Chapter 22

Advanced XODUS Techniques: Simulation Visualization

UPINDER S. BHALLA

22.1 Introduction

One of the biggest hazards in developing a simulation is the pressure to make it user-friendly. If you ever make the mistake of making a simulation easy for people to play with and understand, they will suddenly discover gaping holes in your model design, and start to think up all sorts of "improvements" for you to make. An even more unpleasant situation can arise when you have sent off the final page proofs of your simulation paper, and decide that now is a good time to provide it with a colorful display so that people reading your paper can run the simulation themselves. Inevitably, the display will reveal a fundamental bug in the simulation that no one (least of all yourself) would ever have noticed in all the hundreds of lines of simulation code. The prudent builder of simulations will avoid any compromises when it comes to obfuscation. This chapter, then, is for the reckless, since its stated goal is to reveal all, to shine the bright light of day on the hidden crannies of simulations where bugs lurk, and to display the gory details using the rainbow colorscale in an animated three-dimensional draw widget.

22.2 What Can Your User Interface Do for You?

A user interface is a tool for communication. It has a role wherever you wish to interact with the computer, with data, or with other people. Teaching, demonstrating simulations, or facilitating development are all common applications. A graphical interface can be much more than a set of buttons telling a simulation to start and stop, with a graph or two thrown in. You have already met examples of some of the more interesting things one can do with XODUS in the form of the demos such as *Orient_tut* (Chapter 17) and simulation building tools such as *Neurokit* (Chapter 7). In the technical sense, though, there are four main operations for which user interfaces, and XODUS in particular, are designed.

1. **Input**. An interface should make it easier to provide control signals for a simulation, and to assign parameters.
2. **Output**. An interface should simplify the monitoring of interesting aspects of simulation, and especially be able to provide an easily interpreted view of a complex simulation.
3. **Checking for errors**. This includes, but goes beyond, simply monitoring the progress of a simulation to see if it is behaving strangely. Several of the advanced XODUS widgets are designed to help with analyzing the structure of a simulation, so as to check that the simulation is connected up the way you think it is.
4. **Building simulations**. Writing a tool to help users build simulations is about the hardest user interface task. It embodies all of the aspects listed above, and has special challenges all its own.

In the course of working through this chapter you will be introduced to graphical components that are used for carrying out all the above user interface operations. The latter part of the chapter consists of an extended example that uses many of the user interface components to put together a simulation builder. Along the way you will get to see some of the organizing principles of XODUS, and how they all fit into the framework of GENESIS objects, actions, and functions.

22.3 Draw/Pix Philosophy

As long as one is restricted to displaying single data points (dialogs) or arranging widgets on the screen (forms), the graphical interface world is simple. Displaying anything more complex than text or bitmapped images brings in all sorts of complications such as scaling, rotation, managing events, and so on. In XODUS, all these more complex displays are handled by an elite family of widgets called *draw* widgets, and their children, which are *pix* widgets.[1] Even such an apparently simple operation as plotting a graph turns out to be sufficiently complex that it is handled by a specialized version of a draw widget, although most of the complexities are hidden from you.

From the point of view of the user, a draw widget is a window onto three-dimensional space, and a pix is any item visible in that space. Depending on the perspective of the window, a pix may not be visible (for example, it may be off to the side of the window), or it may be occluded (another pix may be in front of it) or it may simply be too small to see (a 5-micron neuron is not easy to find in a window representing a volume of space that is a meter on each side). There is the inevitable tradeoff here: the flexibility in being able to look at three-dimensional objects and do zooms, pans and rotations has to be paid for by specifying more parameters and by having to do more display computations.

There are some parallels between the draw/pix relationship and the familiar form/widget relationship. First, a pix can only be displayed in a draw. Second, in order to be displayed by a draw, the pix must be a child (or descendant) of the draw. Third, the draw "manages" the pix. Just as a widget is at the mercy of the parent form with regard to resizing, hiding and so on, a pix is only displayed according to what the parent draw decides. A basic organizing principle is that any pix can be displayed in any draw. There is a common set of properties for all draws, and likewise a common set of properties for all pixes, which make this work uniformly. To put it in computer science terms, all draws are subclassed from the *coredraw* object, and all pixes are subclassed from the *pix* object. The operations performed by the coredraw object and its subclasses include:

1. Performing coordinate transforms for the pixes. Different draws project pixes in different ways according to the transformations available in the draw widget.

2. Managing events for the pixes. These include all interface events such as mouse clicks, drags and drops, and resize requests. For each mouse event the draw widget must identify the destination pix and pass the event on to it.

Similarly, all pix subclasses carry out the following set of common operations.

1. Managing a set of coordinates for the display.

2. Doing the actual graphical display once the transformations have been completed by the parent draw widget.

3. Dealing with interface events that have been forwarded from the parent draw widget.

[1]If one is really particular about nomenclature, one should refer to pixes as *gadgets* rather than *widgets*, since *gadget* is what the gurus at Project Athena have chosen to call windowless widgets. I do not bother with this distinction.

22.4 Meet the Cast

The forms, buttons and dialogs you have met in earlier chapters are the lower invertebrates on the interface evolutionary tree. The graph widget is actually a highly evolved interface component that has taken to slumming with the "simple widgets," There is even a rumor that it underwent a frontal lobotomy in order to fit in better — brain damage, by any other name. You have had glimpses of some of the advanced widgets in the tutorials such as *Orient_tut* and *Neurokit*. In this section we provide brief sketches of their function and a simple example or two. As always, for detailed information, read the manuals and documentation, and look at the examples.

