Chapter 7
Multimedia Nuts-and-Bolts

It should come as no surprise that Squeak, the language for the Dynabook, should be rich in support for multimedia. Squeak already has lots of support for audio and graphics, including standard formats such as AIFF, WAV, MIDI, GIF, JPEG, VRML, MIDI, and several other acronyms. It has a terrific 3-D rendering engine (by Andreas Raab), and a powerful new scripting environment for 3-D (by Jeff Pierce). Its networking support is excellent, including web browsing and serving, email, and IRC. As befits an environment where the goal is to explore new kinds of media, there are some new formats that are Squeak-specific that are quite exciting.

The format for this chapter will be less tutorial than in past chapters. At this point, you have seen and worked with Squeak code, and are comfortable manipulating Morphic user interfaces. You know how to take an example and figure out how to use it in your own way. This chapter will be an overview of the various multimedia capabilities in Squeak. The presentation will point out the pieces—the "nuts and bolots"—leaving it to you to dig in for the details.

The pattern of presentation will be much the same as in previous chapters: Concrete before abstract. In each media format discussed, existing Morphic tools that support that media are presented first, then some Squeak code that provides access to the underlying functionality is described.

In general, you should be in Morphic for all of this chapter. Relatively few of the multimedia tools work well in MVC, though most of the underlying functionality will work in either. Morphic is the future of Squeak interfaces, so most new development is occurring there.

1 Text

Most people don't think about text as the very first thing to deal with when they talk about multimedia, but text is still the most important medium for communication on computer displays. Good multimedia often requires good text. Squeak's text support is very good. Besides the normal font and emphases supports, it allows linking between multiple text areas and the ability to embed graphics and flow around them.

1.1 Exploring Squeak's Text

You should explore the Play With Me windows which are collapsed along the side of the image when you first start Squeak. *Play With Me 5* provides an example of Squeak's ability to flow text along curves and through multiple **TextMorph** instances (Figure 1). If you resize the **CurveMorph**

at the top larger (drag out on one of the yellow dots), you'll find that text flows from the bottom rectangle, through the "pipe," and into the curve. Make the curve smaller, and the text flows back down.

There are actually four **TextMorph**s in this picture: One in the large **CurveMorph**, one in the "pipe" **CurveMorph**, a third in the rectangle (an instance of **GradientMorph**), and the fourth embedded in the ellipse inside the rectangle. The first three are *linked* in that text flows from one to the other to the other, and back again. All four text areas are embedded in their respective shapes, and they are set to *fill owner's shape* (an option in the **TextMorph**'s red halo menu when the TextMorph is embedded in a shape).

The bottom rectangle's **TextMorph** is also set to *avoid occlusions* (another red halo menu). When a **TextMorph** is avoiding occlusions, then text flows around a shape laid on top of it. Morphic-select the ellipse and move it around with the brown halo (which moves without pulling it out of its embedded), and you'll see the text flow around the ellipse.

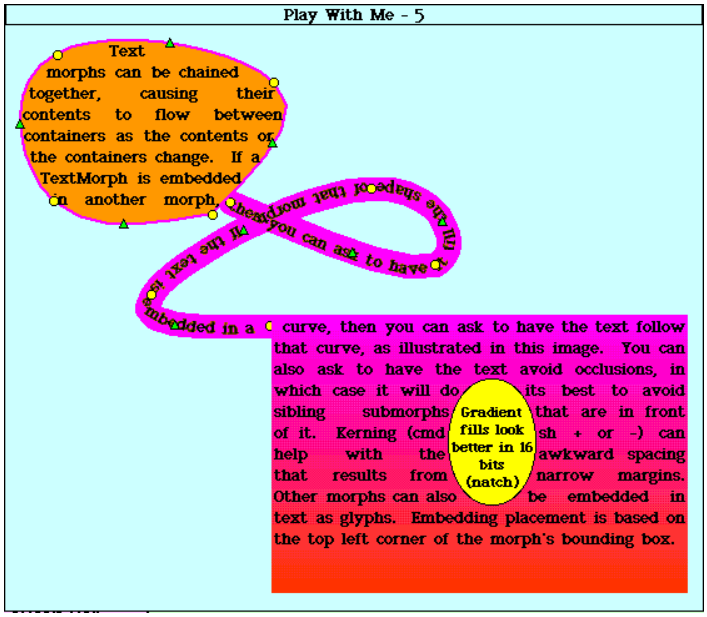

Figure 1: Text Flowing Through Curves

You can make your own linked **TextMorph**s. Create a **TextMorph** (simply drag out "Text for Editing" from the Supplies flap or the Standard Parts Bin), and choose *Add successor* from its red halo menu. A new, empty **TextMorph** will be attached to your cursor—just drop it somewhere. Now, start typing into your original **TextMorph**. When you stop typing, resize the **TextMorph**, and you'll see the text flow between the two.

You already saw in an earlier chapter that **TextMorph**s can change their style, font, and alignment. There are halo menus on **TextMorph**s

for changing each of these. You select the text that you want to modify, then choose the appropriate menu. Changes to emphasis, font, and style always take effect on the currently selected text. There are also command-keys for making text changes. Under the *Help* menu, look at *Command-key help*. On a Macintosh, command-1 will turn selected text to 10-point, command-3 will choose 18-point, and command-7 will boldface. (Repeating command-7 will toggle back to plaintext.) On Windows and UNIX, the command key is the Alt key, e.g., Alt-1 will 10-point selected text.

1.2 Programming Squeak's Text

As you would expect, all of the text manipulations described above are also accessible through Squeak code. The below example demonstrates several capabilities at once:

- Creating a TextMorph and embedding into an Ellipse,
- Making the TextMorph fill the Ellipse,
- Selecting text and changing its emphasis, and
- Creating a second TextMorph to fill into.

```
texta := TextMorph new openInWorld.
ellipse := EllipseMorph new openInWorld.
ellipse addMorph: texta.  "Embed the text in the Ellipse"
texta fillingOnOff. "Make the text fill the Ellipse."
texta contentsWrapped: 'My first textMorph in which I explore
different kinds of flowing.'.

"Demonstrating of changing emphases"
texta editor selectFrom: 1 to: 2.
texta editor setEmphasis: #bold.

"Make a second text area"
textb := TextMorph new openInWorld.

texta setSuccessor: textb. "A flows to B"
textb setPredecessor: texta. "B flows back to A"
```

texta recomposeChain. "Make the flow work."

As you resize the ellipse containing the first **TextMorph** (**texta** in the example), the linked successor (**textb**) automatically takes in the overflow (Figure 2). You can toggle whether or not occlusions are avoided with the message **occlusionsOnOff**. (**fillingOnOff** is also a toggle—the default is to have rectangular shape.)

Notice that the **TextMorph** itself doesn't know how to change text style. The **TextMorph** uses a **ParagraphEditor** to handle those kinds of manipulation. We access the **TextMorph**'s editor with the message **editor** in the above example. You can learn more about how to change alignment, emphasis, and style by looking at the **ParagraphEditor** methods for **changeAlignment**, **changeEmphasis**, and **changeStyle**.

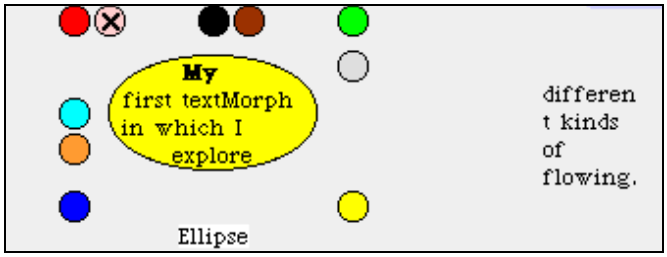

Figure 2: TextMorph Filling a Curve and Linking

Other kinds of fonts are possible in Squeak. The class **StrikeFont** knows how to read *BitFont* and *Strike2* formats in order to define new **TextStyle** font arrays—see **readFromStrike2**: and **readFromBitFont**: methods. The common TrueType font format is not supported for **TextMorph** instances (as of Squeak 2.7), but can be read and manipulated using the **TTFontReader** and its associated classes (such as the demonstration tool **TTSampleFontMorph**).

SideNote: Any Morph can generate Postscript for itself, including text areas. If you just ask a Morph to generate its Postscript (control-click menu or red-halo menu), it will generate EPS (Encapsulated Postscript) meant to be placed inside another document. Later, we'll introduce **BookMorph**s, which generate document Postscript.

Exercises with Text

1. Extend the text style menus to support single item selections for combinations of styles, like bold and italics, or bold and strikeout.

2. (Advanced) There are common font formats on the network that are not currently supported well by Squeak, such as TrueType and the MetaFont format. These are well-documented, and there is already a basic reader for

TrueType fonts. Create conversion routines so that these can be read and converted into **StrikeFont** instances.

2 Graphics and Animation

2.1 Simple Graphics Tools

Graphic images are easy to create or import into Squeak. From the file list, any file whose ending is *.bmp*, *.gif*, or *.jpeg* or *.jpg* can be directly opened as an ImageMorph (yellow button menu, *Open image in a window*). You can also use any of these images as a background image as well.

To create a graphic image, choose *Make New Drawing* from the *New Morph* menu from the World Menu. An onion skin layer (partially transparent) will appear with the painting tools (Figure 3). The onion skin allows you to create a modification for something already on the screen, like the onion skin that animators use for drawing the next cel in a cartoon.

