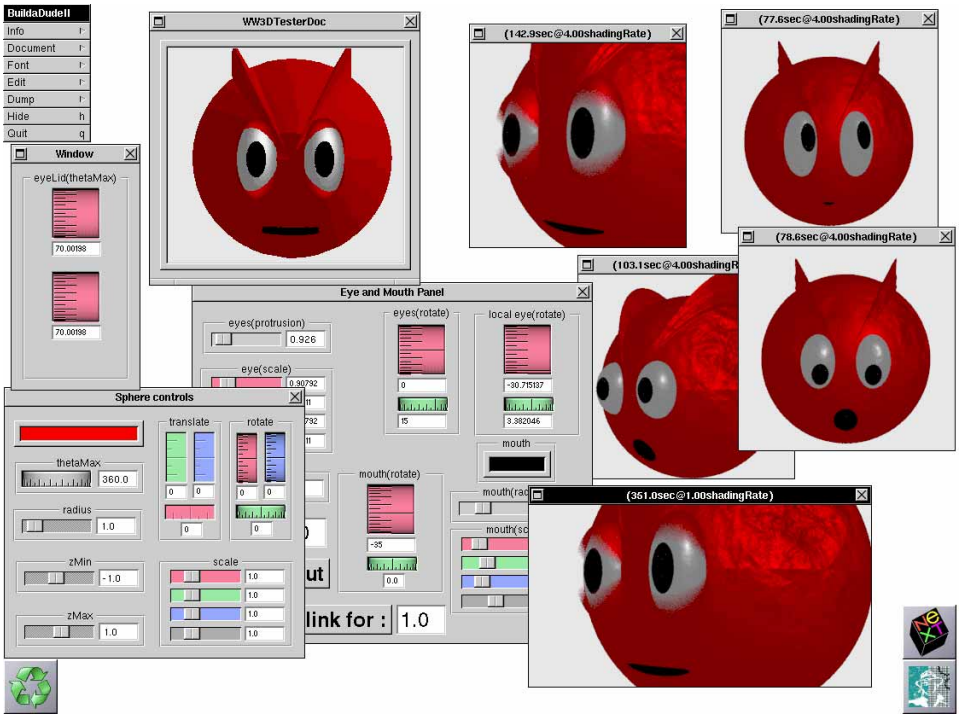


Once you've done that, you might start working on making the eyelids animatable. In order to think about how you might change the eyelids over time, you might start off by attaching a slider to the angles of the eyelids of the eyes of our robot.



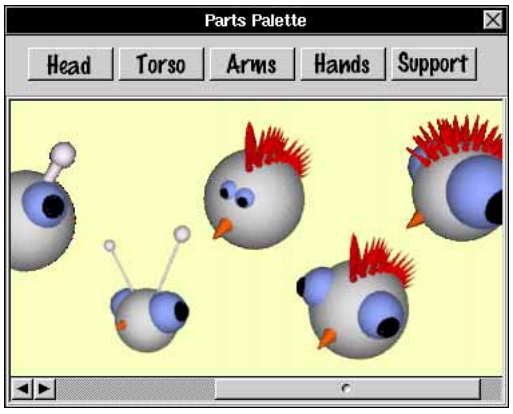
Using these controls, you might develop variations on a simple skill agent and attach

still more GUI elements to attach buttons to make the head perform variations on the motor skill you're developing.

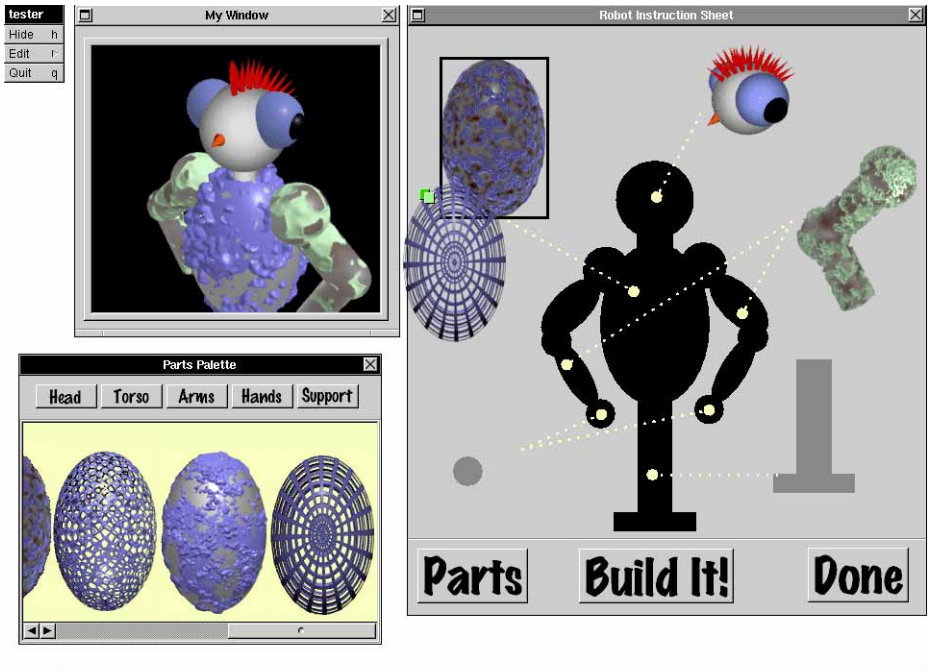


Making Choices Map to Other Choices

Now that you have a basic, functioning implementation of the robot and its environment, you can now iterate over specific portions and improve them. The first thing you might want to do is allow for more variety of shape. You might decide to design several different kinds of heads. Using the tools in WavesWorld, you can quickly and easily build a user interface to allow manipulation of some set of parameters in a head model written in eve. After a bit of experimentation, you might settle on a small variety of heads, such as the following:



Continuing on that tack, you might decide to allow a variety of shape and shading for the various body parts of the virtual actor. Certain combinations of shapes and shading parameters might imply particular values of the shared state of the virtual actor, and might actually change which of several implementations of the skill or sensor agents are used for that particular character. For example, here's a user interface I designed from scratch and implemented in a few hours one evening:



The window on the lower left was a parts palette (containing a `WWSwitchView` and a large number of `WWMovieVarViews`), containing samples of various configurations of body parts. By dragging the `WWVarViews` containing images of the various body parts on to the diagram on the right, the user made certain choices concerning the character under construction (visible in the upper left window). What's especially interesting here is the fact that although each particular choice had obvious mappings to shape and shading ("choose this head; get this kind of shape, choose this torso, get it shaded that way"), it can also be used to control other aspects of the character. For example, if you choose one of the heads with the eyes in "prey position" (i.e., on the side of the head, as opposed to the front), the virtual actor might act in a more skittish or nervous way as it goes about its activities. On the other hand, the fact that you choose one of the peeling, rusted torsos might cause the virtual actor that is built to be somewhat brutish and clumsy, intent on its task with little regard for what it bumped into along the way.