A note on nomenclature: all XODUS widgets and almost all XODUS-related commands begin with an "x". This preceding "x" is usually omitted in the text, although all the examples have the commands and names in full.

22.4.1 The Draw Widget Family

Coredraw

This is the base class of all draw widgets. The only transformation it knows about is projection in the *x*-*y* plane. It is useful if the pix being displayed is flat and has no business using another projection.

Graph

This is a highly specialized draw widget, whose function you already know. Its projections are also confined to the *x*-*y* plane. It is designed specifically to manage axes and plot pixes, and does so in a very stereotypical and bossy manner, even interfering with their creation and destruction. As you have already seen, it creates child plots automatically when a PLOT message is sent to the graph widget. However, one can explicitly add any pix (including plots and axes) to the graph and do the usual manipulations on them.

Dumbdraw

This exists only as an example for those wishing to understand the C-code implementation of the draw widget hierarchy, and how to manage inheritance from the coredraw widget. It only differs from the coredraw in being able to perform projections in the *x*-*z* and *y*-*z* planes as well as the *x*-*y* plane. If one really needs the *x*-*z* and *y*-*z* planes, one could just as well use the draw widget.

Draw

This is the most general draw widget, providing general orthographic as well as perspective transformations. In the examples that follow we use this as illustrative of all the other draw classes, although they cannot handle all of the transformations that this does.

1. Create a draw widget, specifying the region of space that it displays and a function to execute.

   ```
   create xform /form
   create xdraw /form/draw [0,0,100%,100%] \
       -xmin -5 -xmax 5 -ymin -5 -ymax 5 -zmin -5 -zmax 5 \
       -script "echo Hello from <w>"
   ce /form/draw
   xshow /form
   ```

2. In order to see how the coordinate transformations work in the draw widget, we need to have something to display in it. We will use a simple **xshape**, which is a generic pix for drawing shapes, to draw an open green rectangle:

   ```
   create xshape shape -fg green \
       -script "echo Hello from <w>" \
       -coords [0,0,0][0,2,1][3,2,1][3,0,0][0,0,0]
   ```

3. Let us first investigate the keyboard controls of the draw widget for zoom and pan. In order to use keyboard controls for manipulating a draw widget, the mouse must be positioned on the widget.

   - Move the mouse onto the draw widget.
   - Press the arrow keys on your keyboard. The rectangle should move around on the draw widget in the appropriate directions. These are the pan controls.
   - Press the angle-brackets keys (i.e., the comma and period keys). These should shrink and expand the rectangle, respectively.
   - Return the rectangle to a reasonable size and position for the next step.

4. Now let us look at the transformations provided by the draw widget.

   - You are currently in the *x*-*y* plane, looking down from the *z* axis. So, the current transformation is identified by the “z” key. Hit the “y” key to go into the *x*-*z* plane. The rectangle will probably jump a little, because of the pan operations you did earlier. Its dimensions will also change, because you are now looking at it from another direction.

- Go into the *y*-*z* plane by hitting the "x" key. Now the rectangle turns into a tilted line. This is because you are now looking at it edge-on.
- To get a better feel for the rectangle's orientation in three-dimensional space, we will now go into an orthographic projection. This lets us look at the object from any direction, but without using perspective. Hit the "o" key to go into orthographic mode. The rectangle will jump again, and change shape.
- As before, the arrow and angle bracket keys can be used to manipulate the pan and zoom of the display. Now, however, the pan operations can affect all three axes.
- In the orthographic mode, we can change the viewpoint of the draw widget. The viewpoint is a three-dimensional vector along which the observer looks at the draw widget, and is defined by the fields *vx*, *vy* and *vz*. It is equivalent to the normal to the plane of projection of the draw widget. To rotate the rectangle about the *z*-axis, keep the "shift" key pressed while you press the left and right arrow keys. Check how the *vx* and *vy* fields change when you do this, while the *vz* field stays the same. Conversely, you can also assign values to the *vx*, *vy* and *vz* fields to set up predefined viewpoints from scripts.
- To rotate about the horizontal, keep the "shift" key pressed while you hit the up and down arrows.
- It is very likely that by now you have experienced the Necker cube illusion with the rectangle — it is hard to decide which end of it is near you, and which is further away. This is a drawback of the orthographic projection. So let us go the whole hog, and try the perspective projection. Hit the "p" key.
- The rectangle should change shape slightly, and will also shrink by a factor of 2 or so. Try the rotation operations again. Can you see any effect of perspective?
- The perspective transformation introduces yet another parameter: the distance of the observer from the object. As you get closer to the object, the amount of distortion introduced by perspective increases. If you get too near the object, really strange things might happen as error-trapping code kicks in. Play around with the perspective distortion by hitting the square bracket keys on the keyboard, i.e., "[" and "]", and then watching the effect of rotating the viewpoint.
- Note that in the limit, as one gets further away from the object, the perspective transformation approaches the orthographic projection (except for the scaling factor). Test this by toggling between the "o" and "p" projections.

5. Finally, let's check out the effect of mouse actions on the draw widget and its contents. Events in the draw widget are directed to the nearest pix, if it is within a certain

distance (usually around 10 pixels) of the mouse event. Otherwise the draw widget handles the events itself.

- Click on a line of the rectangle. The rectangle should be transiently highlighted, and a line should appear on the console window saying: `Hello from /form/draw/shape`.
- If you click on a clear area of the draw widget, the event gets directed to the draw. The console now says: `Hello from /form/draw`.

We go into considerably more detail on events in XODUS in a later section of this chapter. Now we proceed to the members of the pix family of widgets.