The painting tools allow for basic painting with a number of different sizes of brush, as well as filling (paint bucket) and erasing. The gradated colors is actually a pop-up color picker: When you mouse over that area, you are offered a wide variety of colors to choose from. You can also use the dropper tool to select a color already on the display. You can draw basic objects (lines, rectangles, etc.). To use the stamps on the bottom, click on the corresponding box, then select something already on the display. You can now stamp the object you selected. When your drawing is complete, choose *Keep*, and your drawing will be a **SketchMorph** instance.

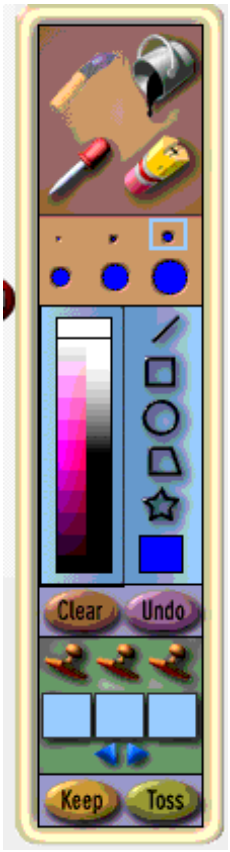

Figure 3: Paint Box for Creating New Graphics

As we've already seen, graphics are easily rotated and resized, as is any morph. The basic halos allow the rotation and resizing of any morph, **ImageMorph**s and **SketchMorph**s included. What is not immediately obvious is that any morph can also be easily animated to follow a simple path, even without using the Viewer framework.

Every morph (even windows) has a red-halo menu item to *Draw new path*. After selecting it, the cursor changes into a square with crosshairs, and you can now drag around the screen where you want the morph to go. When a path has been defined, your red-halo menu will change with the option to follow, delete, or draw a new path. Following the path has the morph move along the path.

2.2 Programming Simple Graphics

We have already seen in Chapter 3 that Squeak provides a powerful collection of image manipulation tools in the class **Form**. **Form**s can be scaled, rotated, chopped into pieces, and manipulated in any number of ways. **Form**s can easily be used in Morphic as well.

ImageMorphs are basically wrappers for **Form**s. Create an **ImageMorph**, set its image to a form, then tell it to **openInWorld**.

```
(ImageMorph new image: (Form fromUser)) openInWorld.
```

Forms can be read from external format files easily. **Form fromFileNamed: 'filename.gif'** will automatically convert GIF, JPEG, BMP, and PCX file formats into a **Form**. **Form**s can also be saved out via **writeBMPfileNamed**:, using the internal format **writeOn**:, and via the **ImageReadWriter** class hierarchy that knows about several external formats (including GIF, JPEG, PCX, and XBM).

To create your own **Form**s, there are **Pen**s for drawing, and classes like **Rectangle**, **Quadrangle**, **Arc**, **Circle**, and **Spline** that know how to draw themselves onto a **Form**. Typically, the display objects and display paths like these have **drawOn:** methods that take a form as an argument. The Color class provides instances that represent various colors, including **Color transparent**.

The simple animation described earlier is built into the **Morph** class. The method **definePath** follows the **Sensor** to fill an ordered collection of points as the **Morph**'s **pathPoints** property. The method **followPath** moves the morph along the points in **pathPoints**.

More sophisticated animations are possible by digging deeper into how graphics are presented in Squeak. Animated displays are really a matter of moving bits around on the screen. The class **BitBlt** (for bit block transfer, pronounced "bit-blit") knows how to do sophisticated translations of bits where the graphics to be laid on the screen are combined in interesting ways with the bits underneath. **BitBlt** has sixteen combination rules that explain how the source and destination bits are mixed. A really compelling example is the class message **alphaBlendDemo** which you can try with the below workspace code.

```
Display restoreAfter: [BitBlt alphaBlendDemo]
```

This demo displays several blocks of varying transparency, and then lets you "paint" (use the red button to lay down paint) which is semi-transparent with the underlying display. But more layers of "paint" is less transparent. This is an example of the sophisticated effects that **BitBlt** allows you to create.

BitBlt was invented by Dan Ingalls when he solved the problem of overlapping windows at Xerox PARC in 1974. The big problem of overlapping windows is saving parts of the underlying window for repainting later, updating as necessary when things move, and doing it all very quickly. **BitBlt** was made for this purpose, but is very general and enables other UI elements (pop-up menus) and animations.

For Squeak, Dan extended **BitBlt** for color, and then invented a successor to **BitBlt**, **WarpBlt**. **WarpBlt** takes a quadrilateral as its source pixels (not necessarily a rectangle anymore). The quadrilateral's first point

is the pixel that will end up in the top left of the destination rectangle, again, combining it with the same kinds of rules that **BitBlt** had. The result is very fast rotations, reflections, and scaling. For an interesting demonstration of **WarpBlt**, try the class message test1.

```
Display restoreAfter: [WarpBlt test1]
```

It's **WarpBlt** that allows Squeak to easily create thumbnails of projects (and **BookMorphs**, as we'll see). It's easy to use WarpBlt to create other kinds of effects. For example, a simple modification of Project's makeThumbnail creates a method for Morph for creating a thumbnail of any Morph.

```
makeThumbnail
    | viewSize thumbnail |
    "Make a thumbnail image of this image from the Display."

    viewSize ←self extent // 8.
    thumbnail ← Form extent: viewSize depth: Display depth.
    (WarpBlt toForm: thumbnail)
            sourceForm: Display;
            cellSize: 2;  "installs a colormap"
            combinationRule: Form over;
            copyQuad: (self bounds) innerCorners
            toRect: (0@0 extent: viewSize).
    (ImageMorph new image: thumbnail) openInWorld.
```

2.3 3-D Graphics for End-Users

When Andreas Raab joined the Squeak team, he built *Balloon*, a portable 3-D rendering engine. Balloon is accessed almost entirely through B3DRenderEngine. It's a very powerful 3-D system that supports different kinds of lights and textures, and pluggable rasterizer, shader, clipper, and transformer. The best quick example of Balloon is to create a new B3DMorph and play with it some.

Jeff Pierce had an internship at Disney Imagineering, and he developed Wonderland in Squeak. Jeff was part of a team (led by Randy Pausch at Carnegie Mellon University) that developed a Windows-specific 3-D scripting system called *Alice*. Wonderland is essentially Alice, but on top of Andreas' Balloon.

The window *Play With Me – 7* (collapsed when you start the image) is an open Wonderland (Figure 4). A Wonderland has (at least) two pieces to it: A **WonderlandCameraMorph** (at left) which shows you the world,

and a **WonderlandEditor** (at right) where the end user scripts the world. This Wonderland holds a **bunny**, which is added to the Wonderland by reading in a model. The Editor provides a workspace for scripting, a hierarchical view of all the objects, and buttons for manipulating the space—including an amazing ability to Undo, even actions executed as scripts.

Figure 4: An Open Wonderland in PWM-7

To create a new Wonderland, execute **Wonderland new**. In new Wonderlands, the editor window includes a *Quick Reference* tab (like the *Script* and *Actor Info* tabs seen in Figure 4) that provides the basic messages that objects understand in Wonderland. There are a standard set of commands that all Wonderland objects understand with respect to motion, rotations, responding to outside actions (like mouse clicks), changing colors, and other categories.

The scripting area in the editor is a modified workspace. Like a workspace, it can have various variables pre-defined. **w** is pre-defined to represent the current Wonderland. **camera** is the current camera, and **cameraWindow** is its window. All the objects in the hierarchical list are also defined in the workspace.

Wonderland can load in *.mdl* (Alice internal format) 3-D object files, VRML files (*.vrml*), and 3-D Design Studio (*.3ds*). The Alice project has made available a large collection of well-designed 3-D objects which is available on the CD and at ftp://st.cs.uiuc.edu/pub/Smalltalk/Squeak/alice/Objects.zip. Each of the methods to load one of these file types expects a path name to the appropriate kind of object—the below examples assume Macintosh pathnames (colon as path delimiter).

- To load a *.mdl* object (using one of provided examples, a snowman), **w makeActorFrom: 'myDisk:Squeak:Objects:Animals:Snowman.mdl'**
- To load a *.vrml* object, **w makeActorFromVRML: 'myDisk:Squeak:VRML:OffWeb.vrml'**.
- To load a *.3ds* object, **w makeActorFrom3DS: 'myDisk:Squeak:3DS:myBox.3ds'**.

Once you have some objects to play with, scripting them is tremendous fun. There are a few really interesting insights into scripting that permeate Wonderland and help to understand it:

- Your commands to Wonderland objects do not cause immediate jumps to the desired state. Rather, all changes *morph* between the current state and the desired state. **bunny head setColor: green** does not immediate turn from pink to green, but makes a visible transition. Morphing between states allows you to script as a series of desired states, and not worry about creating a good visual representation of the process.
- Method names are chosen so that simple actions are simple, and more selectors allow for greater specificity. **bunny turn: right** turns the bunny a bit to the right. **bunny turn: right turns: 2** does two quick rotations right. **bunny turn: right turns: 2 duration: 4** does two rotations over a space of four seconds.
- Wonderland works hard to make the language as obvious as possible. The scripting world is preloaded with symbols (like **right**), and **WonderlandActor**s get method defined for them on the fly so that a **bunny** knows its **head** and can be accessed as **bunny head**.
- All scripts actually return an **Animation** object of some kind. The concreteness of having all scripts correspond to objects allows for a powerful Undo operation—undo can literally undo any animation operation, and thus, any script. But even more, it allows for scripting sequences and patterns without ever writing a method. **fd←snowman move: forward** is valid, and fd points to an Animation that causes the snowman to go forward. **rt←snowman turn: right** also works, and **w doTogether: {fd . rt}** causes the snowman to walk and turn at the same time.