More (Slightly Complex) Examples

This section presents several examples, of varying complexity, of models we've built in WavesWorld. In this section, my emphasis is on the models themselves, and how they are constructed in a way that shows their potential for action. In the next chapter, we'll look at how we might actually take advantage of that potential by manipulating them (via their parameters) over time.

While building models by only directly writing code in eve is possible, it's not usually the way things get done. WavesWorld can directly import RIB® files, which is a popular export format for most commercial modelers. By importing the shapes modeled elsewhere, we can combine these with other elements and proceed to make them animatable. In some cases, the models imported can serve as templates which can be rewritten in eve to become more malleable, using some of the components of the original model (i.e. the *venetian blinds* example, below). In other cases, only some affine transformations (rotate, translate) are needed, and these are straightforward enough to add (i.e. *Dimitri's ring* , below).

Venetian Blinds

One of my hopes for WavesWorld has always been to be able to build the set of a 1940's/film noir private detective piece. One necessary ingredient of any film noir is a fully functioning model of venetian blinds. In 1993, Dan McCoy wrote an excellent plug-in for

Pixar's ShowPlace product on the Mac. The plug-in only generated static RIB files, though. I used the plug-in to generate six different versions of blinds. I saved each out as a RIB® file, and dropped them into a **WW3DWell**. I then saved each out as a .eve model and edited them a bit (taking out extraneous transforms, mainly). I then dropped them back into the **WW3DWell** and, using the shape browser, changed the names of the parts (double click on the shape browser in the WW3DWell's control panel and type in a new name), and saved the models back out again.

I then took one of the slats (a trimmed NURBS surface), dropped it into a **WW3DWell**, and determined how much to translate it to center it about its origin (by selecting "draw origin" in the qrman controls of the **WW3DWell**'s control panel and then manipulating the translate controls until the origin was centered), and added a little GUI to rotate it.

```
defineClass: Slat {angle} {
  AttributeBegin
    ArchiveRecord comment aSlat
    Rotate $angle 1 0 0
    Translate 0 -0.32 -0.015
    TrimCurve { 1 1 1  } { 3 3 3  } \
              { 0 0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 8 0 0 0 1 1 2 2 3 3 4
            4 4 0 0 0 1 1 2 2 3 3 4 4 4  } \
              { 0 0 0  } \
              { 8 4 4  } { 17 9 9  } \
              { 0 0 0.2 0.5 0.8 1 1 1 1 1 0.8 0.5 0.2 0 0 0 0 0.33 0.5 0.67
            0.9 0.67 0.5 0.33 0.1 0.33 0.33 0.5 0.67 0.9 0.67 0.5 0.33 0.1
            0.33  } \
              { 0.01 0 0 0 0 0 0.01 0.5 0.99 1 1 1 1 1 0.99 0.5 0.01 0.29
            0.29 0.29 0.3 0.31 0.31 0.31 0.3 0.29 0.69 0.69 0.69 0.7 0.71
            0.71 0.71 0.7 0.69  } { 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
            1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  }
    NuPatch 3 3 \
            {0 0 0 1 1 1 }  \
            0 1  \
            2 2  \
            {0 0 1 1 } 0 1 \
            P { -0.25 0.318881 0.002822 -0.25 0.322151 0.016410 -0.25
            0.313243 0.027178 0.25 0.318881 0.002822 0.25 0.322151 0.016410
            0.25 0.313243 0.027178 } ;
  AttributeEnd
}
```

I then took off the pelmet (the top part of the blinds assembly) and the batten (the heavy bottom part) and made them into new classes:

```
defineClass: Pelmet {} {

  AttributeBegin
    ArchiveRecord comment pelmet
    Translate 0 -.5 -0.02
    AttributeBegin
      ArchiveRecord comment leftCap
      NuPatch 2 2 \
              {0 0 1 1 } \
              0 1 \
              2 2 \
              {0 0 1 1 } \
              0 1 \
             P { -0.25 0.477500 0.04 -0.25 0.5 0.04 -0.25 0.477500 -0.002500
            -0.25 0.5 -0.002500 } ;
    AttributeEnd
    AttributeBegin
      ArchiveRecord comment rightCap
      NuPatch 2 2 \
              {0 0 1 1 } \
```

```
            0 1 \
            2 2 \
            {0 0 1 1 } \
            0 1 \
            P { 0.25 0.477500 0.04 0.25 0.5 0.04 0.25 0.477500 -0.002500
          0.25 0.5 -0.002500 } ;
    AttributeEnd
    AttributeBegin
      ArchiveRecord comment pelmetBody
      NuPatch 5 2 \
            {0 0 0.25 0.5 0.75 1 1 } \
            0 1 \
            2 2 \
            {0 0 1 1 } \
            0 1 \
            P { -0.25 0.477500 0.04 -0.25 0.5 0.04 -0.25 0.5 -0.002500 -
          0.25 0.477500 -0.002500 -0.25 0.477500 0.04 0.25 0.477500 0.04
          0.25 0.5 0.04 0.25 0.5 -0.002500 0.25 0.477500 -0.002500 0.25
          0.477500 0.04 } ;
    AttributeEnd
  AttributeEnd
}

defineClass: Batten {} {

  AttributeBegin
    ArchiveRecord comment batten
    Translate 0 0 -0.015
    AttributeBegin
      ArchiveRecord comment leftCap
      NuPatch 2 2 \
            {0 0 1 1 } \
            0 1  \
            2 2  \
            {0 0 1 1 }  \
            0 1 \
            P { -0.25 0 0.032500 -0.25 0.005 0.032500 -0.25 0 -0.002500 -
          0.25 0.005 -0.002500 }
   AttributeEnd
    AttributeBegin
     ArchiveRecord comment rightCap
    NuPatch 2 2 \
           {0 0 1 1 } \
           0 1  \
           2 2  \
           {0 0 1 1 }  \
           0 1  \
           P { 0.25 0 0.032500 0.25 0.005 0.032500 0.25 0 -0.002500 0.25
          0.005 -0.002500 }
    AttributeEnd
    AttributeBegin
      ArchiveRecord comment battenBody
      NuPatch 5 2 \
            {0 0 0.25 0.5 0.75 1 1 } \
            0 1 \
            2 2 \
            {0 0 1 1 } \
            0 1 \
           P { -0.25 0 0.032500 -0.25 0.005 0.032500 -0.25 0.005 -0.002500
          -0.25 0 -0.002500 -0.25 0 0.032500 0.25 0 0.032500 0.25 0.005
          0.032500 0.25 0.005 -0.002500 0.25 0 -0.002500 0.25 0 0.032500 }
          ;
    AttributeEnd
  AttributeEnd
}
```

Finally, I put them all together in a model and tweaked and tweaked and tweaked... This model worked fine, and was pretty efficient (it only made the slat rotation animatable), but I could only open and close the blinds—I couldn't pull the blinds up and down. Because the parts were all interconnected, it was difficult to just make a few objects animatable and be done.