22.4.2 The Pix Family

pix

This is the base class of all pix widgets, and its only purpose is to act as the family patriarch. If one creates a pix, it will draw a set of cross-hairs, which are not particularly useful, but we demonstrate it anyway. Continuing with the previous example of the draw widget:

```
create xpix pix
```

will put a pix widget in the draw. The pix widget looks the same, no matter what transformation you use. Try them out. When you have convinced yourself, go into the *x*-*y* plane (the “z” key) for the next set of examples.

The pix widget, and all its subclasses, have the following common set of fields:

fg – foreground color. Try it out by setting *fg* to various colors:

```
setfield pix fg yellow
setfield pix fg blue
```

If you load a colorscale, then 64 colors are accessible as numbers from 0 to 63:

```
xcolorscale rainbow
setfield pix fg 0
setfield pix fg 32
setfield pix fg 63
```

tx, ty, tz – translations to be applied to the pix as a whole, to move it around in space. For example:

```
setfield pix tx -1
```

will move the pix by 1 unit in the negative *x* direction.

script – script command(s) to execute in response to interface events. Assign a command to the pix:

```
setfield pix script "echo this is a pix widget"
```

Now click at the intersection of the crossbars to see what happens.

value – a text string with a value associated with the pix. This is often used as an argument for the script command, using the angle-bracket notation discussed in Sec. 22.5.1.

```
setfield pix value "12345"
setfield pix script "echo the value of the pix <w> is <v>"
```

Now click on the pix again.

pixflags – a set of flags that determine many of the properties of the pix including visibility, sensitivity to mouse events, how it handles transformations, highlighting, and so on. The *pixflags* command from the command line lists the options. A typical flag operation is to make the pix invisible:

```
setfield pix pixflags v
```

Then you can flip the flag back:

```
setfield pix pixflags ~v
```

Another common flag is used to turn off the sensitivity to the mouse:

```
setfield pix pixflags c
```

Now it will ignore mouse events.

sphere

This draws a circle. Its main purpose is to serve as a coding example for those wishing to create their own pixes.

```
create xsphere sphere -fg red -r 0.5
```

gif

This loads in a gif-format image file and displays it within the draw widget. It does not attempt to scale the image according to the transformations of the draw widget, but it does reposition the center of the image appropriately. Locate a suitable gif file. There should be one called *xodus.gif* in the *Scripts/examples/XODUS* directory.

```
create xgif gif -filename xodus.gif
```

If you cannot find a gif file, do not despair. The gif example is not needed for any of the following steps.

plot

This makes plots. It is created automatically by the graph widget when it receives a PLOT message, but can be created explicitly and is perfectly capable of handling messages on its own. It is a perfectly normal pix, so it can be created in other classes of draw widgets as well. It has all sorts of options relating to data compression, display modes, and so on, which are illustrated in previous examples and in the **xgraph** documentation and examples. There is a generic pixflags option that is especially relevant to the plot pix. This is the *flush* option, which turns off the forced updating of a pix after an update. When you have a dozen or so plots, it becomes pretty time-consuming to update the display a dozen times per time step.

axis

This makes axes in three dimensions. Two axis pixes are created automatically (for the *x* and *y* axes, respectively) when a graph widget is created. Again, the axis pix can be created explicitly as well, and is at home in any kind of draw widget. One can provide a vector for both the axis direction and for its tick marks, so it is not restricted to providing axes in the *x*-*y* plane.

shape

This is the workhorse pix. It does what its name implies, which is to draw shapes. The **xshape** pix is used extensively as an icon both on its own in a draw widget, and also as a subordinate pix for some of the really powerful pixes described below. We already have an example of the **xshape** up in the draw widget, which we will continue to manipulate here. First, we have to dig it out from underneath the gif widget.

```
setfield shape ty 3
```

One can display text in a shape widget:

```
setfield shape text "I am an xshape"
setfield shape textcolor blue
setfield shape textfont r24
```

The **xshape** has several drawmodes:

```
setfield shape drawmode FillPoly
setfield shape drawmode DrawSegments
setfield shape drawmode DrawPoints
setfield shape drawmode DrawArrows
setfield shape drawmode DrawLines
```

and so on.

The **xshape** manages a set of coordinates in the form of three *interpol_structs* (tables), for the *x*, *y* and *z* coordinates. These can be examined using the usual "`showfield -a`" command. Since the coordinates are stored in standard interpol_structs, one can use several commands for manipulating them. The most obvious is to simply set them:

```
setfield shape xpts->table[0] -1
```

There are many other options for manipulating interpol_structs, some of which are listed in Chapter 18. There is a special coordinate specification mode for **xshapes** which is retained for backwards compatibility and simplicity. It is meant to be used when the shape is being created, but you can also use it later:

```
create xshape newshape -coords [1,0,0][0,1,0][0,0,1]
setfield newshape coords [2,0,0][0,1,0][0,0,1]
```

var

This pix is used to display values graphically. It can vary almost any of the graphical parameters of xshape (e.g., color; coordinates; text *x*, *y* and *z* offsets) to represent changing values. This is quite a complex pix, and we honor it with a long example. The old example is pretty cluttered by now, so let's clean it up a bit:

```
delete /form/draw
create xdraw /form/draw [0,0,100%,100%] \
    -xmin -5 -xmax 5 -ymin -5 -ymax 5 -zmin -5 -zmax 5
ce /form/draw
```

The **xvar** assumes that a colorscale has been loaded. This was done several steps ago. If you haven't been following through all the examples, now is the time to load in a colorscale.

```
xcolorscale rainbow
```

Now create the new **xvar**:

```
create xvar var
```

The **xvar** displays values by interpolating between the relevant display parameters of two subordinate xshapes, which are created by default when the **xvar** is created.

```
genesis #25 > le var
shape[0-1]
```