It's possible to construct objects even without using any external modeling program. Wonderland understands the core creation commands **makeActor** and **makeActorNamed**: The default Objects folder has lots of base shapes like spheres and squares. These can be moved around and then attached to an actor with **newObject becomeChildOf: newActor**. Objects can easily be colored.

Even more powerful, objects can easily have textures wrapped onto them. **snowman middle setTextureFromUser** will let the user select a portion of the display (a **Form**) then it will convert that to a texture and wrap it around the snowman's middle snowball. The method **setTexture:** will allow for a programmed input texture.

Most powerful is to create an active texture:

```
tex ← w makeActiveTexture.
snowman middle setTexturePointer: tex
```

When the active texture is created, a small blue rectangle appears on the Morphic desktop (Figure 5). Whatever is embedded in this blue rectangle appears in the Wonderland on the object—in real time. Embed a sketch, then resize it, and watch the snowman's middle change in the Wonderland as the rectangle is updated. Even more powerful is to embed something dynamic in the texture (e.g., the pluggable text area from the Scamper web browser)—one can have a working browser sitting in 3-D. When the message is sent **initializeMorphicReactions** to the textured Wonderland actor, then any clicks into the texture causes the 2-D Morphic space to update. With an active texture, the 3-D space can use anything from the 2-D space, and complex, dynamic textures are easily applied to Wonderland actors.

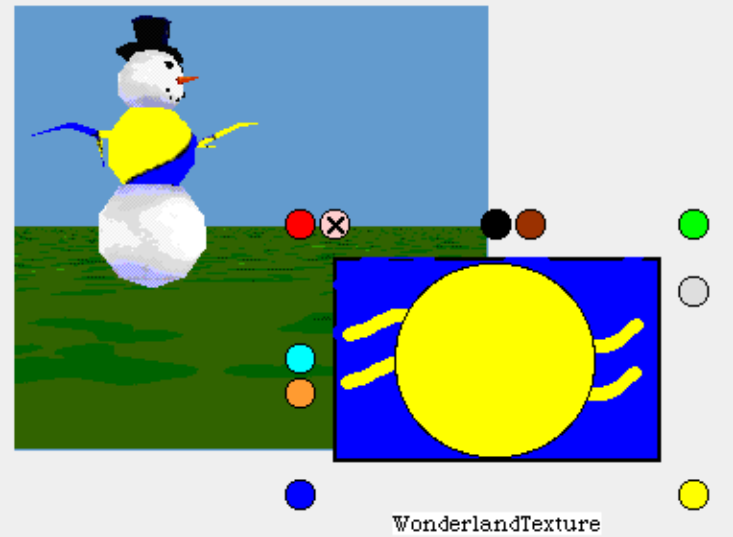

Figure 5: A Wonderland CameraWindow and WonderlandActor, with an ActiveTexture

2.3.1 Programming Wonderland without the Script Editor

It is possible to use Wonderland as a general world for 3-D graphics. It is less efficient this way—the underlying Balloon 3-D rendering engine is powerful and can be accessed directly, and Wonderland's morphing makes

exact control of animation a bit more difficult. But since Wonderland is so wonderfully scriptable, the transition from scripting to lower-level programming is eased by continuing to use Wonderland outside of the script editor.

To create a Wonderland that is under your control without an editor, execute:

```
w ← Wonderland new.
w getEditor hide
```

To bring the editor back:

```
w getEditor show
```

All the basic commands to the Wonderland will still work here, as they did in the script editor, e.g., **w makeActorFrom: 'myDisk:Squeak:Objects:Animals:Snowman.mdl'** as long as w is set to a Wonderland correctly. Obviously, executing a command like this will not put a variable named snowman in your method, as it does automatically for you in the scripting window. However, the step is very small—simply ask the Wonderland for its namespace. (w getNameSpace at: 'snowman') does actually return the WonderlandActor corresponding to the Snowman model, if you've already created it.

Better yet, all the other niceties of the scripting world are still available. Symbols like #right and #left are understood by WonderlandActor methods. Even more amazing, the methods that WonderlandActors understand for accessing the objects in their hierarchy work still. Thus, these messages do what you would expect:

```
(w getNamespace at: 'snowman') turn: #right
(w getNamespace at: 'shark') torso tail turn: #right
```

2.4 Flash Movies

Flash is a vector-based animation format that Macromedia supports with its Flash and Shockwave products (http://www.macromedia.com/). Flash animations are small, interactive, and can contain music as well as graphics. They are also well-supported by Squeak.

Flash movies (with file endings *.fla* and *.swf*) can be opened from the file list, using the yellow button menu. A FlashPlayerMorph will open with the Flash movie embedded. All the interactive components work (e.g., clicking on Next buttons), and sounds will play as expected. Figure 6 is an example taken from Macromedia's Flash tutorial.

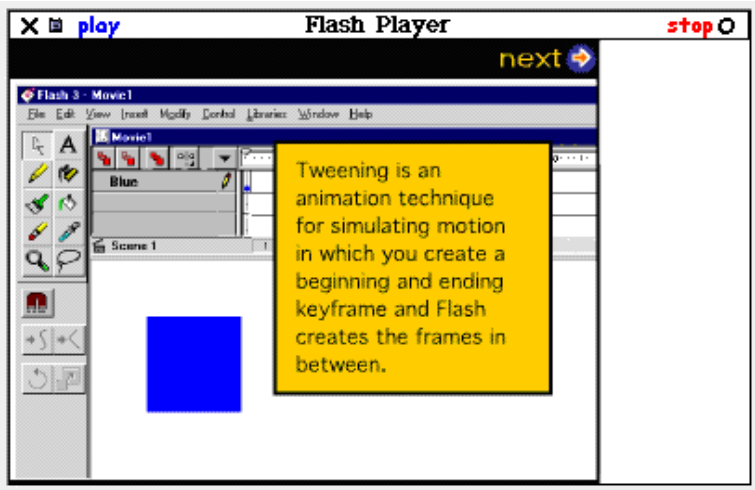

Figure 6: Flash Player on Movie from Macromedia's Flash Tutorial

The red halo menu on a Flash player provides some interesting capabilities. A control panel is available for stepping through and exploring a Flash movie. A thumbnail view shows an entire Flash movie at once, and by selecting a subset of the frames, a new Flash movie can be created.

The Flash support in Squeak is very powerful in its integration with the rest of Morphic. Individual Flash characters can be dragged out of the Flash movie and laid on the desktop, to be re-programmed or re-used as desired (Figure 7). You can Morphic-select objects in the Flash movie, then use the pickup (black) halo to drag them out onto the desktop.

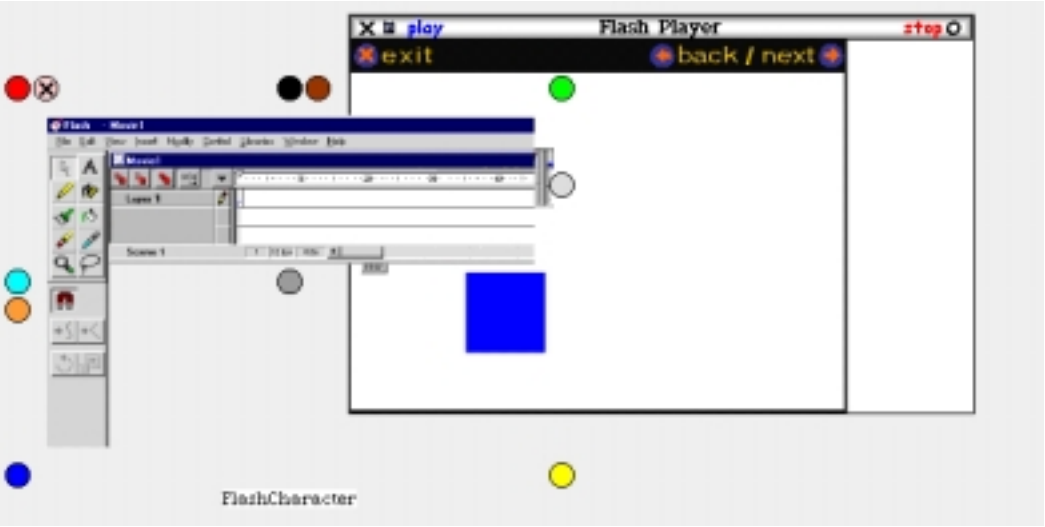

Figure 7: Dragging a Flash Character out of a Flash Movie

Exercises with Graphics and Animation

1. Create a drawing tool, rather than a painting tool, in Squeak. Let the user draw rectangles, lines, and ellipses, with a palette for choosing colors and line thicknesses and the ability to change layering. Provide an option for allowing the user to create the drawing as individual morphs or as a composite **SketchMorph**.

2. The painting tool currently lays down colored pixels in strips that completely overwrite the underlying background. Some painting tools today (like Dabbler and Painter) provide more complex painting, where laid paint can interact with the underlying paint (as do oils or water colors in the real world) and can even be laid in smaller elements, as if by the individual threads in a brush. Create modified brushes in the Squeak paint tool to gain some of these effects.

3. Create a movie or play (depending on your definition) using Wonderland.

4. (Advanced) Build a Flash composition tool, perhaps using the built-in path animation tools to create easily Flash movies.

5. Create a video game using Wonderland. Use the reaction methods so that clicking on objects does things, perhaps changing scenes or moving objects.

6. (Advanced) Used Wonderland to create a metaphor for other things in your system. For example, create a 3-D file manager by mapping files in a directory to objects in 3-D which can be opened if they're moved to a certain spot, or deleted if dropped into another spot.

3 Sound

Squeak's sound support is terrific. Excellent sound support helps to achieve the goal of a Dynabook. Squeak can handle sampled sound, can synthesize sound, and can handle higher-level sound formats like MIDI.