This approach (making small components of a model animatable while the rest is left

static) can work for many situations, but when parts are complexly interrelated (i.e. the length of the cords, the position of the pelmet, the angle of the slats as they get closer together, etc.) it breaks down. What we would like to be able to do is encapsulate the model of the blinds into a single object that was animatable. This goal might seem at odds with our desire to be able to interrelate any two given samples of an object, but given that we can only compose such a complex object out of the core objects that conform to the WWRenderable protocol, it turns out that we can do this. It is exactly this power of composition that makes the WW3DKit so powerful. Here's the rest of the model:

```
defineClass: Handle {} {
  AttributeBegin
    ArchiveRecord comment aHandle
    Translate 0.1675 -0.35 -0.033333
    NuPatch 9 3 \
            {0 0 0 0.25 0.25 0.5 0.5 0.75 0.75 1 1 1 } \
            0 1 \
            5 3 \
            {0 0 0 0.5 0.5 1 1 1 } \
            0 1 \
            P { -0.166750 0.334250 0.035 -0.166750 0.334250 0.035750 -
           0.167500 0.334250 0.035750 -0.168250 0.334250 0.035750 -0.168250
           0.334250 0.035 -0.168250 0.334250 0.034250 -0.167500 0.334250
           0.034250 -0.166750 0.334250 0.034250 -0.166750 0.334250 0.035 -
           0.161500 0.319750 0.035 -0.161500 0.319750 0.041 -0.167500
           0.319750 0.041 -0.173500 0.319750 0.041 -0.173500 0.319750 0.035
           -0.173500 0.319750 0.029 -0.167500 0.319750 0.029 -0.161500
           0.319750 0.029 -0.161500 0.319750 0.035 -0.16 0.321750 0.035 -
           0.16 0.321750 0.042500 -0.167500 0.321750 0.042500 -0.175
           0.321750 0.042500 -0.175 0.321750 0.035 -0.175 0.321750 0.027500
           -0.167500 0.321750 0.027500 -0.16 0.321750 0.027500 -0.16
           0.321750 0.035 -0.16 0.335150 0.035 -0.16 0.335150 0.042500 -
           0.167500 0.335150 0.042500 -0.175 0.335150 0.042500 -0.175
           0.335150 0.035 -0.175 0.335150 0.027500 -0.167500 0.335150
           0.027500 -0.16 0.335150 0.027500 -0.16 0.335150 0.035 -0.166750
           0.341750 0.035 -0.166750 0.341750 0.035750 -0.167500 0.341750
           0.035750 -0.168250 0.341750 0.035750 -0.168250 0.341750 0.035 -
           0.168250 0.341750 0.034250 -0.167500 0.341750 0.034250 -0.166750
           0.341750 0.034250 -0.166750 0.341750 0.035 }
  AttributeEnd
}


defineClass: Blinds {count separation angle percentExtended} {

  if {$angle >  75} {set angle 75} {}
  if {$angle <  -75} {set angle -75} {}
  if {$percentExtended <  .16} {set percentExtended .16} {}
  set angle [expr {$angle * [expr {$percentExtended - .16}]}]
  set sep [expr {$separation * $percentExtended}]

  Surface plastic
  Color {1 .991012 .853832}

  Pelmet
  TransformBegin
    Rotate 5 1 0 0
    ArchiveRecord comment handleOffset
    set heightExtent [expr {$separation * $count}]
   Translate  .1 [expr {-1 * $heightExtent/2 - [expr {$heightExtent * [expr
           {1 - $percentExtended}]}]}] -0.02

    Handle
    Translate 0 0 .0025
    Rotate 90 1 0 0
    Color {.8 .8 .8}
    Cylinder .00125 .015 [expr {-.95 * $heightExtent/2 - [expr
           {$heightExtent * [expr {1 - $percentExtended}]}]}] 360

  TransformEnd
```

```
  TransformBegin
    ArchiveRecord comment rightCord
    Color {.8 .8 .8}
    Rotate 90 1 0 0
    # add up all the ys...
    # the batten is .005
    # the pelmet is .0225
    # the space between the last slat and the batten is .5 * $separation
    # the space between the slats is $count * $sep
    # the space between pelmet and the first slat is .5 * separation
    set heightExtent [expr {$separation + [expr {$sep * $count}]}]
    Translate .1 0 0
    Cylinder .00125 0 $heightExtent 360
    Translate 0 -0.015 0
    Cylinder .00125 0 $heightExtent 360
  TransformEnd

  TransformBegin
    ArchiveRecord comment leftInnerCord
    Rotate 90 1 0 0
    Translate -.1 0 0
    Cylinder .00125 0 $heightExtent 360
    Translate 0 -0.015 0
    Cylinder .00125 0 $heightExtent 360
  TransformEnd

  Color {1 .991012 .853832}
  ArchiveRecord comment the slats
  Translate 0 [expr {-.5 * $separation}] 0
  for {set i 0} {$i <  $count} {incr i} \
  {  Translate 0 [expr {-1 * $sep}] 0
     ArchiveRecord comment slat # $i
     Slat $angle
  }
  Translate 0 [expr {-.5 * $separation}] 0

  Batten
}
```

And then finally:

```
set slat(xRotate) 0
set slat(separation) .025
set slat(percentageExtended) 1.0
set slat(count) 10

startShape theBlinds
  animatable: {Blinds $slat(count) $slat(separation) $slat(xRotate)
           $slat(percentageExtended)}
endShape
```

We therefore end up with a model that only has four articulated variables: how many slats compose the blind, the base seperation between the slats, how much they are rotated (in degrees), and how far the blinds are extended (normalized between 0 (not extended at all) to 1 (fully extended)). The first two parameters are not really "animatable", as you probably wouldn't want to change them over the course of a given scene. The last two variables, though, give very nice high level animation controls over the prop, allowing a character to open or close the blind, or to easily perceive how open or closed it is.

