1 Building Morphic User Interfaces

The real strength of Morphic lies in creating Morphic interfaces within Morphic. Morphic interfaces don't necessarily have to follow the MVC paradigm, but they can. Morphic interfaces can also be assembled rapidly by simply dragging and dropping them. We have already seen that one morph can be *added* to another. From within Morphic, we say that one morph can be *embedded* within another.

In this section, we'll explore how to work with morphs from the user interface perspective, and then from the programmer's perspective. We'll use the same example, a simple simulation of an object falling, to explore both sides. Along the way, we'll describe the workings of Morphic.

1.1 Programming Morphs from the Viewer Framework

The Viewer framework (sometimes called *etoys system*) has been developed by Scott Wallace of the Disney Imagineering Squeak team as an easy-to-use programming environment for end users. It's not a finished item, and it may change dramatically in future versions of Squeak. But as-is, it provides us a way of exploring Morphic before we dig into code.

We're going to create a simulation of an object falling. Our falling object will be a simple EllipseMorph. Our falling object will have a velocity (initially zero) and a constant rate of acceleration due to gravity. We'll just use pixels on the screen as our distance units.

If you recall your physics, the velocity increases at the rate of the acceleration constant. For our simulation, we'll only compute velocity and position *discretely* (i.e., at fixed intervals, rather than all the time the way that the real world works). Each time element, we'll move the object the amount of the velocity, and we'll increment the velocity by the amount of the acceleration. This isn't a very accurate simulation of a falling object, but it's enough for demonstration purposes.

For example, let's say that we would run our discrete simulation every second. Let's say that velocity was currently 10 and the acceleration was 3. We say that the object is falling 10 pixels per second, with an acceleration of 3 pixels per second per second (that is, the velocity increases by 3 pixels per second at each iteration, which occurs every second). When the next second goes by, we add to the velocity so that it's 13 pixels per second, and we move the object 13 pixels (because that's the velocity). And so on.

We'll also create a *Kick* object. When the object is kicked, we'll imagine that the object has been kicked up a few number of pixels, and it's velocity again goes back to zero. Strictly speaking, an upward push on the falling object would result in an upward velocity that would decrease as

gravity pulled the object back down. Again, we're simplifying for the sake of a demonstration.

Create three morphs (from the *New Morph* menu, or from the Standard Parts bin, or from the Supplies flap): A **RectangleMorph** (default gray), an **EllipseMorph** (default yellow), and a **TextMorph** (appears in Supplies and Parts as "Text for Editing"). We're going to use the rectangle and text as our Kicker, and the ellipse as our falling object.

We'll start out by creating our Kicker button. Click on the text so that you can edit it, and change it to say "Kick." Now Morphic-select it, and drag it (via the black *Pick Up* halo) into the rectangle (Figure 1). Use the control-click menu to *embed* the text into the rectangle. After you choose the *embed* menu item, you will be asked to choose which morph you want to embed the text into. Choose the **RectangleMorph**. (As we'll see later in this chapter, the other option, a **PasteUpMorph**, is actually the whole Morphic world. It is possible to embed morphs into the desktop of a Morphic World.) Once embedded, they move as one morph (Figure 2).

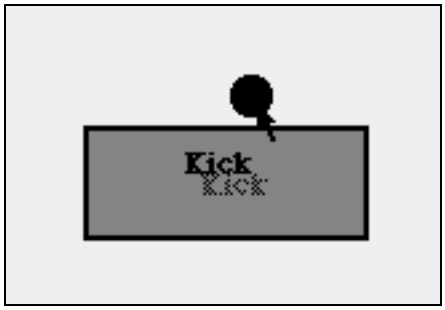

Figure 1: Dragging the TextMorph into the RectangleMorph

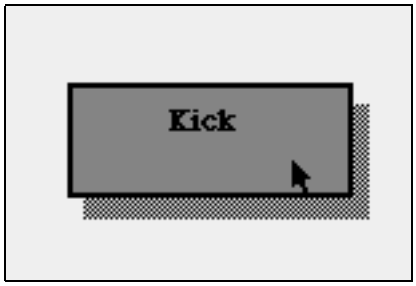

Figure 2: Once Embedded, They Drag Together

Now, let's start programming our two morphs. Morphic-select the ellipse and choose the center left (turquoise) halo, the *View me* halo. When you do, a *Viewer* for the ellipse will open (Figure 3).