3.1 Recording, Viewing, and Editing Sound

The basic sound recording capability is the **RecordingControlsMorph** that allows the user to record and playback sound, up to available memory (Figure 8). (Only works on platforms whose VM's have sound recording implemented, which is Macintosh, Windows, and Linux as of this writing.) Once a sound is recorded, it can be trimmed (to remove the lack of sound at the beginning of a recording), tiled, or shown. A tiled sound can be titled, and is then available in the Viewer framework (Figure 9).

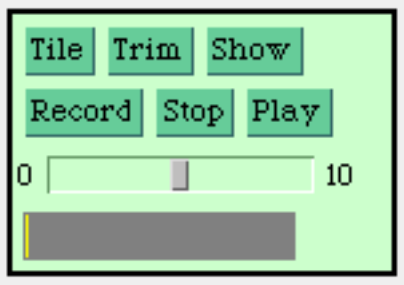

Figure 8: Recording Controls Morph

Figure 9: Tiled Sound

Showing a sound means to open a **WaveEditor** on it (Figure 10). A **WaveEditor** provides an impressive collection of tools for exploring and modifying the sound—setting a cursor, playing before or after the cursor, trimming before or after a cursor, even generating a Fast Fourier Transform (FFT) of the sound (Figure 11). (Roughly, an FFT is a graph of the sound, where the frequencies are on the horizontal axis and the amplitude of each frequency is the vertical axis.) There are additional editing functions available in the menu triggered by the <> button, not available in the obvious buttons.

The sound tile shown in Figure 9 references a named sound that is stored in an internal sound library. Other sound tiles also just point to the named sound, which saves on space. However, for some applications (like embedding sound in a **BookMorph**, discussed later in this chapter), you'd like a sound tile that embeds the sound in the tile itself. The original sound tile in Squeak did that, and it's possible to get it back by simply setting a boolean value to **false** in the **makeTile** method of **RecordingControlsMorph** (Figure 12).

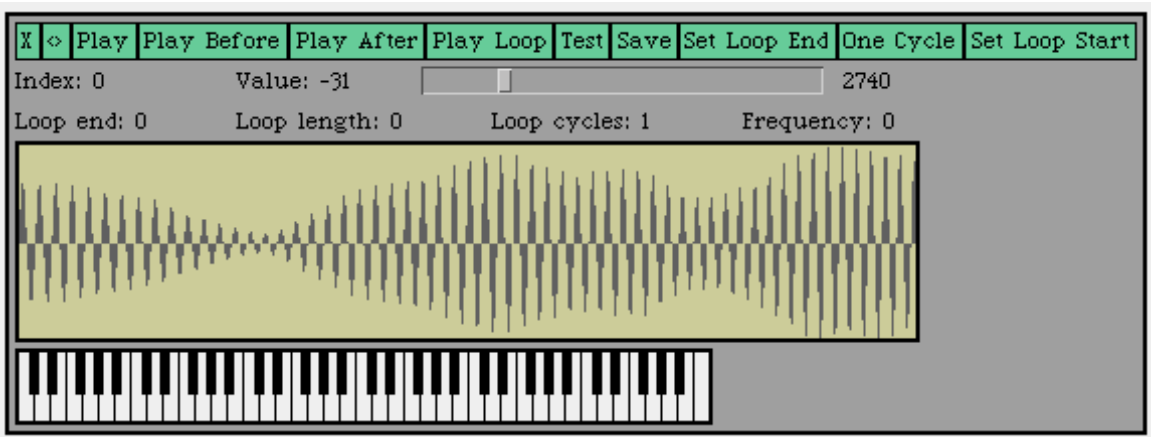

Figure 10: Showing a Wave Form

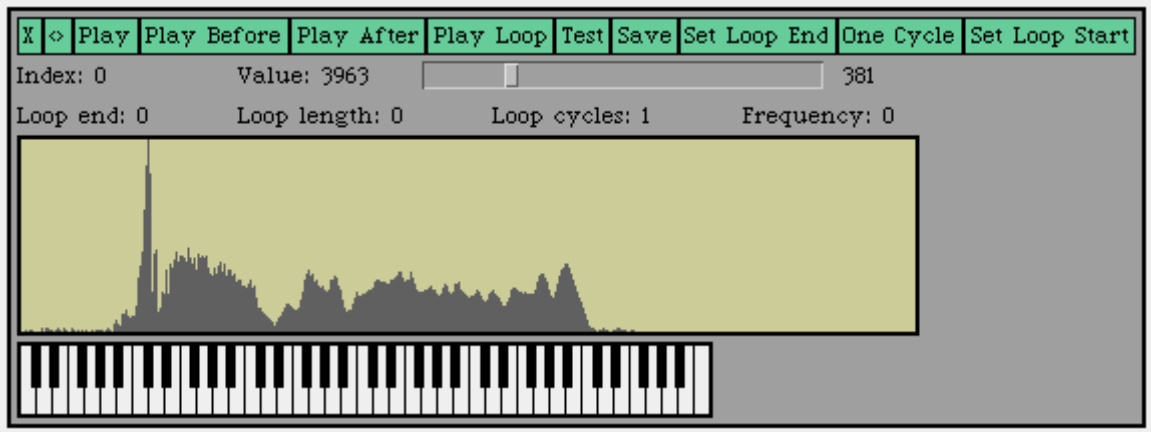

Figure 11: Fast Fourier Transform of the Recording

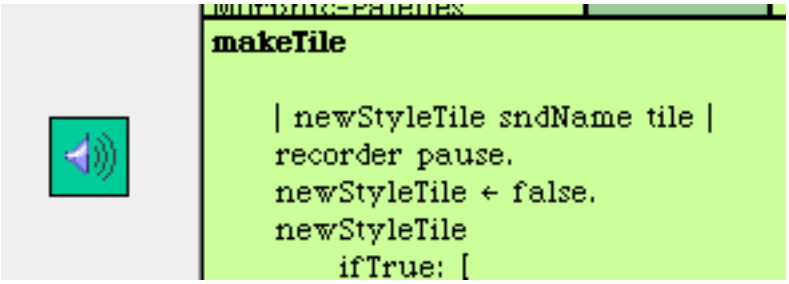

Figure 12: Old-style Tiled Sounds, and Where the Code is Changed

There is one additional morph that is very valuable for exploring sound, and that's the **SpectrumAnalyzerMorph** (Figure 13). The SpectrumAnalyzerMorph can display incoming sound in three different ways: As a sonogram, as a continuous waveform, or as a continually updating FFT. Choosing between types is available in the *Menu* button, and pressing the *Start* button starts recording. The **SpectrumAnalyzerMorph** is an amazing tool for exploring sound—and

makes for a wonderful demonstration when someone claims that Squeak is too slow to do real-time processing!

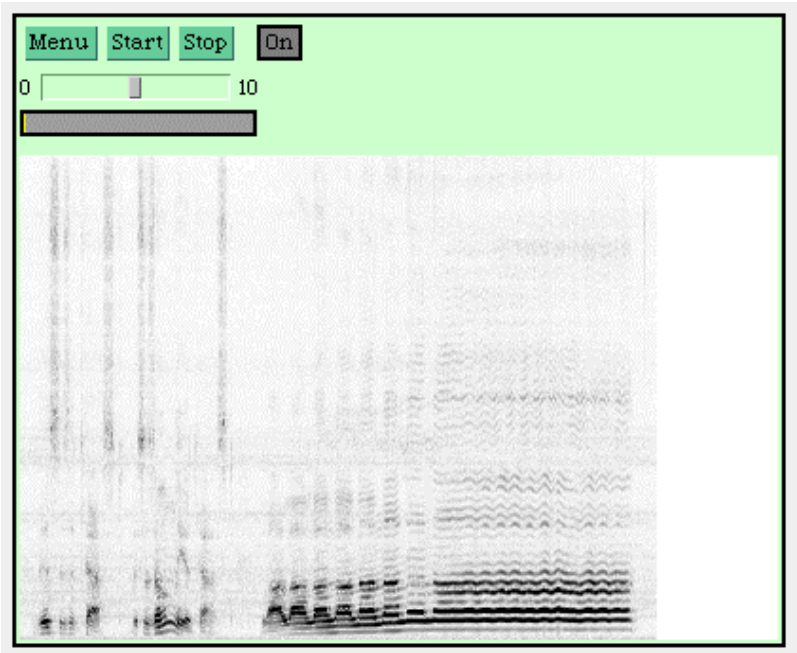

Figure 13: SpectrumAnalyzerMorph

3.2 Sound Classes

The heart of Squeak's sound support is the class **AbstractSound**. As the name implies, **AbstractSound** is an abstract class—it defines functionality for its subclasses, but it's not really useful to instantiate on its own. **AbstractSound** provides the default behavior of being able to **play** a sound, to **viewSamples** (to open it up in a **WaveEditor**), concatenating sounds, and others.

The model of sounds used in **AbstractSound** is that all sounds consist of a set of **samples** (which are sometimes stored in a **SoundBuffer** instance). If you were to graph these samples, you'd get the same kind of waves that you see in the Wave Editor—periodic, values alternating between positive and negative. The **samplingRate** is the number that tells you how many samples map to a second. The basic model of sound-as-samples works just as well for sounds that are *synthesized* (i.e., the samples are actually computed) as well as for those that are recorded or *sampled* (i.e., the samples are actually the input numbers from the interface to the microphone on your computer, via the **SoundRecorder**). Sound *envelopes* can be used as filters or functions that shape the samples as they're being generated to create different effects.