Room

At first glance, this room model seems quite basic. What's interesting about it, though, is how malleable and measurable it is. The room can be easily resized, and the elements making up the wall and floor stay the appropriate size. In other words, if we make the walls 1 foot taller, the bricks don't stretch; more fill in. There are several different materials that can make up the floor: wood planks, stone, ceramic tiles, concrete. The walls can be brick, or cinder block, be painted or have wall paper on them. An earlier version had windows and a doorway, but I removed them in the latest version. There are several different pieces of furniture (all the furniture was imported as RIB® from Pixar's ShowPlace CD) that can be put in the room, and they can be arranged in different locations, some constrained to certain areas.

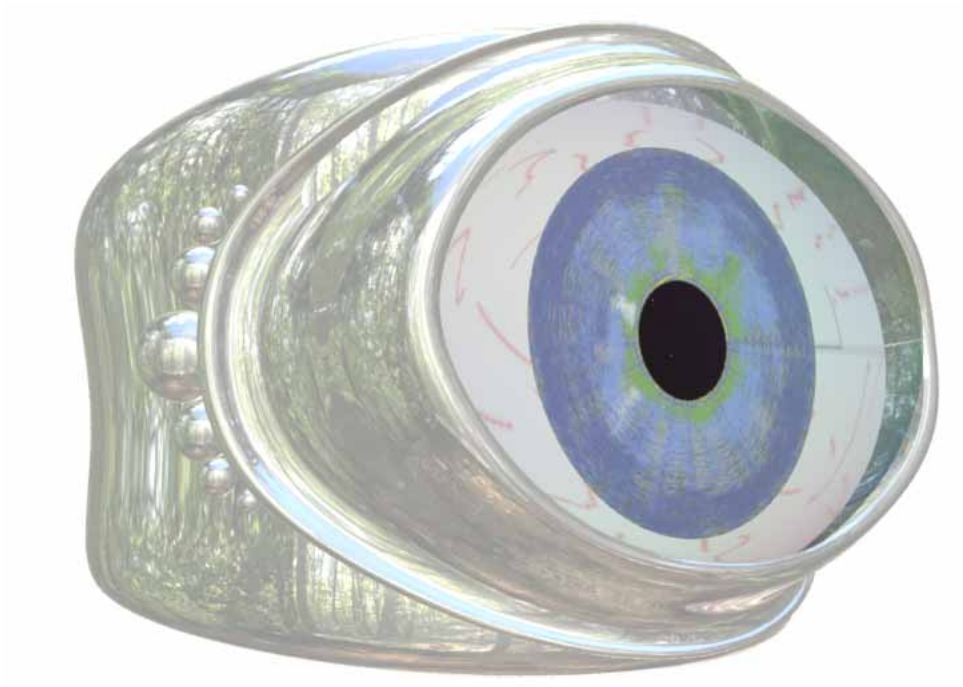
If a character has knowledge of the kinds of materials that might comprise this room, it can perceive quite detailed information about the space. For example, the renderable object (a **`Surface`** object) which draws the cinder block walls has a set of articulated variables associated with it that would allow a character to perceive the base color of the wall, the size and location of the bricks, etc. This means that a character might be told to essentially "trace your finger about the second brick up, 7 bricks over" and it could easily map this into a location on the wall. By building "sense-able" rooms in this way, we can begin to build up a set of prototypes of spaces that we may be able to reuse broadly. Using these prototypes, we can build characters with the appropriate perceptual

mechanisms for sensing these spaces.

Also, the track light model (near the center and top of the frame) was imported and then had a light attached to it. I constrained both to stay on the ceiling (no matter its height) using a `wwSet`. The intensity of the light was constrained to know how high the ceiling was, and modulate itself accordingly so that it would always illuminate the floor. The light can be turned to shine on various parts of the room, and its twist is coupled to what the track light's rotation is.

One interesting side effect of having this single spotlight as the main source of illumination in the room is the fact that it's straightforward to have a character in this room "know" if it is in the spotlight or not. By measuring the intensity and width of the light (both manipulatable and measurable values on the model), a character that wanted to know the intensity or location of the light could easily measure it by a little bit of trigonometry. We'll use this information in the next chapter.

Dimitri's Ring from "Dizzy Horse"



Early last year, I was asked to help with a short (20 minute) film called "Dizzy Horse." It was a story about a young boy named Dimitri and his warm relationship with his grandfather, who used to tell him stories. One in particular concerned a horse on a merry go round, where the other horses chided it for being "dizzy", because it was always looking away from the ride, off to the outside world. Originally, there was to be a wooden sign that would lead the boy through the forest, and the filmmaker wanted to do it digitally. Since this seemed a good test for WavesWorld, I agreed to help.

As time went on, the story changed, and finally it was to be a ring that the grandfather had given Dimitri that would fly through the forest. Also, the writer/director, Gary Cohen, didn't want the ring to have too much "character", he just wanted it to magically move through the forest as a kind of beacon. Also, the visual challenge increased, because the ring was real; the grandfather gives the boy the ring early in the film and it is seen quite clearly by the audience on several occasions. Since the final version would be going to 35mm film, This implied that the CG version of the ring would need to be photorealistic, and it also implied (since the ring would be moving fast) that the shots needed to incorporate motion blur. In other words, this was a perfect opportunity to see if WavesWorld was up to the task I mentioned in Chapter 1: "How can we construct them such that they can be seamlessly integrated into virtual environments containing both real

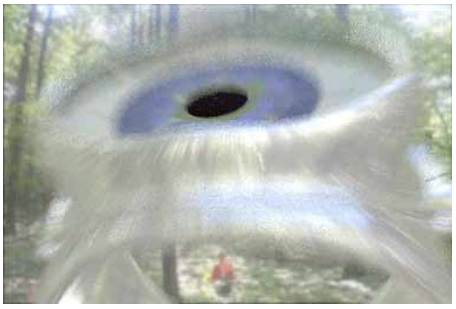
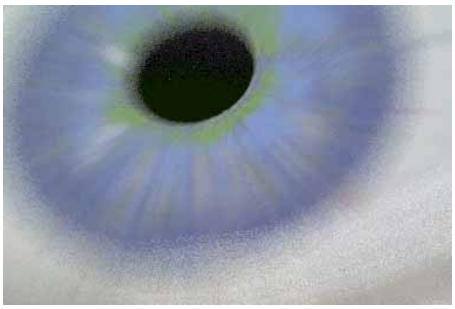
and computer generated imagery at arbitrarily high levels of quality."