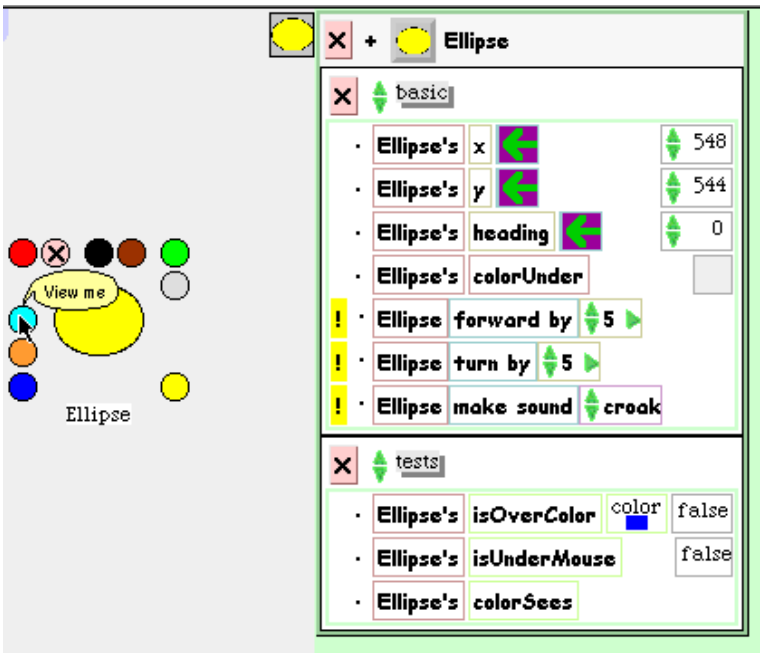

Figure 3: Opening a Viewer on the Ellipse

The Viewer is a kind of browser on a morph. It allows you to create methods for this morph, instance variables for the given morph, and to directly manipulate the morph. Click on one of the yellow exclamation points—whatever the command is (say, *Ellipse forward by 5*) will be executed, and the morph will move five pixels. Directly change the number of the x or y coordinate, and the morph will move.

For what we want to do, change the heading of the ellipse to 180. That means, it's heading will be straight down. That's important because objects fall down. If the heading were zero, our object would fall up.

1.1.1 Adding an Instance Variable

We are going to need a velocity for our falling object, so let's add an instance variable to our ellipse. Click on the small tile of the ellipse inside the viewer itself. (The leftmost tile of the ellipse in Figure 3 is actually a tab. Click on it, and the viewer will slide to the right. Click it again to open the viewer back up.) A pop-up menu will provide a number of programming items, including adding a new instance variable (Figure 4). Choose *add a new instance variable* and enter the name as *velocity.*

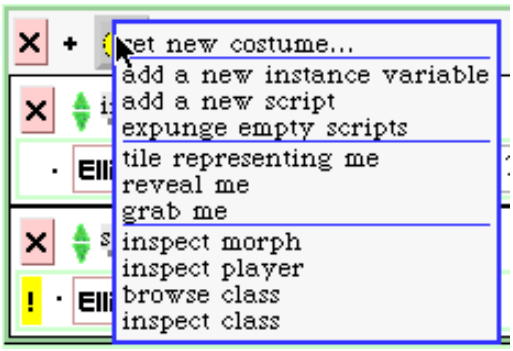

Figure 4: Adding an Instance Variable to a Morph

> ***SideNote***: Take note of what we're doing here: We're adding an instance variable *directly to an instance*, not to the *class*. The Viewer system offers a different kind of object-oriented programming, called *Prototype-based objects*. Each of the morphs is a prototype that can be given variables and methods *directly*. It is possible to then create new instance morphs from these prototypes, and the new morphs will inherit the variables and methods (called *scripts* in the Viewer system). We won't be going that far into Viewers in this book.

The viewer will then update to show the new instance variable (Figure 5). This instance variable can be accessed or set, just like any other instance variable. In a few steps, we'll use it in an equation for changing the velocity by the amount of a gravitational constant.

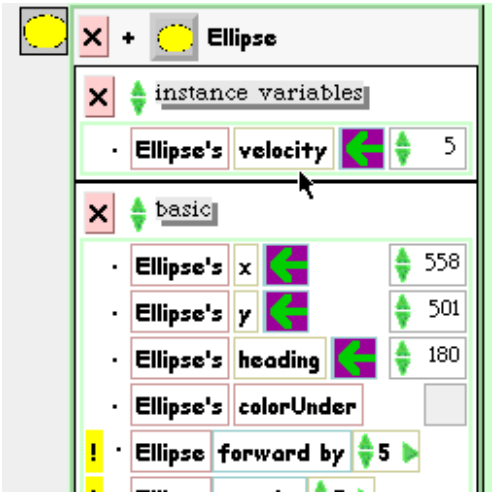

Figure 5: Ellipse's Viewer with the new Velocity Instance Variable

1.1.2 Making our Ellipse Fall

We can then begin to program our falling object. Click on the “forward by” tile and drag it off the viewer.

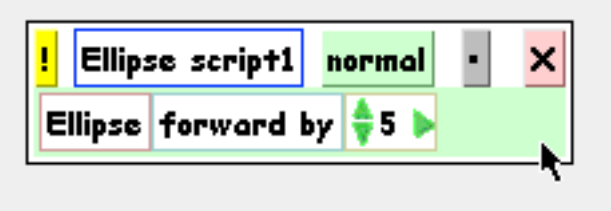

Figure 6: Creating Our First Viewer Script

Let’s make this script run all by itself. We’ll trigger it upon clicking the mouse down upon the ellipse. Click and hold on the word *normal.* You’ll get a pop-up menu of the conditions on which the script should run (Figure 7). Choose *mouseDown* (Figure 8).

