Alice in a Squeak Wonderland

Jeff Pierce

Computer Science Department

Carnegie Mellon University

About this Chapter

This chapter is an introduction to Squeak Alice, an authoring tool for building interactive 3D worlds in Squeak. The first part of this chapter introduces some of the commands that Squeak Alice provides, as well as the ideas behind them. This part of the chapter does not require any previous knowledge of 3D graphics and should be accessible to everyone from Squeak novices to Squeak experts. The second part of this chapter describes the implementation of Squeak Alice, and requires a more advanced understanding of Squeak classes and 3D graphics.

What is Alice?

Printed text, radio, and motion pictures are all different types of media. Although each medium has different strengths and weaknesses, one factor common to all of them is that people use the medium for *storytelling*. Storytelling is one of the oldest and most persistent professions: sooner or later people attempt to use every new medium for storytelling.

Interactive 3D graphics is a new medium that people are still experimenting with for storytelling. The need for expensive and specialized hardware used to be a barrier for potential 3D authors, but the development of inexpensive graphics accelerator cards has largely eliminated this barrier. A significant barrier that still remains is the *authoring* problem: creating an interactive 3D world requires specialized training that most people interested in telling stories do not possess.

Working with 3D graphics typically requires experience with the C or C++ programming languages, as well as familiarity with linear algebra (e.g. 4x4 homogeneous matrix transformations). Unfortunately, the people who possess the skills to work with 3D graphics are not typically the same people who want to use the medium to tell new kinds of stories. To allow the latter group of people to work with interactive 3D graphics you need a new type of 3D authoring tool. The goal of the Alice project was to create an authoring tool for building interactive 3D worlds that is easy for novices to learn and use.

Where did Alice come from?

Randy Pausch and his research group started the Alice project at the University of Virginia. The stated goal of the project was to make it possible for a sophomore Art or English major with little or no programming experience to build an interactive 3D world. We actually

exceeded this goal: we commonly find that motivated high school students, and even some elementary school students, can use Alice.

The first version of Alice ran on Silicon Graphics workstations, but increasingly powerful hardware made it possible for us to port Alice to Windows PC computers at the end of 1995. We made the first public release of Alice at SIGGRAPH 1996, and to date over 100,000 people have downloaded Alice and tried it out for themselves. The current version of Alice is available free from *http://www.alice.org* for Windows 95, 98, and NT.

In 1997 Randy Pausch and some of the original Alice team moved to Carnegie Mellon University (CMU) to form the Stage 3 Research Group (*http://www.cs.cmu.edu/~stage3*). At CMU we continue to develop Alice and learn how to make interactive 3D graphics even more accessible to people. Our current goal is to make Alice easier for younger children by eliminating as much of the typing as possible.

How Squeak Alice got started

The impetus for the Squeak version of Alice, or simply Squeak Alice, was born when Randy Pausch, the director of the Alice project, and Alan Kay, the leader of the Squeak development team, met and exchanged views on how to make authoring in different media more accessible to children. As a result of that meeting, Alan decided that he wanted to take the lessons we learned from developing Alice and implement them in Squeak. To accomplish that goal he asked me to spend a semester interning at Disney Imagineering to implement a Squeak version of Alice. As a Ph.D. student working with Randy and a member of the Alice design team I was familiar with both the lessons we learned from Alice and the structure of the system itself. I leaped at the chance to work with Alan, and after three and a half months of hard work during the 1999 spring semester the first version of Squeak Alice was born.

Using Alice in Squeak

To use Squeak Alice you first need to create a Wonderland. A Wonderland is essentially the collection of all the things you will need to create your own interactive 3D world: a camera window to give you a view into your world and a script editor that allows you load actors into your world, give them commands, and create behaviors for them.

How to create a Wonderland

To create a Wonderland you need to first make sure you are in a Morphic project, and then open a Workspace. To open a workspace first display the World menu by left clicking your mouse, select “open…”, and in the new menu that appears select “workspace”. In the workspace that appears type:

```
Wonderland new
```

and tell Squeak to DoIt (by hitting Alt-D for PC users, Cmd-D for Mac users). A number of windows will pop up as Squeak creates your Wonderland.

The window on the left is the camera window. This window is your view into the 3D world. In this window you will see the effects of the commands you execute. You can also "reach through" this window to manipulate 3D objects in your world using the mouse.

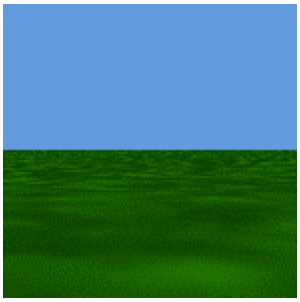

Figure 1: The Camera Window

The window on the right is the Wonderland script editor. The editor is composed of four different parts. On the far left is the object tree, which lists the objects in your scene. Later in this chapter you will learn more about the object tree. To the right of the object tree at the top of the window are three buttons. These buttons choose which part of the editor is active: the script tab, the actor info tab, or the quick reference tab.