The bulk of the original ring was modeled in Alias PowerAnimator by Chris Perry (with kibitzing from me) modeled the basic ring and eye holder in Alias on an SGI. Since we didn't have Alias 5.1 (which has native RIB® export), Rick Sayre at Pixar was kind enough to take the Alias wire file and export it as a RIB® file. I then took the RIB® file and turned it into eve and built a GUI for manipulating and naming the parts. I added the eye as a two disks, one differentially scaled, and a hemisphere. Chris then drew the texture map for the pupil and iris, and I did the blood vessels. I then experimented with a few environment map shaders and tweaked one from (**Upstill89**).

The finished model has 33 articulated variables, but only 6 (X, Y, Z rotation and X, Y, Z translation) were used to animate the ring for the film. The final 6 second sequence that was used for the film was rendered at 1536x1024, 24 frames per second, 1/48 second exposure, composited against 35mm background plates I shot and scanned in via PhotoCD and finally was put on 35mm film at DuArt, NY.

In the first 3 seconds, the ring flies from Dimitri's hand


into the camera,

and cuts directly to this sequence, where the ring flies over and past the camera's "left shoulder" and heads down the path

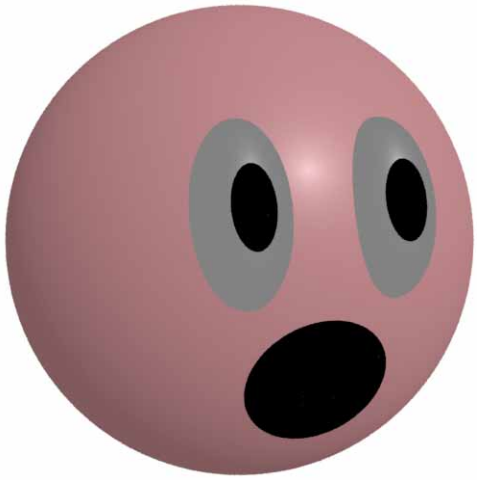

to disappear over the ridge over the course of the next 3 seconds


Note that the camera does a "follow focus" on the ring, so the background starts in

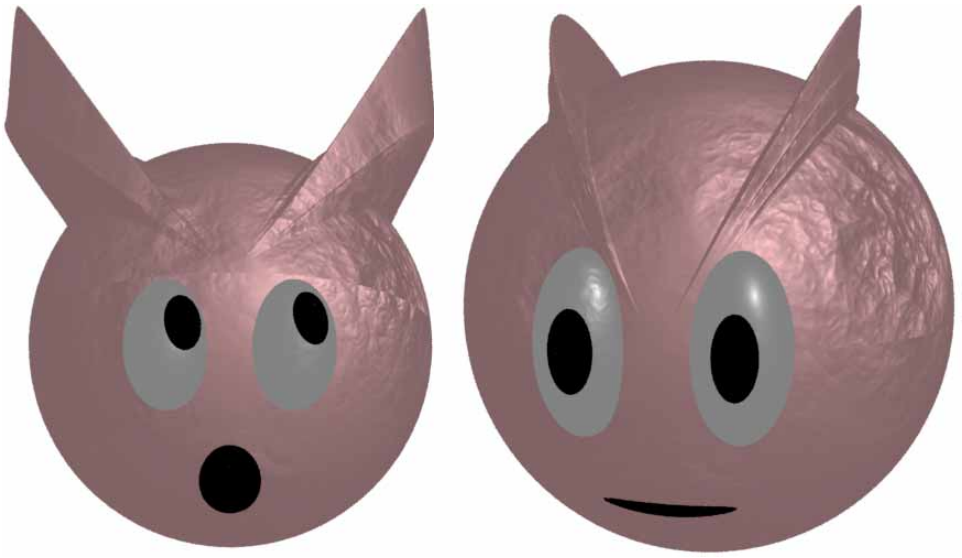
focus, gets blurry as the ring approaches the camera, and then gets more in focus as the ring goes away from the camera.

SphereHead

SphereHead is usually seen as a simple head with a sphere for a head, two eyes (with pupils), and a mouth.



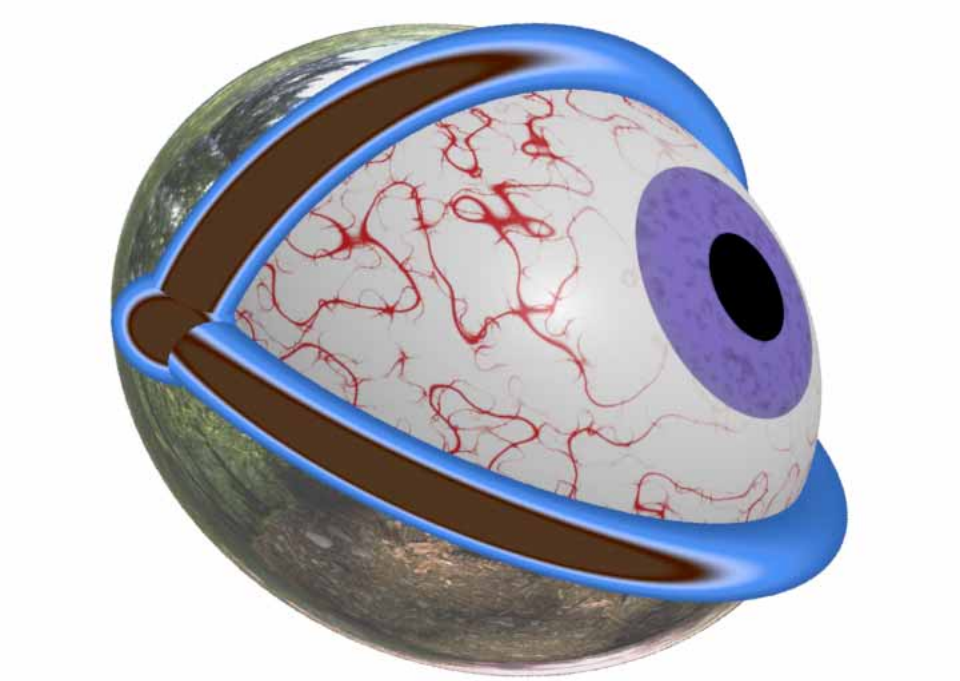
The eye and head can change scale (both preserving and not preserving volume) and rotate, and the mouth can change size and rotate about the body. Depending on the state of a variable, the model may or may not have eyelids. Sometimes another object may be substituted for a perfect sphere, like these two:



SphereHead has at least 34 articulated variables that are useful for animation. Since its shape and shading info is so simple, it's ideal for real-time manipulation and prototyping of behaviors modifying a character's head.