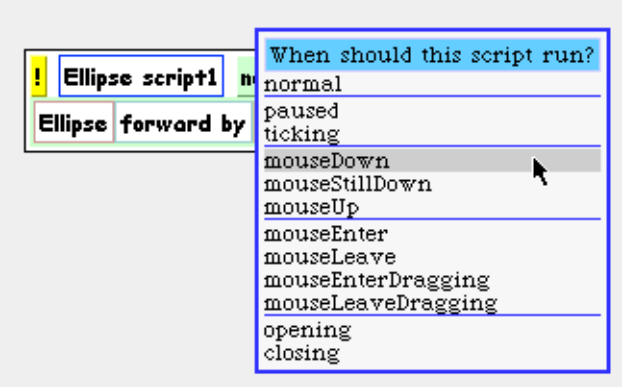

Figure 7: Changing the Conditions of the Script

Figure 8: How the Script Window Changes

Now, click on the ellipse. Each time that you click on it (actually, as soon as you click down on it), it should jump forward five steps. You can play with the amount of the jump in the script1 window to get different amounts of jump.

When an object falls, it should move as much as its velocity, using the simplified model of physics that we’re using. So, instead of the constant in the script, we need to reference the velocity instance variable that we’ve built. That’s fairly easily done. Click on the velocity tile in the ellipse’s Viewer, and drag it over the constant in the script (Figure 9). Now, when

you click down on the ellipse, it moves forward as much as the value of the velocity.

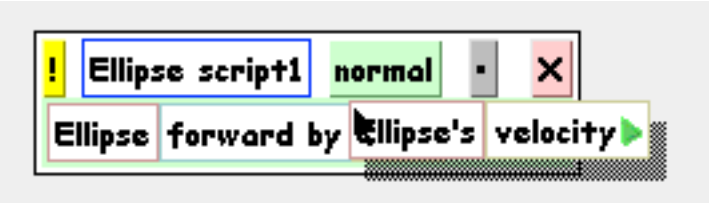

Figure 9: Dragging the Velocity over the Constant

The next step is to make the velocity increase at each time interval. Go back up to the Viewer and click-and-drag on the arrow next to the velocity. You're now grabbing a set of tiles for *setting* the velocity. Drag them into your script window, just above the *forward by* tiles. (You'll find that the other tiles literally move out of your way as you drag in your tiles.) You'll now be setting the velocity to 1 (Figure 10). Now click on the little green arrow next to the 1. The line will expand to *1 + 1* (Figure 11). Go back up the Viewer and drag the velocity instance variable tile over the second 1 (Figure 12). You've now constructed the falling script. Your rate of acceleration is 1, and velocity will increase by it at each time interval.

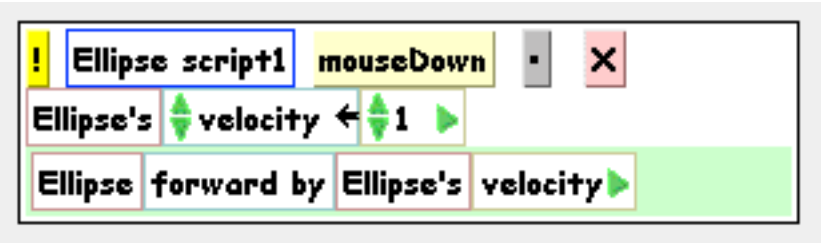

Figure 10: Setting Velocity to 1

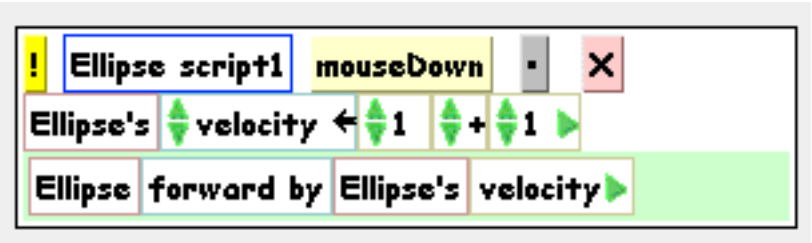

Figure 11: Setting Velocity to 1 + 1

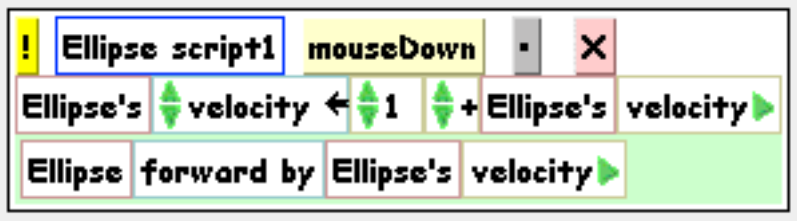

Figure 12: Setting Velocity to 1 + Velocity

You can really make this work now. Change the *mouseDown* trigger on the script to *ticking*. A ticking script fires continuously at a regular interval. (You can change the interval by clicking on the *Ellipse script1* tile and choosing the menu item there.) You will find your ellipse falling ever more rapidly toward the bottom, and then bounce when it gets to the bottom. (That's default Viewer behavior.) You can set the script back to triggering *normal* (which means that it just sits) to stop the falling and to be able to move the ellipse elsewhere.