Before we go any further, let's set up a couple of tables to use as inputs to the **xvar**.

```
create table /tab1 // this table will emit a triangle wave
call /tab1 TABCREATE  2  0  1
setfield /tab1 step_mode 1 stepsize 0.002 \
    table->table[0] 0 table->table[1] 1 table->table[2] 0
create table /tab2 // this table will emit a sawtooth wave
call /tab2 TABCREATE  2  0  1
setfield /tab2 step_mode 1 stepsize 0.001 \
    table->table[0] 0 table->table[1] 1 table->table[2] 2
addmsg /tab1 var VAL1 output
addmsg /tab2 var VAL2 output
```

First, let's look at the default display modes of the **xvar**.

```
reset
step 1000
```

You should see an expanding box, which changes color as it expands. This is the default display mode, called *colorboxview*. If you are the owner of an obscenely fast machine, this example may have whizzed by too fast to see. You can handicap your machine to get it to display at human speeds by reducing the step size of the tables by an order of magnitude, and increasing the number of steps correspondingly.

There are several other built-in display modes, which do fairly straightforward things to the display. Try:

```
setfield var varmode boxview
reset
step 1000
setfield var varmode colorview
reset
step 1000
setfield var varmode fillboxview
reset
step 1000
```

There are far more interesting things one can do with the var widget.

1. At this point you are using the default square shapes that the var created. Let's change the coordinates of the first shape, so that instead of starting out as a small square, the var starts out as a big triangle:

   ```
   setfield var/shape[0] coords [-2,0,0][0,2,0][2,0,0][0,0,0]
   reset
   step 1000
   ```

   You should see a triangle "morphing" into a square. As already mentioned, the way that var displays values is by interpolating between display parameters whose extreme values are specified by the child **xshapes**. In the *fillboxview* mode we saw previously, the original shapes were a small and a large square, respectively. Now the shapes are a triangle and a square. Since the current display parameter refers to the coordinates, the net effect of interpolation is morphing.

2. You can display a value using more than one graphical parameter at a time. Let us use the *y* offset as another such parameter. First we need to have different *y* offsets in the shapes:

   ```
   setfield var/shape[1] ty 3
   ```

   Now we need to tell the var that it should use the information coming in on message VAL1 to display using the *y* offset:

   ```
   setfield var yoffset_val 1
   reset
   step 1000
   ```

   and you see that in addition to changing shape, the display is also bouncing up and down.

3. You can display more than one value at a time. We already have two messages coming in to the var. Let's display the second one using colors from 0 to 63.

   ```
   setfield var/shape[0] fg 63
   setfield var/shape[1] fg 0
   ```

   Note that the range of */tab2* is from 0 to 2. The minimum value is the same as the default, but we have to specify the appropriate upper limit in the *value_max* table.

```
        setfield var color_val 2 value_max[1] 2
        reset
        step 1000
```

Many of these concepts of displaying values in terms of different graphical parameters are used in other pixes as well, such as the **xview** pix and the **xcell**.

view

As a first approximation, this pix is something like an array of **xvars**, but much more efficient. Suppose, in the previous example, you wanted to display 100 versions of */tab1*. If you were to make 100 **xvars**, that would mean 200 child **xshapes**, and each of them would have to be set up independently. Instead, you could use an **xview**. This would require just 2 **xshapes**, but the display would look just the same.

When an **xview** displays the values of many elements it usually arranges all the element icons on the screen based on the three-dimensional coordinates of the elements. These coordinates are specified automatically when using the *path* field option illustrated below, or using the COORD message when using the message-based display options.

An **xview** has a superset of the options available to **xvar**. One of the most important enhancements is the ability to use "paths" as well as messages for sending values to the **xview**. The *path* field specifies a wildcard list of elements that will send values to the **xview**. There are several other options related to the path field, which enable display of subelements from the path, of messages connected to the path, and so on. These are illustrated in the *Orient_tut* tutorial (Chapter 18), and discussed in some detail in the reference manual.

The **xview** happens to be ideal for building a very useful display feature, which I illustrate here:

```
    create xform /scale [200,200,400,100]
    create xdraw /scale/draw [0,0,100%,100%] \
        -xmin -2 -xmax 65 -ymin -2 -ymax 2
    xcolorscale rainbow
    createmap neutral /colorscale 63 1 -object
    create xview /scale/draw/view -viewmode colorview \
        -path /colorscale/proto[] -field x -value_max[0] 64
    reset
    step
    xshow /scale
```

Voila! you have just constructed a colorscale. Try loading in other colorscales to see what they look like:

```
    xcolorscale hot
    step
```

cell

The **xcell** pix, like the **xview pix**, is dedicated to displaying a lot of values at the same time. It is highly specialized for displaying cells, that is, compartmental models of neurons, where each compartment is assigned a diameter and position in three-dimensional space. The reader is referred to the reference manual and examples for further information.

tree

This pix is used to display, edit, and build simulations in XODUS. Typical uses include exploring and manipulating the element hierarchy, managing a library of objects, and building simulations using drag-drop operations. The **xtree** is related somewhat to the **xview** widget, in that it uses a set of subordinate xshapes to represent the simulation components it is displaying. Typically, each class of object is assigned a distinct **xshape** icon. There are several aspects to what **xtree** does:

1. It displays objects or elements graphically. The positioning can be in a tree hierarchy, or based on the three-dimensional coordinates of the elements, or in a grid, or user-defined.

2. It displays messages between elements.

3. It manages graphical interface events, especially drag-drop operations, between elements it displays. In particular, it enables the user to attach script functions to drag-drops between subsets of elements in the display.