AudreyeII

AudreyeII is simpler than SphereHead but is more interesting to look at because it has several interesting surface shaders attached to it. It's named AudreyeII because its demeanor is reminiscent of AudreyeII from the "Little Shop of Horrors" musical.



Several of the animation possibilities in this model are tied to the surface shader on the eyeball (written by Larry Gritz). The pupil can constrict or dilate, and the amount that the eye is bloodshot can also change. In addition, the model can be made to blink, and the eyeball can look left and right and up and down. Finally, the whole model can be squashed and stretched, and because of the way the scaling is set up, the scaling is skewed in an interesting way.

AudreyeII has about fifteen articulated variables that are useful for animation. As with many of the models presented in this section, AudreyeII has many more degrees of freedom than that, but they're not really amenable to be changed over the course of a scene. For example, changing AudreyeII's location (i.e. the articulated variables associated with its X, Y, and Z position) is quite reasonable over the course of a given scene, but changing the surface shader on AudreyeII's exterior is not. On the other hand, modulating a single parameter of a given shader (which is exactly what we're doing to get the eye more or less bloodshot, or dilate the pupil) is, many times, reasonable.

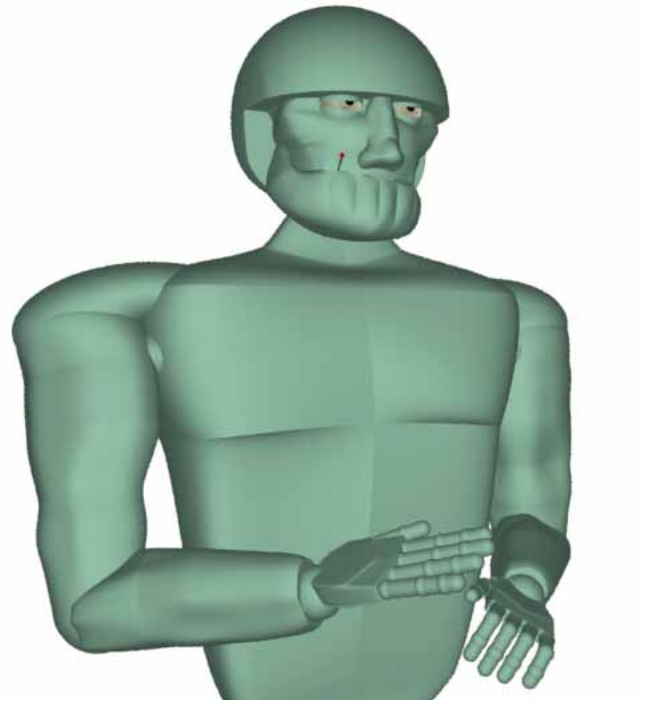
Rocky da Humanoid

One of the driving images in my head for my graduate work is the notion of a film noir private detective slumped over his desk in his office, passed out from the night before, with the neon outside his office illuminating the scene. Several years ago, I convinced a highly talented modeler (Keith Hunter, now head of modeling at Rhythm & Hues in Hollywood), to model a character I called Dexter, from the old Alex Raymond/ Chandler comic strip, Secret Agent X-9.

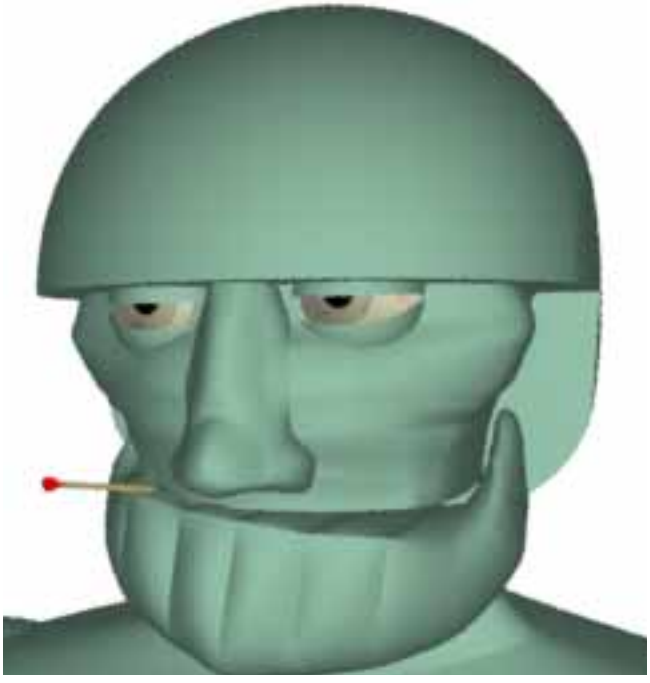
Anyway, Keith whipped out a model for me in a few hours on his Mac, and it served me in good stead over the years. As I built the latest version of WavesWorld, though, it was clear that the polygonal model that Keith had made for me (exactly what I needed at the time), was not going to fly in my current system. Also, since I'm not doing legged locomotion, I didn't need the lower body. A few months back, in preparation for using the data suit (**Bers95**) they built upstairs, I built a reasonable torso and arms (with fully articulated fingers) based on my own proportions so it would easy to map motion capture information on to it:

Well, the torso is nice, but it obviously needed a head to be of any use to me. I talked to my friend Gorham Palmer, a free-lance illustrator, who had done some 3D modeling for me a few years ago, and he was interested in doing a head for me. He made some sketches, which I really liked. We finally sat down a few months ago and made a new

head. Unfortunately, it really wasn't Dexter, so I decided that this was really Rocky, the dumber, much uglier half-brother of Dexter. Rocky is the most complex of the models in terms of degrees of freedom, and consequently he's the most difficult of these models to create behaviors for.