Feel free to explore different values than 1 for the acceleration constant. You can make small changes by clicking on the up or down arrows next to the 1, or click right on the 1 and type whatever you want. Be careful how large you make it, though! Remember that this value is the amount of change of the *velocity*, so it compounds quickly.

If you want, you can now *name* your script. Click on the *Ellipse script1* tile, and choose *Rename this script* (Figure 13). You might call it *Fall*.

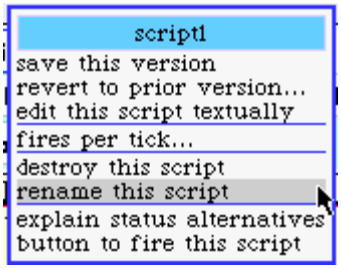

Figure 13: Changing the Name of a Script

1.1.3 Building the Kicker

Now let's build the kicker. Open up a Viewer on your kicker rectangle. Drag out the tile that has the rectangle making a sound, and drop it to make a new script. With this start, whenever we "kick" the ellipse, a sound will be made. Feel free to use the up and down arrows on the sound tile to explore other sounds, and pick the one that makes sense as the "kick" sound to you. (Next chapter, we'll talk about how to record new sounds to use in the sound tile.) Go ahead and make this script work on *mouseDown*. You can click the kick rectangle to hear the sound.

When we kick the object, we should move the object up a few pixels (the effect of our kick), and we should set the velocity to zero. Your final script should look like the top of Figure 14. Set the kicker's script to fire on *mouseDown* and the falling object's script to fire on *ticking*, and you should have a working simulation of a falling object that you can kick.

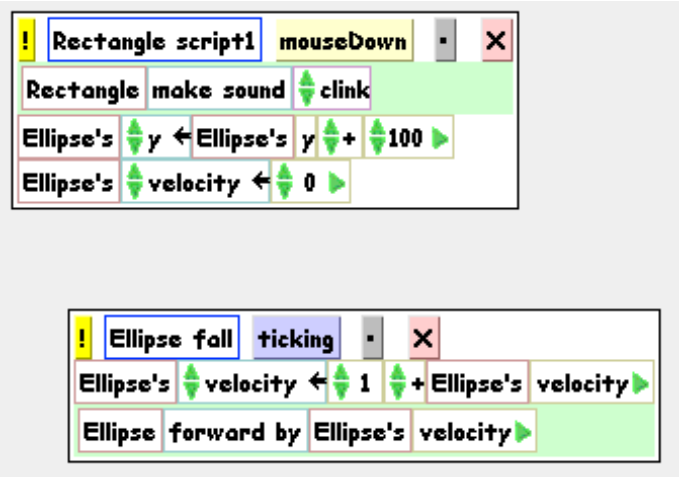

Figure 14: The Final Scripts

You can save these morphs and share them with others as-is. Control-click on any of the morphs and choose *Save morph in file*. You can name the file, and its file extension will be ".morph". You can send this file to others (via email or even on the Web). Others can load it back in to their image. From the file list, when you select a morph file, your yellow button menu will let you file in the morph and recreate it—scripts and all.

The references between objects may get messed up in this process. For example, the kicker's script will probably need to remapped to the falling object. That's what the *Make A Tile* halo (just under the Viewer halo) is good for. Simply make a tile and drag it into each of the "Ellipse" tiles in the kicker's script.

Exercises: Improving the Viewer Falling Object

9. Should the kick script belong to the kicker or the falling object? We currently have it as the kicker, but maybe the falling object should figure out how it should fall, and the kicker should just tell the falling object to fall. Rebuild the system that way.

10. Our velocity is really the *vertical* velocity. Add *horizontal* velocity to the object. Create a launcher that fires out the falling object at a given vertical and horizontal velocity. If you do it right, the object should fall in an arc. (Remember why from your physics?)

11. Remembering your physics, figure out how you need to set things up, without changing the kicker, such that kicking the object stops it dead.

12. How would you make the falling object fall *up*, that is, fall as if the gravitation pull was from the top of the screen rather than the bottom? (Hint: The gravity's impact in our simplistic simulation is through the acceleration on the object.)

13. Brainstorm a bit over class-based versus prototype-based object systems. When is one an advantage over the other? Consider at least these two scenarios: (1) When prototyping a new object and (2) when maintaining objects that were designed five years ago.

1.2 Programming Basic Variables and Events of Morphs

The previous section gave you a sense of how easy it can be to manipulate morphs. For working through how you want your interface to work, this is a great process. You can quickly assemble a morph that you want, and even test out functionality. However, it gets hard to make many of them, or to create abstractions over them (e.g., subclasses, abstract classes), or to control things like connections between objects. Also, the Viewer system doesn't yet provide all the tools of the text-based programming, such as a debugger.

Typically, you still want to use text to build your more complex systems. The transition between the tiling world and the scripting world isn't as complex as you might think. If you click on the *script1* tile, you get a pop-up menu that allows you to view your script textually (Figure 15). This provides you the opportunity to see what the mapping is from the Viewer system into the text world.