Section 22.6 gives an extended example of a network builder using the **xtree** pix.

22.5 XODUS Events

At various points, we have mysteriously mentioned events as something to which XODUS objects respond. To be more specific, events are any user operations that impinge upon the user interface. In XODUS, all events are mapped onto standard GENESIS actions. Not all widgets or pixes recognize all events. Some of them (such as labels and forms) do not recognize any events at all.

All widgets that are sensitive to events have a *script* field (and a `-script` option used with *create*) which, as described in previous chapters, executes functions when something happens to the widget. So far, you have mainly used the default event, a standard mouse click. In order to select a specific event, one can attach a suffix to the end of the script function name. For example, we can set the script for the draw widget to respond to a click of mouse button #2 (the middle button):

```
setfield /form/draw script "echo.d2 This was mouse button 2"
```

Or, we could have it respond to a double click on any mouse button:

```
setfield /form/draw script "echo.D This was a double click"
```

We can specify multiple functions in the script field, and each can be associated with any event. Each such function is separated by a semicolon:

```
setfield /form/draw script \
     "echo.d1 d1; echo.d1 Also d1; echo.d2 d2; reset.d3"
```

Table 22.1 shows the mapping of events to widgets.

	Event	*Action*	*Suffix*	*Widgets*
1	Mouse button click	B1DOWN etc.	none; or d, d1, d2, d3	all
2	Return to off state	B1UP etc.	u, u1, u2, or u3	toggle
3	Mouse double click	B1DOUBLE etc.	D, D1, D2, or D3	all
4	Update (internal)	XUPDATE	Not available	all
5	Keypress	KEYPRESS	k	dialog
6	Drag from (called from source widget)	XODRAG	y	draw/pix
7	Drop into (called from dest widget)	XODROP	p	draw/pix
8	Drop into (called from source widget)	XOWASDROPPED	w	draw/pix
9	Script access	XOCOMMAND	c	shape

Table 22.1 Mapping of XODUS events to actions.

22.5.1 Returning Arguments to Script Functions

A very powerful feature of the script-calling syntax is the ability to pass specific arguments to the function. These arguments are passed in using angle brackets. All widgets that have a script function can take the `<widget>` argument, usually abbreviated `<w>`. This returns the pathname of the widget that called them. In addition, all widgets that have a *value* field (or in the case of the toggle widget, a *state* field) can also pass the value using the `<value>` (or `<v>`) argument. The dialog widget can pass individual keypresses in the `<k>` argument. When one is doing drag-and-drop operations, one needs to keep track of the source and destination widget and their values. These are identified by the `<s>`, `<d>` and the `<S>`, `<D>` argument pairs, respectively. Finally, draws and pixes can all pass coordinate arguments: `<x>`, `<y>` and `<z>`. Most of these are illustrated in the network builder example that follows.

22.6 Using Advanced Widgets: A Network Builder

As a working example of many of the features we have discussed so far, we will build *Netkit*, a skeleton interface for constructing network simulations. At a first pass, you can see how the various XODUS components work together. At a more advanced level, particularly if you ever plan to build your own interface, it may be worthwhile to note how the interface has been modularized. This modular design has been developed for several recent simulation tools including *Neurokit2* and *Kinetikit*, and will enable you to merge your interface with these existing ones and make use of some of the powerful tools such as the parameter search genie. The network builder is designed to do the following operations.

1. Manage a set of prototype cells using an **xtree** element.
2. Drag prototype cells into an editor window.
3. Double-click on cells to edit them.
4. Click-and-drag between cells to set up connections.
5. Drag cells to a graph window to set up plots.

To avoid cluttering up the chapter with non-XODUS related code, the cell models have been relegated to Appendix B. Here we only go over the interface-specific portions of the network builder. If the prospect of typing in all this code (although it is under 150 lines) is intimidating, you may retrieve it from the *Scripts/examples* directory of the GENESIS distribution. The example presented here is organized in a stepwise progression suited to a tutorial. In the example files, and in real simulations, one would organize it much more systematically in the form of modules related to specific components of the interface.

22.6.1 The Library Window

To start off, we will set up the library window and the **xtree** that manages the prototypes. In this case, the **xtree** is doing two things of note. First, it is arranging the cell prototypes for you, by putting them into a grid, i.e., a rectangular array. These prototypes are identified here using the wildcard path, assuming that all objects of type **compartment** under the */proto* element are cells. Second, the **xtree** specifies a function call (*create_cell*, which we will write later) that will be invoked when the prototypes are dragged into the work window.

```
create xform /edit [0,150,500,550]
xshow /edit
create xcoredraw /edit/lib [0,0,100%,30%] \
        -xmin -3 -xmax 3 -ymin -1  -ymax 2
    create xtree /edit/lib/tree -path "/proto/##[TYPE=compartment]" \
        -treemode grid \
        -script "create_cell.w <d> <S> <x> <y>"
```

22.6.2 Making Prototype Cells

In order to give the library something to display, let us set up a couple of cell prototypes. First, the excitatory cell. Because there is a lot of uninteresting (at least from a graphical perspective) code involved in setting up a cell model, we will just load in the cell, using the *make_cell* function defined in the *cellproto.g* file. This puts it under the */proto* element which is always created by default.

```
    include cellproto.g
    make_cell // This loads a cell into /proto
```

Alternatively, if you can't find *cellproto.g* and are eager to proceed, just create a dummy cell by typing:

```
    create compartment /proto/cell
    create synchan /proto/cell/glu
    create synchan /proto/cell/GABA
    create spikegen /proto/cell/axon
```