The method that generates the samples to be played is **mixSampleCount:into:startingAt:leftVol:rightVol:**. For objects that synthesize sounds (like **FMSound** and **PluckedSound**), this method actually figures out the sound of an instrument like an oboe or a plucked string instrument and computes the appropriate samples on the fly. For classes that handle recorded sounds (like **SampledSound**), this method provides the right number of samples from the given recording. As you might expect from the name of the method, Squeak's sound support automatically mixes sounds played together and can handle stereo sound with different volumes for the left and right speakers. The actual production of the sound (that is, sending the samples to the sound generation hardware) is handled by the class **SoundPlayer**.

AbstractSound is one of those classes that gives away many of its secrets by poking through its class methods. There are example methods there for playing scales and a Bach fugue. There are also examples of the simple class methods providing for easily generating music, the most significant of which is **noteSequenceOn:from:**. This method takes a sound and a collection of triplets in an array, and has that sound play those triplets. The triplets represent a pitch (name or frequency number), a duration, and a loudness, or **#rest** and a duration.

A collection of synthesized instruments is built into Squeak. They are available through the class message **AbstractSound soundNames**. (**FMSound**, as a subclass of **AbstractSound** that doesn't override **soundNames**, also has access to the same method.) By asking for the **soundNamed**:, you can get the sound object that synthesizes a given instrument, then use it as input to **noteSequenceOn:from:**.

```
(AbstractSound noteSequenceOn:
        (FMSound soundNamed: 'brass1') from:
                #((c4 1.0 500) (d4 1.0 500) (e4 1.0 500)
                (c5 1.0 500) (d5 1.0 500) (e5 1.0 500))) play
```

"Play c, d, e in the fourth octave, then c, d, e in the fifth. 1.0 duration. 500 volume."

You can add to the synthesized instruments in a couple of different ways. One way is to create an instance (e.g., of **FMSound**) that will generate samples that simulate an instrument. That's how the oboe and clarinet instruments are provided in Squeak. Another way is to provide a sample (recording) of an instrument. Most commercial synthesizers use samples to generate instruments. The advantage is high quality, but the disadvantage can be very large memory costs. (A 5Mb or more sample is not uncommon.) To create a sampled instrument, you need to make one or more recordings of different notes on the same instrument, then identify

loop points—points in the wave which can be repeated for as long as a given note needs to be sustained. The **WaveEditor** can be used for this process, as can many commercial and shareware sound recording packages.

A second sound library is built into the **SampleSound** class, also available through **soundNames** and **soundNamed**:. These are the sounds that are available as tiles in the Viewer framework. New-style sound tiles store their samples into this library, and the sound tile only stores the name of the sound in the library.

Sounds can be compressed, decompressed, and read and stored from standard compressed sound formats. AIFF files can be read and written (**fromAIFFfileNamed**: and **storeAIFFOnFileNamed**:), WAV files can be read (**fromWaveFileNamed**:) and easily written (see Chapter [MAT]), and U-Law files are easily handled (**uLawEncode**: and **uLawDecode**:). The more general compressing-decompressing (*codec*) classes are those inheriting from **SoundCodec**. **SoundCodec** is an abstract class defining an architecture for codec classes (see **compressSound**: and **decompressSound**:). Squeak comes with codec classes for ADPCM, GSM, MuLaw, and Wavelect codecs. These can be explored with the **CodecDemoMorph**, which accepts dropped (new-style) sound tiles, then compresses them, plays them back, and finally returns them to their original state. Since some of these are "lossy" algorithms (i.e., some sound quality is lost in favor of better compression), this gives you an opportunity to explore the sound quality tradeoffs of different algorithms.

3.3 MIDI Support

MIDI is the Musical Instrument Digital Interface. It is a hardware and software protocol which defines how to get different instruments to communicate with one another, so that different keyboards, MIDI guitars, drumpads, and other instruments can talk to one another and be easily controlled. MIDI files (typically ending in *.midi* or *.mid*) are found in many repositories on the Internet.

Squeak can play and manipulate MIDI files. The easiest way to open up the MIDI **ScorePlayerMorph** is by selecting a MIDI file in the FileList then playing it as MIDI from the yellow-button menu. The ScorePlayerMorph (left side of Figure 14) lets you see all the tracks in the MIDI piece and the textual description of what instrument is in the track. Each track can be muted, panned from left to right, and have its volume changed.

A Piano Roll representation of the MIDI score can be generated from the **ScorePlayerMorph**. The Piano Roll representation shows each track as a separate color, where each note is a line segment (right of Figure 14).

The vertical position of the line segment indicates the pitch of the note, and the length indicates the duration. The vertical red line is a cursor indicating the current position in the song.

The Piano Roll notation allows for manipulation of the song and the representation. Red button clicking in the piano roll opens up a menu that allows you to change the view (e.g., contract or expand time) or to open a keyboard (lower right of Figure 14) for inserting new notes into the score. By yellow button clicking and dragging over the notes, notes can be selected, copied, and pasted.

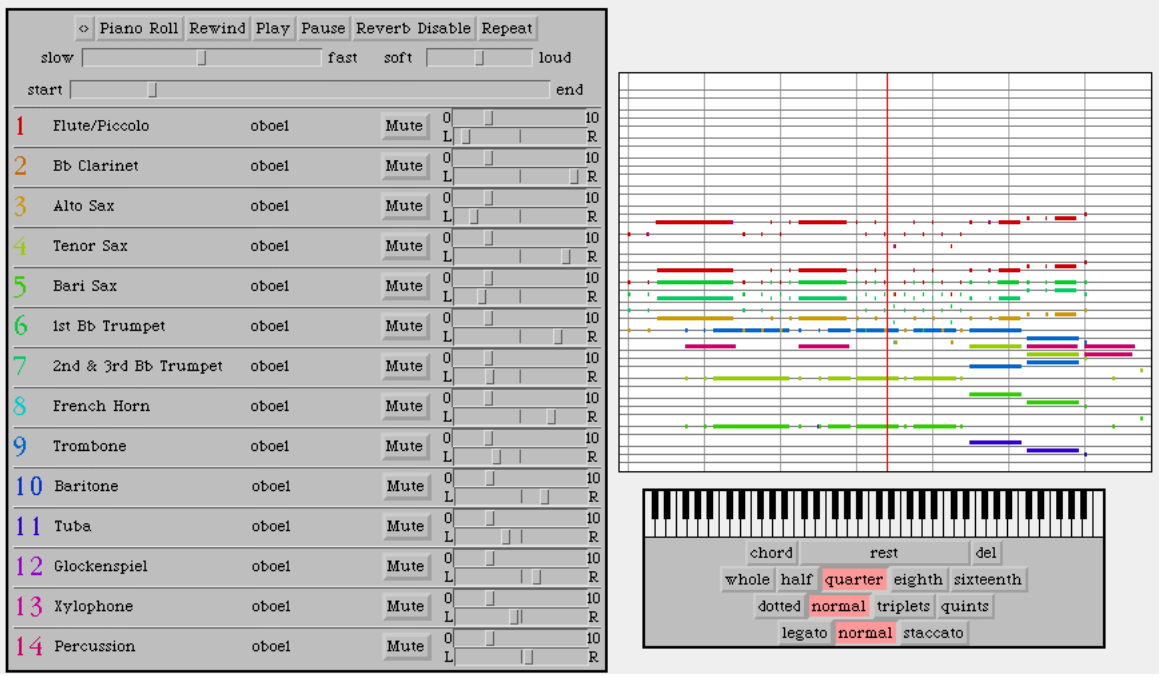

Figure 14: MIDI Tools in Squeak

> ***HistoricalNote:*** The Piano Roll notation in Figure 14 dates back to the *Twang* music editing system built by Ted Kaehler for Smalltalk-72.

MIDI scores can be played several different ways in Squeak. The default is to use the internal music synthesis classes. When using the internal classes, each individual track's instrument can be changed. In Figure 14, they're all set to *oboe1*. Clicking on the instrument name opens a menu for changing the instrument, or even editing the instrument.

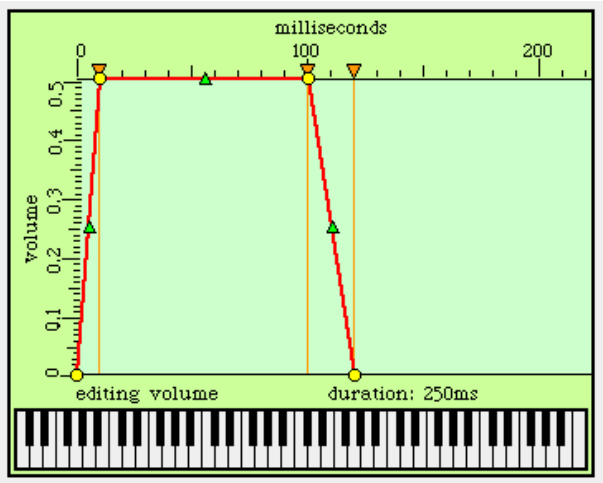

Figure 15: EnvelopeEditorMorph on Oboe1

When you edit the instrument, you can change its *envelopes* (Figure 15) or even add new envelopes. Envelopes can modify the volume, pitch, or other parameters of the sound as it's being generated. For example, in Figure 15, the oboe1 sound increases its volume sharply as it's first being generated (note the first vertical bar, which can be adjusted) called the *attack*, then held for the length of the note (*sustain*), then dropped as the note *decays*. By clicking and dragging the **EnvelopeLineMorph** (a subclass of **PolygonMorph**), you can make changes like have the volume drop during the sustain for a warble effect. The editing label is actually a pop-up menu for choosing or adding a different envelope. The keyboard beneath the editor lets you test the sound as you edit it. The control-click menu on the **EnvelopeEditorMorph** allows you save your new instrument.