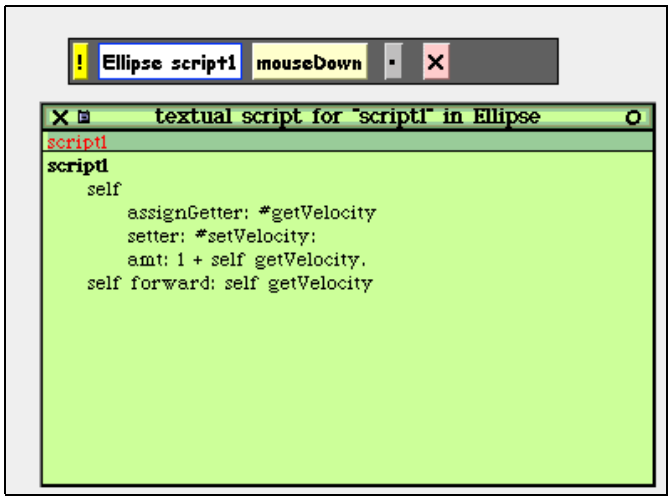

Figure 15: Viewing a Tile Script as Text

But the text world is clearly more complicated than the tile world. We need to know some more things about Morphic in order to dig into programming there. This section introduces the key instance variables, events, and methods needed to program in Morphic.

1.2.1 Instance Variables and Properties

The below table summarizes the main instance variables that are common to every morph. Each of these can be set and accessed using the normal Smalltalk conventions. The **bounds** is accessed using the **bounds** method and set using the **bounds:** method. One of the interesting thing about Morphic is that any change is immediately apparent in the system. Changing the **bounds** makes the morph change its size immediately. You don't have to do any kind of refresh to make it happen.

Instance variable	Meaning
bounds	The rectangle defining the shape of this morph. Change the bounds to resize or move the morph. (**fullBounds** is the bounds of the morph and all of its submorphs. They're most often the same.)
owner	The containing morph. It's **nil** for the World, but is otherwise the morph in which **self** is embedded.
submorphs	The morphs inside me, typically changed with **addMorph**:
color	The main color of the morph.
name	Morphs can be named, and that's what shows up at the bottom of the halows when you Morphic-select an object.

The last instance variable, **name**, is a bit of a trick. Yes, you can use **name**: on any morph, but if you look at the definition of the class **Morph**, you won't find **name** there. Instead, there is another instance variable named **extension** that refers to an instance of **MorphExtension**, and it is the **MorphExtension** that knows how to be named.

What's going on here is a cost savings technique. Every object on the screen in Morphic is a morph. Morphs must therefore be cheap to have around. Thus, extra things like **name** (not every morph needs a name) are *extensions*. If you set the name of a morph, it will check to see if it has an extension, and create one if it doesn't (see the **Morph** method **assureExtension**), then set the name in the extension. The name accessor asks the extension for the name. This is an example of the *delegation* introduced in Chapter 2.

MorphExtension provides many other instance variables, some of which are:

Instance variable	Meaning
balloonText, balloonTextSelector	Any morph can do **self extension balloonText: 'This is all about me…'** and will set the balloon help for themselves. A morph can also set its **balloonTextSelector** which will be used to access balloon text dynamically.
visible	Determines whether a morph is visible or not
locked	Manipulate with **lock** and **unlock**. A locked morph can't even be selected.
sticky	A sticky morph can't be moved. Change it with **toggleStickiness**.

There are other interesting instance variables in **MorphExtension**, but these are the most critical, save one: **otherProperties**. There is built-in space for additional properties in **MorphExtension**, without having to add additional instance variables.

otherProperties is a **Dictionary**. You can add properties with **setProperty:toValue:** and retrieve them with **valueOfProperty:** and ask if a property were there with **hasProperty:**. The name of a property is typically a symbol, and the value can be anything you want. The properties won't be as fast to access as an instance variable, but this allows for great expandability without ever changing the basic structure of **MorphExtension** instances.

1.2.2 Morphic Events

Programming user interfaces in Morphic is much easier than under the MVC window model. Conceptually, the complicated controller part is built into the toolkit. A handful of predefined user interface events are passed on to morphs that want them. The basic model is that a morph is asked if it would like to handle a particular kind of event, and if so, the event is sent by calling a predefined method in your morph. (**Morph**, of course, defines all of these and will catch them if your subclass doesn't override them.)

The object passed around is a **MorphicEvent**. A **MorphicEvent** understands many of the same things as **Sensor**, but encapsulates the event into an object. You don't poll **MorphicEvent** the way that you do **Sensor**. Instead, you can ask a **MorphicEvent** whether **redButtonPressed** is **true** if it's a mouse event (**isMouse** would

return true), or you can ask the **MorphicEvent** what the **keyCharacter** is (if **isKeystroke** is true).

The below table summarizes how to handle the most common kinds of events.