This will not spike, of course, but for now it will get things started. Now we can set about turning this into an excitatory cell for our interface. First, we need to use the *move* command to rename the cell:

```
    move /proto/cell /proto/ecell
```

Now we add some extended fields to the cell to handle user-interface information. The *transmitter* field, obviously enough, indicates what transmitter type the cell uses. The *xtree_fg_req* field is a special field name that the **xtree** widget recognizes, and is used to assign a color to the cell icon.

```
    addfield /proto/ecell transmitter
    addfield /proto/ecell xtree_fg_req
    setfield /proto/ecell transmitter glu  xtree_fg_req green
```

Well, that was easy enough. Now we make a copy for an inhibitory cell, which is just the same as the excitatory cell except for the color and the transmitter:

```
    copy /proto/ecell /proto/icell
    setfield /proto/icell transmitter GABA xtree_fg_req orange
```

Where, you might ask, are the cell icons that the **xtree** widget is supposed to display? The widget needs to be explicitly told to go and update its contents, using the *call* command to invoke its RESET action:

```
call /edit/lib/tree RESET
```

Now we have a couple of cells in the library. They get assigned a nice boring rectangular icon by default; this is the *shape* child element you would have seen if you did an "`le /edit/lib/tree`". We can make them appear a little more interesting by creating an **xshape** as an icon for them:

```
create xshape /edit/lib/tree/shape -autoindex \
    -value compartment -pixflags v -pixflags c \
    -coords [-0.5,0,0][0,0.5,0][0.5,0,0] -drawmode FillPoly
call /edit/lib/tree RESET
```

Well, I only said a *little* more interesting. If you feel creative, you can put any fanciful shape you like to act as an icon for the cells. Note that the **xtree** widget uses the *value* field of the **xshapes** to decide which object classes the **xshapes** are meant to represent.

22.6.3 The Work Window

The next step is to set up the work window. In this case, the tree uses the *geometry* treemode, which displays elements according to their position in space. There are two function calls handled by this tree element. First, it calls a function to edit the cells when they are double-clicked (*edit_cell*). Second, it handles repositioning of cells in space by click-and-drag operations (*move_cell*).

```
create xcoredraw /edit/work [0,5:lib,100%,65%] \
    -xmin -4 -xmax 4 -ymin -4  -ymax 4
create xtree /edit/work/tree -path "/net/#[TYPE=compartment]" \
    -treemode geometry \
    -script "edit_cell.D <v>; move_cell.w <d> <S> <x> <y>"
copy /edit/lib/tree/shape[1] /edit/work/tree
```

We are almost ready to do the first of our *Netkit* operations: dragging cells from the library into the work window. We just need to write the *create_cell* function which the */edit/lib/tree* widget will use:

```
create neutral /net // The new cells are created on top of /net
function create_cell(dest,srcval,x,y)
    str dest,srcval
    float x,y
    // Make sure that the destination is the work window
    if ({strcmp {dest} "/edit/work"} == 0)
```

```
            copy {srcval} /net -autoindex
            position ^ {x} {y} I
            call /edit/work/# RESET
        end
    end
```

Note the use of the wildcard symbol "#" so that the RESET action will be called for all elements directly below */edit/work*. This is all you need in order to drag the prototype cells from the library into the work window. Go ahead and try it! If you list the child elements of */net*, (type "`le /net`") you will see all the cells you have created, which of course correspond to the ones visible in the work window.

By now you should have dragged in enough cells to clutter up your work window, so it is time to write a little function that lets you reposition them. This function is very similar to the last one.

```
    function move_cell(dest,srcval,x,y)
        str dest,srcval
        float x,y
        // Make sure that the destination is the work window
        if ({strcmp {dest} "/edit/work"} == 0)
            position {srcval} {x} {y} I
            call /edit/work/# RESET
        end
    end
```

22.6.4 Editing Cells

At this point, the cells you have created are just colored triangles in the work window. To make them a little more useful you have to be able to edit their fields. This requires three things: a parameter editor window, a function to update parameter values, and a function that calls up the parameter editor when a cell is double-clicked. In addition, there is a utility function *set_named_field* whose purpose is obvious. None of these functions is particularly exciting, so I present them without further comment.

```
    function make_xedit_cell
        create xform /parmedit/cell [500,500,500,150]
        addfield /parmedit/cell elmpath -description "path of elm"
        pushe /parmedit/cell
        create xdialog name -title "Name"
        create xdialog Em [0,0:name,50%,30] \
            -script "set_named_field <w> <v>"
        create xdialog Vm [50%,0:name,50%,30] \
            -script "set_named_field <w> <v>"
        create xdialog inject -script "set_named_field <w> <v>"
        create xbutton UPDATE [0%,0:inject,50%,30] \
             -script "do_update_cellinfo"
        create xbutton HIDE [50%,0:inject,50%,30] \
            -script "xhide /parmedit/cell"
        pope
    end
    create neutral /parmedit // placeholder for param editors
    make_xedit_cell // actually make the cell param editor

    function set_named_field(widget,value)
        str widget,value
        str elm = {getfield {widget}/.. elmpath}
        str field = {getfield {widget} name}
        setfield {elm} {field} {value}
    end

    function do_update_cellinfo
        str cell = {getfield /parmedit/cell elmpath}
        setfield /parmedit/cell/name value {cell}
        setfield /parmedit/cell/Em value {getfield {cell} Em }
        setfield /parmedit/cell/Vm value {getfield {cell} Vm }
        setfield /parmedit/cell/inject value {getfield {cell} inject}
    end

    function edit_cell(reac)
        str reac
        setfield /parmedit/cell elmpath {reac}
        do_update_cellinfo
        xshowontop /parmedit/cell
    end
```

Now you should be able to double-click on any of the cells and get a window with the vital statistics of the cell. If there is any interface-related lesson to be learned from this editor function, it is that the boilerplate code, which does really boring and basic stuff, takes up more space than all the interesting code put together.