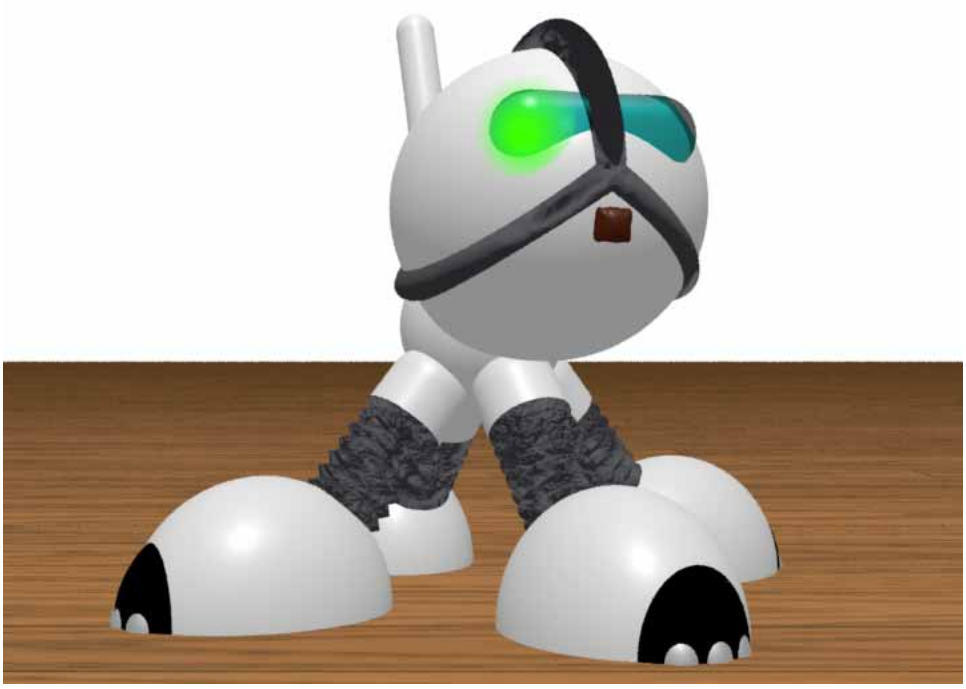
Rocky has 10 fingers, with all the appropriate joints and constrained degrees of



freedom. His wrists, elbows and shoulders can also rotate. He has 3 rotational degrees of

freedom in his neck, his jaw can open, rotate, and slide, his eyelids can open and close, his eyes have all the degrees of freedom of AudreyeII (they use the same eyeball shader with different parameter ranges). Finally, the Rocky model has a prop that is usually attached; a matchstick dangling from his mouth, which can be moved up and down and around the mouth.

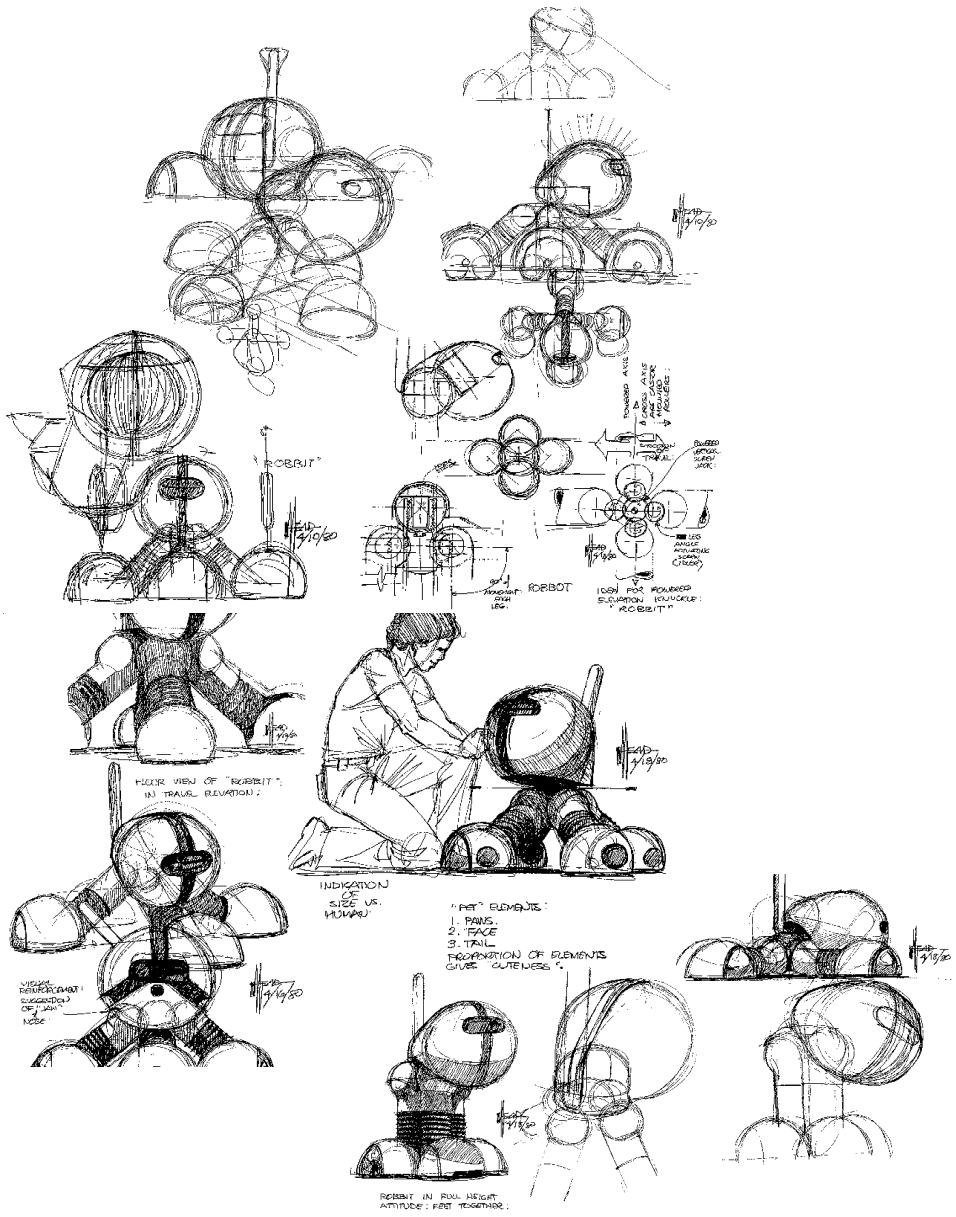
Robbit



The model I call Robbit was directly inspired by several pages of drawings in Syd Mead's book *Kronovecta* (**Mead90**). For those unfamiliar with his work, Mead was the "visual futurist" on such films as *Blade Runner* and *Star Trek: The Movie*. The other models I had done were either designed by myself or done by working in direct collaboration with someone. I was interested in seeing how WavesWorld would fare when used to build up a character based on someone else's design, where there could be no given and take during the specification phase.