Event you want your morph to handle	How to handle it
MouseDown	Have a method **handlesMouseDown**: which takes a **MorphicEvent** as input, and return **true**. Have a method named **mouseDown**: which takes a **MorphicEvent**, and deal with the mouse down as you wish.
MouseUp and MouseOver (mouse passes over the object)	Similarly, have a **handlesMouseUp**: or **handlesMouseOver**: method, then a **mouseUp**: and **mouseOver**: method.
MouseEnter and MouseLeave	Return **true** for **handlesMouseOver**:, then define **mouseEnter**: and **mouseLeave**:
MouseMove (within the morph)	Return **true** for **handlesMouseDown**: then implement **mouseMove**:
Key Strokes	When your morph should capture keystrokes, return **true** for **hasFocus**, then accept events in **keyStroke**: When the focus is changing, your morph will be sent **keyboardFocusChange**:, **true** for receiving and **false** for losing.

There are more subtleties to the Morphic event handling model. For example, if a morph's extension defines an **eventHandler**, then your events can be delegated to the object referenced by the **eventHandler**. There are also events associated with mouse clicks starting text entry or not, accepting drag-and-drop, and catching whether the mouse is already carrying an object when it enters the bounds of the morph. More details on these can be found in the event handling category of **Morph** instance methods, but the above are the most common cases.

1.2.3 Animation

One of the most interesting things about Morphic is that it makes animated user interfaces very easy to build. To make your morph animate, you need

to implement just one method, **step**, and optionally one other method, **stepTime**.

- At regular intervals, the method **step** is called on all morphs. In your morphs' **step** methods, you can change the appearance, update the display, poll a model to ask for its current values, or do whatever else you'd like.
- The default step interval is once a second. **stepTime** can return a different value, which is the number of milliseconds between each time you want **step** to be called.

An easy-to-understand example of using **step** and **stepTime** is the **ClockMorph**. The **ClockMorph** is a subclass of **StringMorph**, and all it does is display the time. The **stepTime** method simply returns 1000—the clock updates once a second (1000 milliseconds). The **step** method simply sets the contents of the string (**self**) to the current time. That's all that's needed to create an updating string with the time.

1.2.4 Custom menus

There is a custom menu associated with each morph, available from the control-click menu and from the red halo menu. You can easily add morph-specific items to this menu, by overriding the method **addCustomMenuItems: aCustomMenu hand: aHandMorph**. This method is called whenever the menu is requested by the user (via control-click or red-halo click). Simply use **add:action:**, **add:target:action:**, and **addLine** methods to add additional items to the menu being handed to the method.

Most of the time, you will want to allow your morph's superclass a chance to add its menu items, via **super addCustomMenuItems: aCustomMenu hand: aHandMorph**. But if you'd like to limit the menu items that a user sees, you don't need to call the superclass. The menu will still have many generic **Morph** items in it, though.

For an example menu customization, **ImageMorphs** provide user-accessible manipulations through this method.

addCustomMenuItems: aCustomMenu hand: aHandMorph

```
	super addCustomMenuItems: aCustomMenu hand: aHandMorph.
	aCustomMenu add: 'choose new graphic...' target: self action: #chooseNewGraphic.
	aCustomMenu add: 'read from file' action: #readFromFile.
	aCustomMenu add: 'grab from screen' action: #grabFromScreen.
```

1.2.5 Structure of Morphic

The Morphic world may be clearer if some of the internal structure is described. It's important to realize that, just as everything in Squeak is an object, everything in Morphic is a morph (i.e., an instance of a subclass of **Morph**). This includes the desktop itself and even the cursor.

The desktop itself, the World, is an instance of the class **PasteUpMorph**. There are many **PasteUpMorph**s around. The Standard Parts Bin and the flaps are also **PasteUpMorph**s. **PasteUpMorph**s are general "playfields" (as some of them are named) which can hold other morphs.

The World **PasteUpMorph** does something very important: It runs **doOneCycleNow** repeatedly. This method updates the cursors, processes user interface events for the given cursor, runs step methods, and updates the display. The method **doOneCycleNow** appears below:

doOneCycleNow

```
"Do one cycle of the interactive loop. This method is called repeatedly
when the world is running."


"process user input events"
self handsDo: [:h |
        self activeHand: h.
        h processEvents.
        self activeHand: nil].


self runStepMethods.
self displayWorldSafely.
StillAlive ← true.
```

Notice that the above paragraph (and above code) make it clear that events are handled *for each cursor*. A Morphic world can have multiple cursors at once. Each is an instance of **HandMorph**. It is **HandMorph** that sends the events to morphs. Because of this implementation, it is possible to have multiple users interacting in the same Morphic world. There is an option under the *Help* menu from the World Menu called *Telemorphic* which lets you connect multiple users to the same image each with their own cursor.

The **HandMorph** provides many core behaviors to Morphic. As can be seen in the above code, it's the **processEvents** method in **HandMorph** which deals with sending the appropriate messages to the appropriate morphs when user input comes in. It's also the **HandMorph**

which creates the control-click menu, in the method **buildMorphMenuFor**:. The **HandMorph** puts up the halos, builds the halo menus, and even builds the World Menu. So, if you want to change the halos or core menus of the system, you start by modifying or subclassing **HandMorph**.