22.6.5 Connecting Cells

To prove my last point, here we set up the function that does the far more interesting operation of connecting cells. First, the function itself:

```
    function connect_cells(A,B)
        str A,B
        addmsg {A}/axon {B}/{getfield {A} transmitter} SPIKE
        call /edit/work/# RESET
    end
```

Then we set up the work tree to invoke the connect function. The arguments here include source and destination paths, valid message types and destination element types, a color for the arrows (which this particular tree does not actually have to deal with), a couple of flags, and then three script functions that are called in different situations. The syntax for the following example is explained in gory detail in the GENESIS Reference Manual.

```
    call /edit/work/tree ADDMSGARROW "/net/##[TYPE=compartment]" \
        "/net/##[TYPE=compartment]" all compartment green 0 0 \
        "connect_cells.p <S> <D>" "" ""
```

There is a little complication here. In most situations, one would use a single tree to display elements, call functions to set up interconnections, and display the interconnections as arrows. If that were the case, we would be done by now. In this example, however, we want to interconnect cells, but the actual messages are between the axon and synapse child elements of the cells. So we create another **xtree** to display the messages as red arrows, and change its default *shape* icon to draw points so that it doesn't clutter up the screen.

```
    create xtree /edit/work/arrows -path "/net/#[]/##[]" \
        -treemode geometry \
        -namemode "none" \
        -pixflags c
    call /edit/work/arrows ADDMSGARROW \
        all all all all red 0 0 "" "" ""
    setfield /edit/work/arrows/shape[0] drawmode DrawPoints
```

Now you can connect cells by the simple operation of clicking on cell A, and dragging it onto cell B. A red arrow appears between the two to represent the connection.

22.6.6 Plotting Cell Activity

By now you have a skeleton simulation builder, but it doesn't actually *do* anything! One possible way of watching what the simulation was doing would be to put an **xview** widget in the work window as well, and use colors to represent cellular activity. Here, though, we will draw graphs instead. All we need to do is set up a graph window, and write a function to generate a graph when a cell is dragged from the work window into the graph window. Drag in a few cells to see how it works.

```
    create xform /graph [550,0,500,500]
    create xgraph /graph/graph -hgeom 100% -ymin -0.1 -ymax 0.25 \
        -xmin 0 -xmax 0.5 -yoffset 0.15 \
        -script "add_plot.p <S>"
    xshow /graph
```

```
function add_plot(src)
    str src
    str srcname = {getfield {src} name} @ "_" \
        @ {getfield {src} index} @ ".Vm"
    create xplot /graph/graph/{srcname} -pixflags f
    addmsg {src} /graph/graph/{srcname} PLOT \
        Vm *{srcname} *{getfield {src} xtree_fg_req}
    useclock /graph/graph/{srcname} 2
end
```

22.6.7 Running Netkit

Almost everything is in place now. We still need to set up a few basic simulation parameters:

```
setclock 0 50e-6 // a 50 usec time step
setclock 1 1e-3
setclock 2 0.2e-3
```

If you want a quick preview, you can do the following:

```
reset
step 0.5 -time
```

To round off the network builder, let us put in a control panel as a final amenity. As with the cell parameter editor, this is also boring old boilerplate code.

```
function update_time
    setfield /control/currtime value {getstat -time}
end
create xform /control [0,0,500,100]
create xbutton /control/start [0,0,33%,30] \
    -script "step 0.5 -time"
create xbutton /control/reset [0:last,0,34%,30] \
    -script reset
create xbutton /control/stop [0:last,0,33%,30] -script stop
create xdialog /control/currtime \
    -label "Current time (sec)" -value 0
addaction /control/currtime PROCESS update_time
useclock /control/# 1
create xbutton /control/quit -script quit
xshow /control
```

And that is it for our network builder example. If you have correctly followed the steps outlined above, you should be rewarded with a display similar to that shown in Fig. 22.1. As promised, you now have a simulation-builder interface that manages a set of prototypes, uses click-and-drag operations for creating cells, connecting them and graphing them, and gives you access to your cell parameters.