The MIDI player also allows you to play the score through an external MIDI device or a platform-specific software MIDI synthesizer. (Access to the platform-specific software MIDI synthesizer is currently only available for Macintosh and Windows systems, though all the code is available to port it to other platforms.) The <> button on the **ScorePlayerMorph** pops up a menu for choosing what synthesizer you wish to use. If any software synthesizers are available (e.g., via Apple QuickTime), they'll appear as an option, and the MIDI player will output through the selected one. If you choose an external MIDI synthesizer, the controls on panning, volume, and instrument selection are ineffective.

CautionaryNote: If you tell the **ScorePlayerMorph** to output to an external MIDI synthesizer, *but you don't actually have one*, you can cause your system to hang. The timing access for an external MIDI interface is written at a low-level which can't be interrupted from Squeak. If the external interface isn't available, your system will hang waiting for it.

It is possible to input MIDI from a keyboard or other device. The **MidiInputMorph** will let you input MIDI and map it to a synthesized voice. There isn't built-in support to do anything else with MIDI input other than simply playing it, but the classes are there to create more sophisticated tools that blend MIDI input with other sound tools.

3.3.1 MIDI Support Classes

The classes to support MIDI in Squeak are rich and well-designed. A **MIDIFileReader** reads a MIDI file (which must be in binary mode, as opposed to the default character mode). The **MIDIFileReader** can generate a **MIDIScore** with the **asScore** conversion message. A **ScorePlayer** can play a score on the internal MIDI synthesizer, by default. The below workspace code will play a given MIDI file (here, with a Macintosh path and filename).

```
f ← FileStream fileNamed:
        'MyHardDisk:midi:candle.mid'.      "Open the file"
f binary.                          "and make it binary."
        "Read it as MIDI and convert it to MIDIScore"
score ← (MIDIFileReader new readMIDIFrom: f) asScore.
f close. "Close the file"
        "Open a ScorePlayer"
scorePlayer ← ScorePlayer onScore: score.
scorePlayer reset.                 "Reset it to start playing."
scorePlayer resumePlaying.         "And start it playing."
```

Access to external MIDI devices involves some extra classes. The **SimpleMIDIPort** class is used as the interface to the external MIDI output devices. The class **MIDISynth** is used to handle input from external MIDI devices. The below variation of the workspace code asks the user for an external MIDI port, then plays the MIDI score through that.

```
f  ← FileStream fileNamed:
        'MyHardDisk:midi:candle.mid'. "Open the file"
f binary.                          "and make it binary."
        "Read it as MIDI and convert it to MIDIScore"
score ← (MIDIFileReader new readMIDIFrom: f) asScore.
f close. "Close the file"
         "Open a ScorePlayer"
```

```
scorePlayer ← ScorePlayer onScore: score.
    "Ask the user where to send the output: External vs. Platform-specific internal"
portNum ← SimpleMIDIPort outputPortNumFromUser.
    "Tell the scorePlayer to use this MIDI port."
scorePlayer openMIDIPort: portNum.
    "Reset it to initialize, then start playing."
scorePlayer reset.
scorePlayer resumePlaying.
```

CautionaryNote: While playing around, it's possible to get your external MIDI ports into an odd state, e.g., where some are opened but you've lost an object reference to close them. **SimpleMIDIPort closeAllPorts** is usually effective for putting everything back into a usable state.

Exercises on Sound

7. Create a new kind of instrument by editing the envelopes of the oboe sound. Can you express what each of the different envelopes does to the sound?

8. Find a piece of music and transcribe it into Squeak. (Best if you have to play multiple voices at once.) Try different orchestrations (different instruments) for the piece.

9. Use the Sonogram and see if you can figure out the wave pattern difference between each of the vowel sounds. Is it the same for you and for someone else?

10. (Advanced) Use the Wave Editor and sample an instrument. (There are directions on the Squeak Swiki.)

4 Networking

Squeak has been enhanced since the original Smalltalk-80 with good support for networking. Since the advent of the Web, a lot of multimedia work has focused on making things available on the Internet and using material that is out there. Thus, good networking support becomes an important part of a multimedia toolbox. FTP, HTTP, mail protocols (SMTP and POP), and IRC are all supported. User interface tools exist for all of these in Squeak.

4.1 User Interfaces for Networking

Scamper is Squeak's web-browser (Figure 16). You open it from the *Open…* menu and choosing *Web browser*. It can handle basic HTML, forms, and images (in GIF, JPEG, and BMP formats). It has no support for Java or JavaScript. As of Squeak 2.7, there is no support for tables nor frames.

Figure 16: Scamper Web Browser

The basic FileList is also an FTP client (Figure 17). If you click all the way up to the top level of the directory tree, you see not only your mounted disks, but known FTP servers (preceded by "↑" in the directory contents pane). You can add servers by choosing *Add server…* from the yellow button menu in the directory pane (the one containing **[]**).

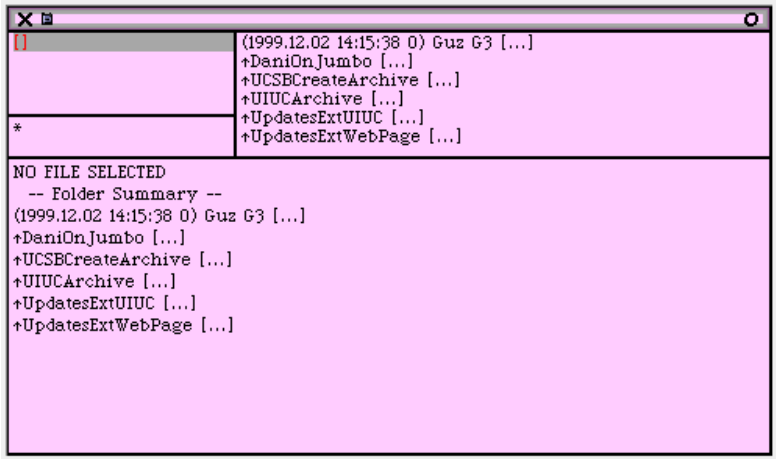

Figure 17: FTP Servers in FileList

Celeste is Squeak's email client (Figure 18). Celeste speaks POP and SMTP mail protocols. It can filter messages automatically into mailboxes based on subject, sender, or custom queries. It can import or export mailboxes from Eudora or UNIX formats.

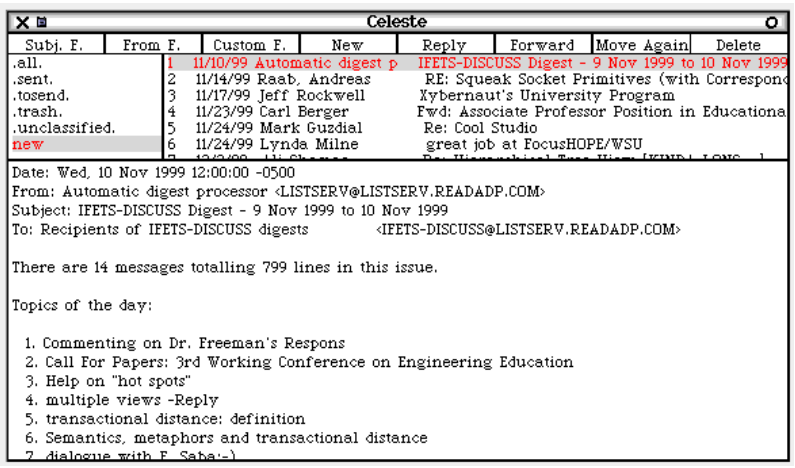

Figure 18: Celeste the Email Client

4.2 Programming Network Access

Squeak's networking support provides low-level and high-level access. At the low-level is the basic **Socket** class. **Socket**s are a standard mechanism for creating Internet connections across many platforms. Squeak's sockets can support TCP/IP and UDP access.

> ***CautionaryNote:*** While working with networking, some tools may assume that a network connection already exists. If one hasn't been set up, use of the tool may generate an error. To initialize the network, simply do **Socket initializeNetwork**.

Subclasses of **Socket** provide more conceptually high-level access for different protocols. **SimpleClientSocket** provides the basic mechanism for accessing a variety of protocols. **POPSocket** and **SMTPSocket** provide class methods that demonstrate how to read and write email.

FTPSocket provides access to FTP servers, though it's typically used through **ServerDirectory** that provides an abstraction for dealing with internal files, FTP, and HTTP access in a similar way. Workspace code that stores and retrieves from an FTP server follows:

```
ftp ← ServerDirectory new.
ftp server: 'cleon.cc.gatech.edu'.    "host"
ftp user: 'guzdial'.
ftp password: 'fredflintstone'.
ftp directory: '/net/faculty/guzdial'.
ftp openFTP.
```

```
ftp putFile: (FileStream fileNamed: 'myfile') named: 'remotefile'.
ftp getFileNamed: 'remotefile' into: (FileStream fileNamed: 'myfile-downloaded') open.
ftp quit.
```

HTTPSocket provides several class methods for directly accessing material on webservers. The most general access method is with the class method **httpGet**: that takes a URL as a string. What it returns is a **RWBinaryOrTextStream**, a very general stream that can be interpreted as text or binary (e.g., for images) as you choose. See the class methods **httpShowPage**: to see how to grab the text out of the returned stream, and **httpGif**: and **httpJpeg**: show how to grab a GIF or JPEG image out of the returned stream.