The process of displaying the world safely (**displayWorldSafely**) leads to asking each submorph of the world to **drawOn**: the world's **Canvas**. The **drawOn**: method is the hook for creating your own look to Morphs, if you want something different than a composition or slight modification to the base morphs. **drawOn**: takes a **Canvas** object as its argument. An instance of **Canvas** knows how to draw basic objects (like rectangles and ovals) as well as draw arbitrary **Form**s.

1.3 Programming A Morphic Falling Object

Let's re-do the falling object simulation, but this time, from textual Squeak. The idea is to create the same kind of interaction as the Viewer version, but using the Morphic programming structure described in Section 1.2. By creating a textual version, we have objects that we can later build upon in other contexts. This code is on the CD as **programmedFall.cs**.

We won't go through a CRC Card analysis here, because we already know what basic objects we want. We need a kicker and a falling object. We will shift responsibilities a bit from the Viewer version: It's the falling object that knows how to be kicked. The kicker just tells the falling object to kick.

Because the textual version will not have the code as accessible as the Viewer version, we'll need to add some user interface to do the kind of exploration that a user might want to do. Probably the most common manipulation will be to change the gravitational acceleration constant. In terms of responsibility, it seems natural to let the falling object hold a menu item for allowing the user to change the gravitational constant. But given that our falling object will be moving constantly, it's easier on the user to stick it in the kicker.

Just to make the falling object a little more interesting, we'll create it as a subclass of ImageMorph. An ImageMorph can hold any kind of Form, which means that we can have any kind of falling object we may wish. Think about what kind of images you might want to have crashing on your screen, with a clear user interface for kicking those objects.

A UML diagram of our classes appears in Figure 16. We'll create a **KickButtonMorph** as our kick button, and a **FallingImageMorph** as our falling object. The **KickButtonMorph** will keep track of the **ball** that it kicks. It will have hooks into the user interface, for the gravity-

setting menu item (**addCustomMenuItems:hand:**) and for capturing button clicks (**mouseDown**:). The **FallingImageMorph** will keep track of its **gravity** (more correctly, the constant acceleration due to gravity) and **velocity**, provide setters and getters for these, and implement a **kick** method. It will have a **step** method where it will implement falling.

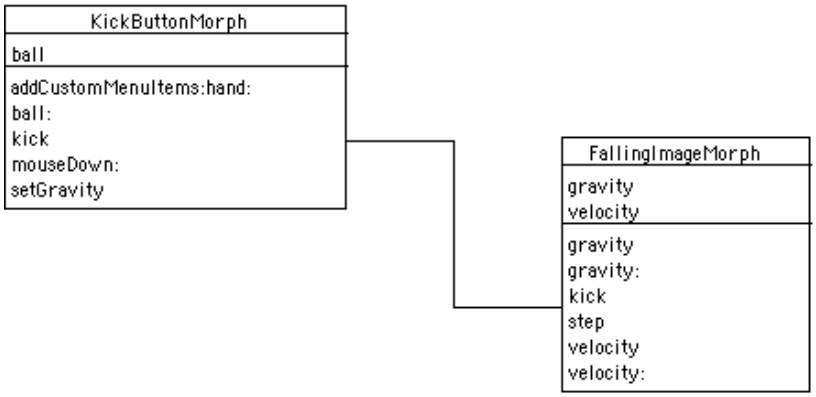

Figure 16: UML Diagram for Textual Falling Object Simulation

We can now begin implementing our classes with some class definitions. While we said that the falling object would be a subclass of **ImageMorph**, we didn't talk yet about what the kicker would be subclassed from. A good solution is to do in code just what we did via direct manipulation of the morphs: We'll start from a **RectangleMorph**. We'll override **initialize** so that our **KickButtonMorph** gets the label that we want.

Notice that there *is* a **SimpleButtonMorph** that would make sense to subclass from. Similarly, there are many button subclasses that would be useful to explore and subclass. However, they make it *too* easy—if we used one of those, we would never deal with mouseDown or setting our own label. We would only provide an action method. While that's what you'll do in normal practice, we'll unpack the details a bit here to show better how the button is constructed.

```
ImageMorph subclass: #FallingImageMorph
        instanceVariableNames: 'velocity gravity '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Morphic-Demo'
RectangleMorph subclass: #KickButtonMorph
        instanceVariableNames: 'ball '
        classVariableNames: ''
```

```
poolDictionaries: ''
category: 'Morphic-Demo'
```

1.3.1 Implementing the Falling Object

Let's start out by implementing the basic falling procedure. We know what this looks like from our Viewer implementation, and we know from our discussion of Morphic animation that we fall in a **step** method. Falling is a process of incrementing the velocity by the acceleration due to gravity, and then moving the object down by the amount of its velocity.

step

```
velocity ← velocity + gravity. "Increase velocity by gravitational constant"
self bounds: (self bounds translateBy: (0@(velocity))).
```

As mentioned earlier, the position and size of a morph is determined by its **bounds**. If we move the **bounds**, we move the object. The **bounds** is a **Rectangle**. To move a rectangle is to *translate* it, and the method **translateBy**: handles the translation. The amount of translation is a **Point**: The amount of horizontal translation and the amount of vertical translation. To move an object down, then, we translate it by **0 @ velocity**.