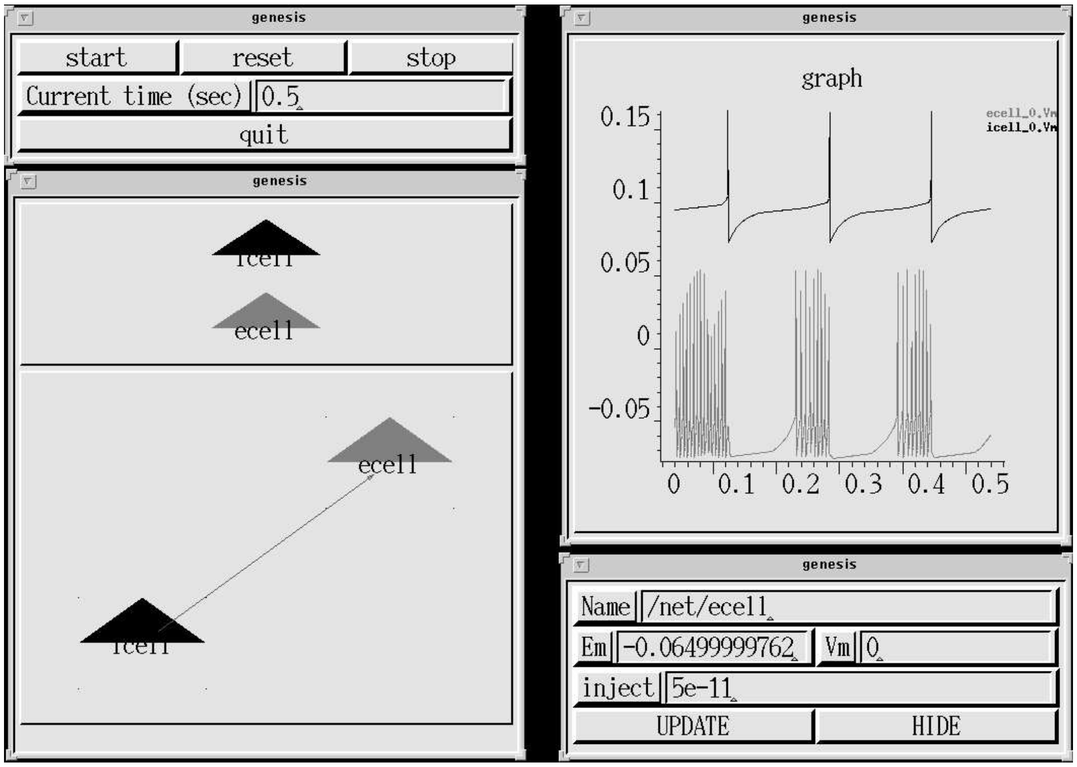

Figure 22.1 The display produced by the completed *Netkit* example. The general simulation controls are located in the upper left-hand corner. The edit window, containing the draw widgets for the library and work areas, is below them. The library widget contains icons for the excitatory (ecell) and inhibitory (icell) cell prototypes. The work widget contains the actual network. The graph widget is displayed on the upper right, and the parameter editing window (currently displaying the excitatory cell in the network) is below this. The simple network illustrated here has an inhibitory cell firing at its basal rate, connected to an excitatory cell that is being driven by 50 *pA* of injected current.

22.6.8 Extending Netkit

The version of *Netkit* you have just written is a very limited one, although it does a remarkable amount for under 150 lines of code. Although the tutorial format of the example blurs the organization somewhat, it should be apparent that the interface can be rather neatly divided into the general interface modules (like the edit and graph windows) and the component modules that handle the cell-specific operations of prototyping, interconnection and parameter editing. If you wanted to add an entirely new module, say, a module for delivering a repetitive stimulus, you would simply have to add a file with the functions for these three operations. This organization is clearly visible in the *Netkit* example files in the GENESIS distribution. The structure of the code should make it fairly clear how it could be extended:

1. There are obvious enhancements to the overall interface, such as a save and restore option, more control over simulation parameters such as clocks and the run time, and so on.

2. There are a whole slew of other network components that are not represented, for example, stimulus objects, globally applied neuromodulators, and so on. Each of these could be built up in the same sort of framework used for the single cell module.

3. The editing interface for the cell is presently very limited. There are a lot more parameters one might wish to change, such as the channel kinetics, synaptic weights, etc.

4. This leads towards a fundamental question: how should this network builder interface be linked into other related interfaces? For instance, for making and editing the cell prototypes one would want a complete neuron-building interface such as *Neurokit*, and for hooking together populations of cells perhaps we need yet another level of network interface.

I leave this example with these little "exercises for the reader." The objective is not so much to lure you into an endless exercise in improving *Netkit*, which will probably be carried out by dedicated hackers long before you read this chapter. It is rather to get you thinking in terms of interfaces as modular constructs, and to see how the building blocks of XODUS can themselves be lumped into larger prefabricated modules that slot together to make pretty powerful interfaces.

22.7 Interface vs. Simulation

The most important part of an XODUS interface is the simulation. We have already stressed modularity in building an interface. The first step towards this is keeping the simulation and the interface separate. This applies both to the GENESIS code (scripts) that sets up the simulation, and to the hierarchy of the simulation itself. In other words, write separate functions for building the graphics and the simulation, and try to keep your simulation components on different element hierarchies from your interface components. This is important both from the viewpoint of managing the code, and also for efficiency. Graphics add overhead, no matter how efficiently you do them. In fact, graphics can easily consume more computer resources than the simulation itself. There are a few key points to keep in mind to minimize this overhead:

1. Although it may seem strange advice in a chapter on interfaces, one of the first things you are likely to want to do once you are in "production mode" with a simulation is to turn off the interface. This is a good reason for keeping the interface and simulation separate!

2. Use slow clock rates for the graphical components. You rarely need to keep tabs on values more often than once in a hundred or so numerical time steps.

3. Use messages for passing data to widgets if they are being updated continuously. The draw, plot, dialog, var, view and cell widgets can all use messages as well as script value assignments. Messages are the native construct for shunting information around in GENESIS and are much faster than script-based updates. Messages are also readily parallelized in GENESIS, which may be important when you decide to run your simulation on multiple machines.

22.8 Summary

In this chapter you have met a menagerie of widgets, with a few examples for the simpler ones. We have seen how mouse and keyboard operations relate to actions and functions in the simulator. We then worked through the *Netkit* example where we saw several of the widgets in action, and learned about some basic design principles for XODUS-based interfaces. Beyond all this, the chapter has also had a hidden agenda, which may have been obvious to you. It is to persuade you to take the leap from learning about simulations, and trying out other people's models, to building your own. Having come this far, you have been exposed to a spectrum of neural simulation lore, ranging from theory and basic physiology to the specifics of GENESIS simulation objects. These are all just tools. Now go ahead and use them.