Scamper has many supporting classes that enable it to interpret the network intelligently. **HTMLParser** has a class method **parse**: that accepts an HTML document and returns an **HTMLDocument**. **HTMLDocument**, in turn, can answer conceptual entities of itself, such as **head** and **body**. The class **Url** and its subclasses (e.g., **BrowserUrl**) know how to parse themselves and retrieve their contents. The class **MIMEDocument** knows about MIME types, and can store a document and its type. These classes are designed to be reused in new network applications.

4.3 Web Serving with PWS

Squeak also comes complete with a webserver, the Pluggable WebServer (PWS). The class **PWS** serves two different roles between its class and instance method sides (which isn't a particularly good object-oriented design). On the class methods side, it implements a webserver. On the instance method side, it represents a specific request to the webserver. The **PWS** was originally written by the author, based heavily on a Squeak-based webserver by Georg Gollman.

PWS can work simply as a basic webserver. You must modify the class method **serverDirectory** in the class **ServerAction** so that it returns the path to the directory where you will serve files. (Be sure that the last character of the path is your platform's path separator character.) You can then execute the below:

```
"Make the default server action be serving a file."
PWS link: 'default' to: ServerAction new.
"Start serving"
PWS serveOnPort: 8080 loggingTo: 'log.txt'.
```

To stop the server, execute **PWS stopServer**.

With the server running, any HTML, GIF, or other files stored in your server directory are available on the Web (assuming that your computer is on the Internet). That is, if you create an HTML file named *myfile.html* and place it in the directory that you specified in serverDirectory, any browser in the world can access that file via http://your-machines-address:8080 /myfile.html. For the author, this might be http://guzdial.cc.gatech.edu:8080/myfile.html (You can get your machine's address by PrintIt on **NetNameResolver nameForAddress: (NetNameResolver localHostAddress) timeout: 30**). The *8080* indicates the *port* number of the Web server. Most webservers serve from port 80, but many systems (notably, Windows NT and UNIX systems) require the user to be an administrator to create a webserver on port 80. Any user can use ports above 1024 on just about any system.

If **PWS** could only serve files it would be only mildly interesting, but it's more powerful than that. You can also generate information dynamically and serve it via **PWS**, the way that you can with *CGI scripts* on other webservers. There is a collection of examples of PWS interactive web pages, including two different collaboration tools, at http://guzdial.cc.gatech.edu/st/server.tar

To use these tools, unpack the archive, and move all the files into your server directory (e.g., the folder *swiki* should be within your server directory). You can use **PWS initializeAll** to install all the examples.

Several of the examples have to do with supporting different forms of collaboration on the Web. One of the examples is a simple chat page. Go to the address http://your-machines-address:8080/chat.html to conduct a chat in HTML. The page refreshes itself every so often, so other visitors to the same page will see your comments and their own. Another one is a more structured comment space: Try http://your-machines-address:8080/comment.Squeak

The most powerful of these examples is the Swiki. Ward Cunningham (who invented CRC cards with Kent Beck) invented a kind of website called the *WikiWikiWeb*. WikiWiki is Hawaiian creole for "quick." The quickest way in the world to create a website is to invite everyone on the Internet to edit and create pages on your website. The original WikiWikWeb is available at http://c2.com/cgi/wiki?WelcomeVisitors Swiki is a Squeak interpretation of Ward's idea, thus, a Squeak Wiki or *Swiki*. The Squeak Swiki is at http://minnow.cc.gatech.edu/squeak.1

To create a Swiki (assuming that the examples files are downloaded and installed), you start by creating a folder in your server directory. You should make the folder's name a single word (no spaces) to make it easier to access, like *myswiki*. Now, set up the Swiki with **SwikiAction setUp:**

'myswiki' (replacing *myswiki* with the name of your folder). Just before you start up your server, you should execute **SwikiAction new restore: 'myswiki'**. Your Swiki will be available at http://your-machines-address:8080/myswiki.1

4.3.1 Programming the PWS

The purpose for the **PWS** was to provide a simple mechanism for building new kinds of interactive Web applications. **PWS** makes it simple by providing some pieces for making Web applications simple and by supporting embedded Squeak. This section introduces both of these.

You write a **PWS** application by defining an object that responds to the message **process**: with an argument of a **PWS** instance (that is, a request to your webserver). Let's say that you define a class named **SimpleExampleAction** (with no instance variables, and it can just be a subclass of **Object**) and gave it an instance method like this:

```
process: aRequest
        "Return a code indicating success"
        aRequest reply: PWS success.
        "Now, tell the browser that we're giving it HTML."
        aRequest reply: PWS contentHTML, PWS crlf.

        "Send it the HTML. Note the commas concatenating generated stuff from literal text"
        aRequest reply: '<html>
<head><title>A Simple Example</title></head>
<body>The time is: ',(Time now printString),'
<p>The date is: ',(Date today printString),'
<p>The URL you used to get here was: ',(aRequest message printString),'
</body></html>'
```

What this method does is:

- Tells the connecting browser that the connection was successful. The request object knows how to reply to the browser, and **PWS success** is the appropriate code. (As opposed to **PWS notFound** that tells the browser that it has a bad URL.)
- Tells the browser that what's being returned is HTML (**PWS contentHTML**) and returns a final carriage-return-line-feed (**PWS crlf**) to end the header.
- Returns the HTML to the browser. The HTML above happens to generate some elements on-the-fly, like the time, the date, and the

URL used to reach this method (**message**). The PWS request instance also responds to the message **fields** which returns a dictionary of all the HTML form elements that may have been posted to the URL, already all parsed out into easily accessible elements. This makes creation of interactive Web applications easy.

To install this application as a URL, we have to link it to PWS with a keyword. The keyword is the name of the application to be included in the browser. **PWS link: 'example' to: (SimpleExampleAction new)** would work, so that http://your-machines-address:8080/example would trigger the above method (Figure 19). All the PWS linkages are stored in its **actions** dictionary.

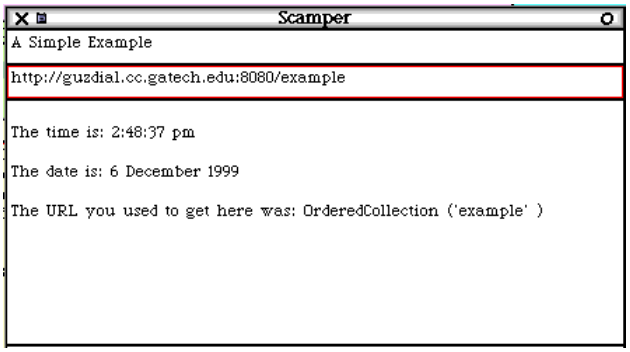

Figure 19: Scamper view of SimpleExampleAction

The class that implements basic file serving in PWS is **ServerAction**. It is designed to be easily overriden by subclasses to implement specific serving features. One of these subclasses is **EmbeddedServerAction**. Before returning files referenced through instances of this class, the action first asks **HTMLformatter** to evaluate embedded Squeak in the file. Anything enclosed in the made-up tags <? and ?> is treated as Squeak code, evaluated, and the returned result is included in the returned file.

The **SimpleExampleAction** can be done with embedded Squeak like the below. This example is included in the examples and is available as http://your-machines-address:8080/embedded/Sample.html

```
<html><title>Sample Embedded Page</title>
<body> <h2>Welcome</h2>
<p>Today is <?Date today printString?>
<p>Now is <?Time now printString?>
</body> </html>
```

Using embedded Squeak and accessing forms input via the **fields** message, it's easy to create a one file interactive application. The variable **request** can be accessed from within embedded Squeak to get the actual

PWS instance. The below file (http://your-machines-address:8080/embedded/Factorial.html) prompts you for a number, and then returns the factorial of that number. Note that **fields** returns **nil** if no form data is available (like when first visiting this page), and returns a string for everything else (which is why **asNumber** is needed before computing the factorial). Specifying the same file as the *action* for the *POST* means that this same file will be served again when the *submit* button is pressed.

```
 <html><title>Factorial Calculator</title>
<body>
<form method="POST" action="factorial.html">
<p><b>Number to compute:</b>
<input type=text name="number"
        value="<?request fields notNil
                ifTrue: [request fields at: 'number' ifAbsent: ['0']]
                 ifFalse: ['0']?>"
        size=10 maxlength=10>
<p><input type=submit name="action" value="Compute Factorial">
<hr>
<p><b>Factorial</b>
<p> <?request fields notNil
        ifTrue: [(request fields at: 'number' ifAbsent: ['0'])
                asNumber factorial]
        ifFalse: ['nothing yet']?>
</form> </body> </html>
```

Exercises with Networking

11. (Advanced) Modify Scamper so that each visit to a page generates a thumbnail of the page *so that* clicking on the thumbnail opens the same Web page. This creates a new kind of web history.

12. (Advanced) Use the networking primitives to create MIDI-at-a-distance. Input MIDI from a keyboard, send it across the network to a client, and play the sound on the client's computer.

13. Use the Web access methods in HTTPSocket and the TextMorph's seen earlier to create a personal newspaper. Generate a well-formatted

newspaper of your user's favorite websites, with multiple columns and images.

5 New Media in Squeak

Squeak is also being used to invent new media forms unique to Squeak. These media are being invented as steppingstones toward the Dynabook vision of personal dynamic media. Two of these media, SqueakMovies and BookMorphs, are presented in this section. Both are fairly new and are still in development, but they point toward exciting new areas for inventing media in Squeak.

5.1 SqueakMovies

SqueakMovies are not as sophisticated as MPEG or QuickTime movies (though they may be in their final form). But they do allow you to easily capture activity in Squeak. Used in combination with things like Wonderland, they form a fascinating way of exploring animated movies—using Wonderland to script, and SqueakMovies to capture and deliver the final presentation.