We don't want the step to happen too often, so we'll provide a **stepTime** method. We'll use one second as the step interval, so that our velocity is in the simple units of pixels per second, and our gravity constant is pixels per second per second.

stepTime

```
"Amount of time in milliseconds between steps"
^1000
```

Next, we need the ability to kick the object. Kicking, as we defined it earlier, sets the velocity back to zero and moves the object back up 100 pixels. Again, this is a translation, where the vertical coordinate is negative because it's a move up.

kick

```
velocity ← 0. "Set velocity to zero"
self bounds: (self bounds translateBy: (0@(100 negated))).
```

Finally, let's provide an initialize method that sets the velocity and acceleration to a reasonable state.

initialize

```
super initialize. "Do normal image."
velocity ← 0. "Start out not falling."
```

```
gravity ← 1. "Acceleration due to gravity."
```

We will need methods for getting and setting the gravity, if not the velocity, too. Those are left as an exercise for the reader.

1.3.2 Implementing the Kicker

The main requirement for the kicker is that it be able to kick an object, so let's begin with that. We'll trigger the kicking action on mouse down, which means that we have to announce that our morph will handle mouse down, then provide a **mouseDown**: method.

```
handlesMouseDown: evt

    "Yes, handle mouse down"

    ^true

mouseDown: evt

    self kick.
```

Kicking is pretty easy when the kicked object implements the kicking.

```
kick

    ball kick.
```

That's enough to allow for kicking. We'll need an ability to set the ball to be kicked (**ball**:), but that's actually enough to start our simulation. However, if we created our objects right now, our kicker would only be a raw rectangle without a label. If we want to have a different look, we should override the **initialize** method.

The initialize method first does whatever rectangles do for initialization, then sets up a label. Our label will be a string (StringMorph) saying "Kick the Ball." StringMorph's know their size (extent), so we'll set the kicker's extent to match it. Then we'll add the string into the rectangle, and place the center of the button wherever the mouse is.

```
initialize

    | myLabel |

    super initialize. "It's a normal rectangle plus..."


    myLabel ← StringMorph new initialize.

    myLabel contents: 'KickTheBall'.

    self extent: (myLabel extent). "Make the rectangle big enough for the
label"

    self addMorph: myLabel.


    self center: (Sensor mousePoint). "Put it wherever the mouse is."
```

1.3.3 Running the Text Falling Simulation

In a workspace, we can now run our simulation. We need to create each object, initialize it, and open it in the world. We need to tell the kicker what its ball is. We'll set the form for the falling object to be selected by the user, so when you execute the below code, you'll have to click and drag a rectangle of interesting display before it'll run. (Feel free to replace that with a form of your own choosing.)

```
aBall ← FallingImageMorph new initialize.

aBall newForm: (Form fromUser). "Here's where you select a form"

aKicker ← KickButtonMorph new initialize.

aKicker ball: aBall.

aBall openInWorld.

aKicker openInWorld.
```

With this, you can bounce the ball around (Figure 17). (Though, it probably doesn't look like a ball, unless you selected one.) However, all you can do is bounce the ball here—not much more exploration than that.

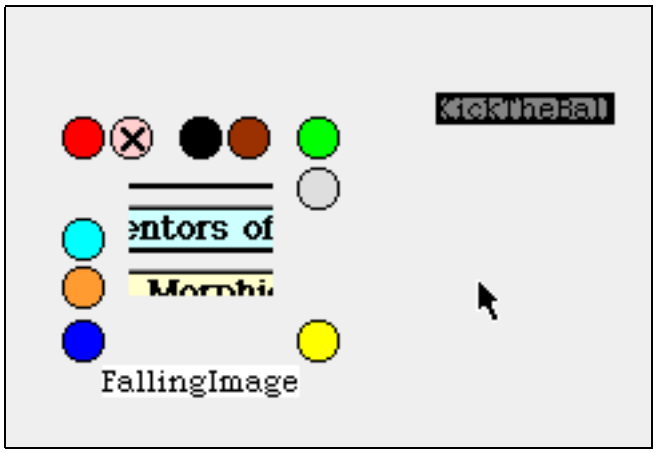

Figure 17: FallingImageMorph and KickButtonMorph

1.3.4 Changing the Gravitational Constant

As seen in our original design, we plan to make a menu item available for changing the gravitational constant for the falling object. We can do that pretty easily. First, we add it to the control-click menu.

```
addCustomMenuItems: aCustomMenu hand: aHandMorph

    super addCustomMenuItems: aCustomMenu hand: aHandMorph. "Do
normal stuff"


    aCustomMenu add: 'set gravity' action: #setGravity.
```

Then, we provide a method for setting the gravity. Setting the gravity will use a FillInTheBlank to let the user know what the current gravity is and to input a new gravity. The gravity is a number, but FillInTheBlank accepts an initial answer and returns a string, so we need to convert.

setGravity

```
"Set the gravity of the ball"
| newGravity |
newGravity ← FillInTheBlank request: 'New gravity'
                    initialAnswer: ball gravity printString.
ball gravity: (newGravity asNumber).
```

Now, try control-clicking on the kicker and changing the gravity for the falling object.