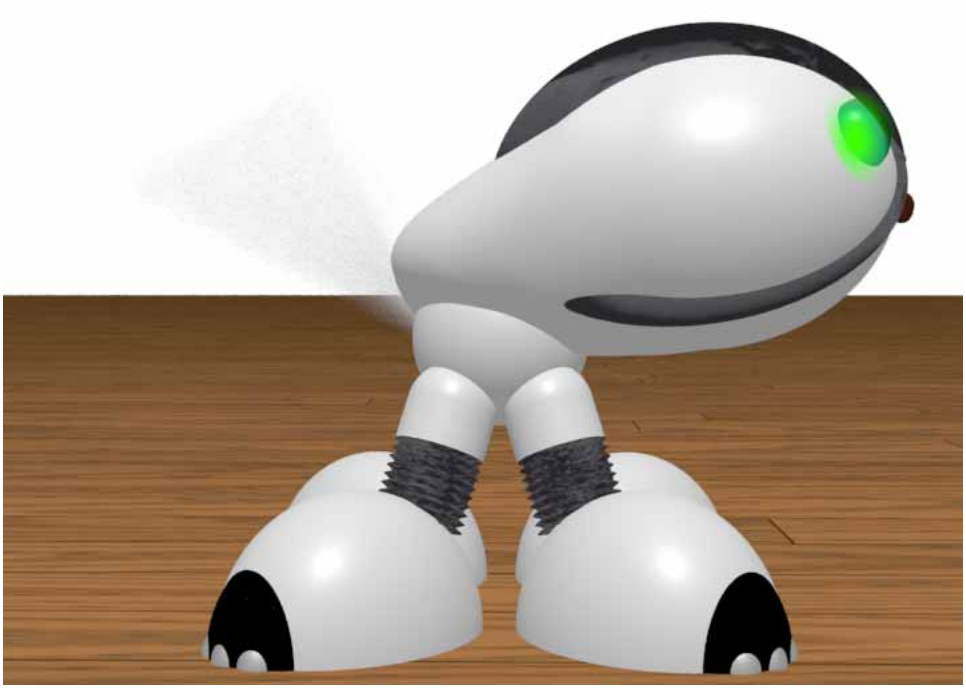
Since my model of Robbit is based directly on Mead's drawings and text, so it's probably best to quote him directly:

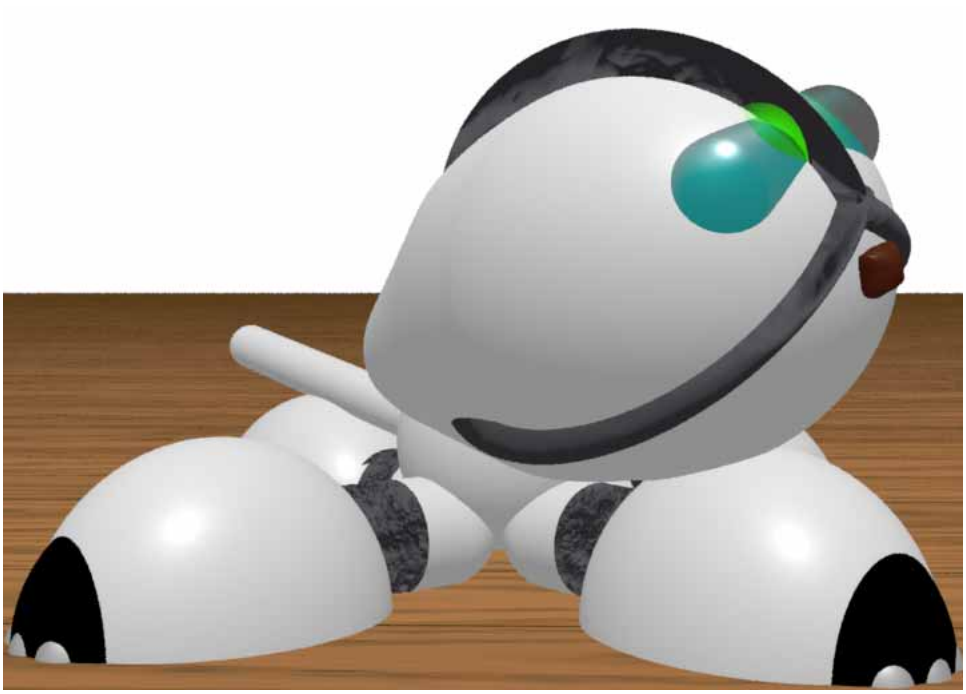
> *"Dogs endear themselves to their owners. Puppies, with their exaggerated attitudes and poses, become irresistible. Combining the clumsy posture of a puppy and slightly rearranging the mechanical components of "dog", the final proposal captured the essence of "pet." Since puppies are all head, legs and feet, this became the guiding rationale. The head was a large, pear-shaped form with a simplified slot at the right position to suggest "face" without venturing into complicated separate features. An "eye" spot was to move back and forth in the slot to provide expression. The head was mounted on the minimal body which, really, was merely a connection element for all of the parts. The four legs and feet were rotated 90 degrees so that the feet contacted the ground in a diamond pattern. This had the delightful result of reinforcing the odd machine look and forcing the "head" to position on either side of the front "foot" when the pet layed* (sic) *down, duplicating the way puppies tend to rest their chin on one or the other front paw. The tail, a simple rounded-end cylinder, was where the batteries went.*
>
> *Motive power was intended to be in the form of powered rollers inside the large*



paws."

Since Mead's drawings were line art, I had to decide on coloration and shading. I decided on a completely clean plastic look to robbit, as if he had arrived fresh from the robot factory that morning. I made the accordion joints and the head pieces seem to be built out of hard rubber, with the intent that the head pieces would act as a bumper, allowing him to bump into things. I also gave him a "button" nose, and changed the eye slot to a more semi-transparent visor-like piece and hence made the "eye" a green glowing sphere. that can move back and forth.





Conclusion

This chapter introduced and explained much of the substrate of WavesWorld that gives it its flexibility and power as a testbed for experimenting with issues surrounding building semi-autonomous animated characters. You should now see how we can construct models containing the shape, shading, and state information that comprise both the static and dynamic elements that compose a character. In the next chapter, we'll finally see how we might go about *animating* these character parts, getting us another step closer to our goal of beginning to understand how to construct 3D, semi-autonomous animated characters.