To create a movie from Wonderland, first create one with **Wonderland new** then load up an actor.

```
w makeActorFrom:
'MyHardDisk:Squeak:Objects:Animals:Snowman.mdl'
```

The below code (to be executed in a Wonderland script editor) creates a movie named *snowman.movie* where the snowman wanders around in a circle. There was an important problem that had to be solved to make this code work. Wonderland creates intermediate steps in an animation, and schedules them over some period of time to create a pleasing effect. But to capture the Wonderland script to movie, it was important to get the animation to occur immediately, in order to capture the movie. The tricks were (a) to tell Wonderland that the duration was **rightNow** and (b) to get the Morphic **World** to update the Wonderland by forcing it to **doOneCycle**.

```
| frame1  |
out ← FileStream newFileNamed: 'snowman.movie'.
out binary.
“Create a header”
frame1 ← (Form fromDisplay: (cameraWindow bounds)).
out nextInt32Put: 22. "Treat as Magic number for now"
out nextInt32Put: (frame1 extent x).
```

```
        out nextInt32Put: (frame1 extent y).
        out nextInt32Put: (frame1 depth).
        out nextInt32Put: 44. "frames"
        out nextInt32Put: 100000. "Time in microseconds between
frames"
        (7 to: 32)                    "Padding"
                do: [:i | out nextInt32Put: i].


        "Now, do the walking and turning, and create frames.
        Each frame is a Form snapped from the Display within the
        cameraWindow's bounds. Forms understand writeToMovie:"
        1 to: 22 do: [:count |
                snowman move: forward distance: 0.3
                        duration: rightNow.
                World doOneCycle.
                (Form fromDisplay: (cameraWindow bounds))
                        writeOnMovie: out.
                snowman turn: right turns: 0.3
                        duration: rightNow.
                World doOneCycle.
                (Form fromDisplay: (cameraWindow bounds))
                        writeOnMovie: out.
        ].
        out close. "End the movie"
```

To view the movie, find it in a file list and use the yellow button menu to open it as a movie (Figure 20). (Click on the rightside hollow circle to get all the controller buttons seen in Figure 20.) The movie player can be used to add a soundtrack to a movie. Choose *Add soundtrack* from the menu button (<<>>). A list of WAV and AIFF files in your current directory will pop-up. Pick one, and when you play the movie, the sound will be played at the same time.

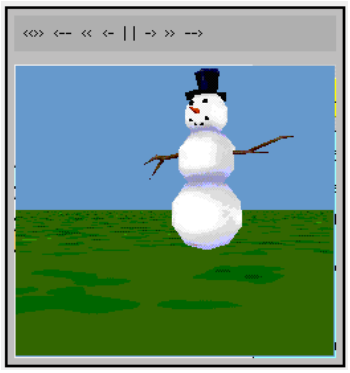

Figure 20: Squeak Movie Player

You can synchronize a movie with MIDI in an even more interesting manner. The thumbnails that you can generate from a movie frame are actually synchronization tools. If you open a Piano Roll for drag-and-drop (from its red halo menu), you can drop the thumbnails at key points in the music. When the MIDI score is played, the movie will make sure that the right frame is showing at the right point in the score (Figure 21).

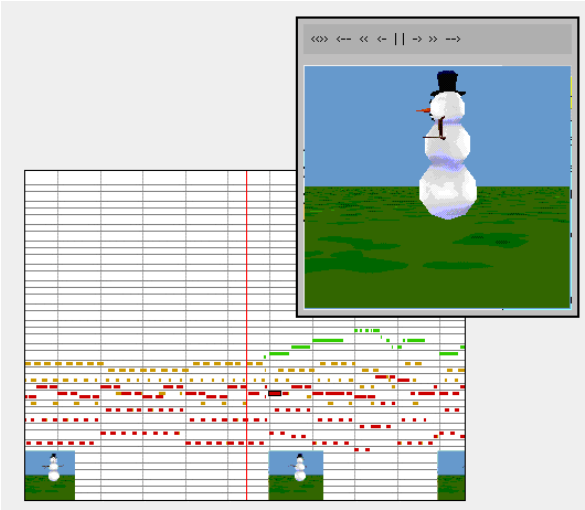

Figure 21: Synchronizing MIDI with a Movie (note thumbnails at bottom of Piano Roll)

5.2 BookMorphs

Apple's HyperCard was a really powerful tool, in that it allowed anyone to easily create small applications. These applications could contain text, buttons, and link to other kinds of media. In that way, HyperCard took a step closer to a Dynabook.

The basic metaphor of HyperCard was a *stack* of *pages*. Each page could contain different elements, and a basic structure for a set of pages could be defined in a shared *background*. Users could use HyperCard

stacks as notebooks or libraries (storing things on different pages with a common structure in the background), as slides shows (with transitions and sounds between pages), or even as more complex applications by *scripting* (programming in a simple language) the stack.

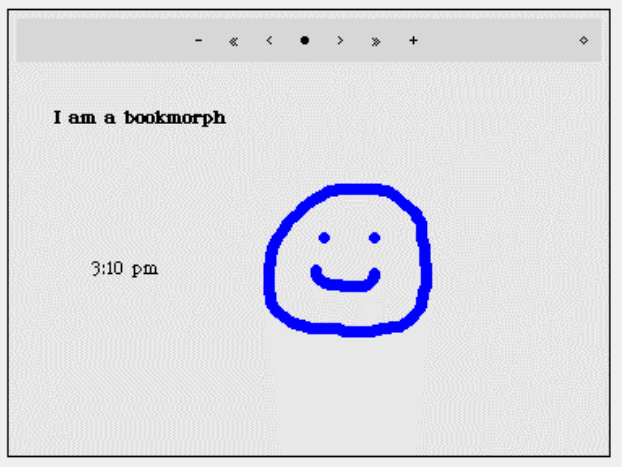

Figure 22: A Simple BookMorph

The HyperCard stack maps to Squeak through the **BookMorph** (Figure 22). Like HyperCard stacks, BookMorphs consist of pages—**PasteUpMorph**s. Like HyperCard, basic structures can be stored on the book prototype (any page can be made the prototype by clicking on the sold circle menu button) and are then available on every new page. Like HyperCard, transitions and sounds can be defined between pages. (BookMorphs also offer a PowerPoint-like page sorter, which is available from the same solid circle menu button.)

Unlike HyperCard, a **BookMorph** can hold *any* morph, so it can be dynamic and well-structured. Embedded in a **BookMorph**, **ClockMorph**s update, **BouncingAtomMorph**s bounce, and live graphs can change in response to updates in Web-based data or queries from users. Simulations, construction kits, and even whole programming environments can be embedded inside a **BookMorph**.

BookMorphs are designed to be easily saved and loaded from disk or network. The menu item *Send all pages to server* from the circle button menu will let you save your book. Choose *Use page numbers* when prompted, and then provide a URL for the book. **BookMorph**s know how to deal with ftp:// and file:// URLs. Your book will be saved with a filename ending in *.bo* for the book index and *.sp1*, *.sp2*, and so on for the pages.

You can open a book from a FileList by using the yellow button menu with the book index file is selected (from disk or FTP). Or, from a workspace, you can execute **BookMorph grabURL: 'file://My Hard Disk/mybook.bo'** with any URL. The **BookMorph** is smart about

managing memory and pages. Each page is not loaded until it is needed, and older pages are purged as they are no longer needed.

A **BookMorph** becomes a powerful mechanism for creating multimedia documents in Squeak and sharing them with others. Ted Kaehler, who worked on HyperCard and built most of the **BookMorph**s, talks about using them to write *active essays*—documents that have text and graphics, but where some illustrations are active, dynamic, explorations of the concepts in the book. In this way, the **BookMorph** becomes an exploration space for Dynabook ideas.

6 Making the Dynabook in Squeak

Despite the rapid pace of this chapter and the many forms of media supported in Squeak, there is much more in Squeak than is discussed here that moves it toward a Dynabook—and even more that needs to be built and explored yet. Some of the additional features of Squeak 2.7 that we're not discussing but are relevant include:

- Support for high-resolution tablets, as in the original Flex. (See class methods in **Pen** for these.)
- Support for writing recognition (see class **CharRecog**).
- Support for language understanding (see class **WordNet**).
- Support for exceptions in order to create more robust systems (see class **Exception**).
- Support for extending the VM and creating low-level primitives through Squeak. (See the class category *Squeak-Plugins* for information on this.)

Even then, there's much more to be done before a Dynabook is realized and many questions to be answered. While the basic functionality for a Dynabook is emerging in Squeak, it's not all in a form that any user can use. Reading and writing are skills that most people learn, and pen-and-paper can be used by just about everyone. A Dynabook should enable people to read and write personal dynamic media, and the creation tools should be as easy (and omnipresent) as pen-and-paper.

But even once all that functionality is available, there is a question of what's possible and how it will be used. In books, we have standard features like page numbers and table of contents. What will the equivalent standard practices be in computer-based media? Web pages are slowly developing standards (e.g., the common navigation links across the left hand side or the top of a page), but the Web is not nearly as dynamic as the Dynabook may become.

It's commonly stated that Thomas Edison invented the motion picture but D.W. Griffith invented film. The distinction highlights the difference

between the technology and its use. Edison created the tools with which Griffith created editing and filming techniques that are in common use today in everything we see on television or in the movies. This chapter highlights some of the technologies that can lead to the creation of a Dynabook, but these only suggest the potential available in its use. How a Dynabook is used and what can be done with it are important questions when considering what the impact of personal computers can be.

References

The Swiki on Swikis is at http://pbl.cc.gatech.edu:8080/myswiki.1

A new kind of Swiki is available at http://seaweed.cc.gatech.edu:8080/