The script tab is where you type commands to Squeak Alice and execute scripts. This tab is very similar to a Workspace (DoIt will execute commands, while PrintIt will print out results), but also pre-defines names (e.g. left, green, and camera) for the Wonderland. This chapter will provide lots of sample commands to try. To try out a command, type it into the script tab and execute it using DoIt.

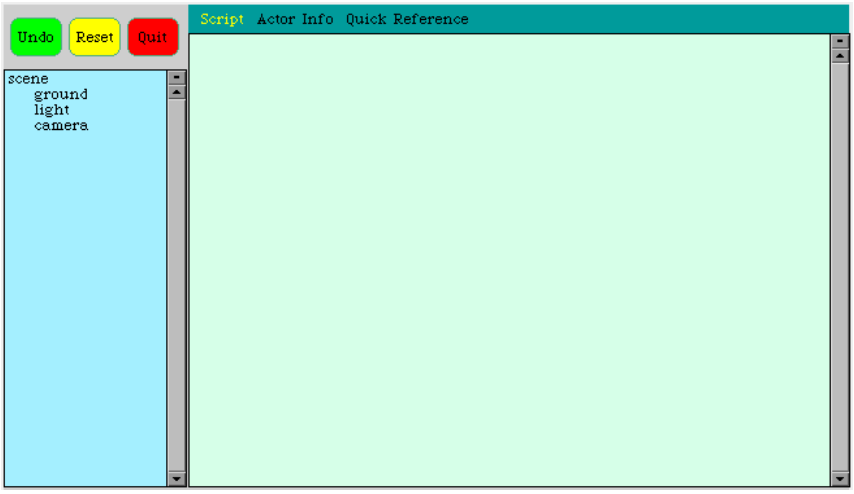

Figure 2: The Script Editor Showing the Script Tab

The actor info tab provides a view of the actor and some current information about that actor. Left clicking on an actor in the object tree determines what actor the actor info tab will display information about.

Figure 3: The Actor Info Tab

The quick reference tab provides examples for many of the commands that Squeak Alice provides. If you forget how to format a command you can check this tab.

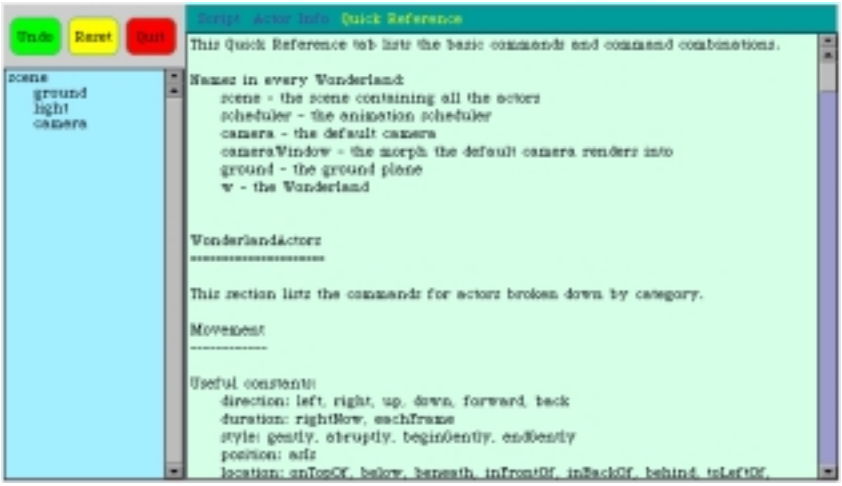

Figure 4: The Quick Reference Tab

Changing the display depth

To run faster and conserve memory Squeak initially uses a low *display depth*. The display depth is the number of bits Squeak uses to represent the color for each pixel. 3D graphics generally requires a higher display depth to look nice, so you will probably want to change the display depth to 16 or 32 bits. To do this, left click to open the World menu, select “appearance…”, then “set display depth…”, and finally choose “16” or “32”.

Creating an Actor

Now that you have created your Wonderland, you need to add some 3D objects to it. In Squeak Alice 3D objects are known as *actors*. Squeak

Alice can create actors from a variety of 3D file formats, including Alice MDL files, 3DS files, and VRML files. To create an actor you need to tell the Wonderland what file to use. For example, to create an Actor from the Alice MDL file bunny.mdl you would type and execute the following command in the script tab:

w makeActorFrom: 'path/to/file/bunny.mdl'

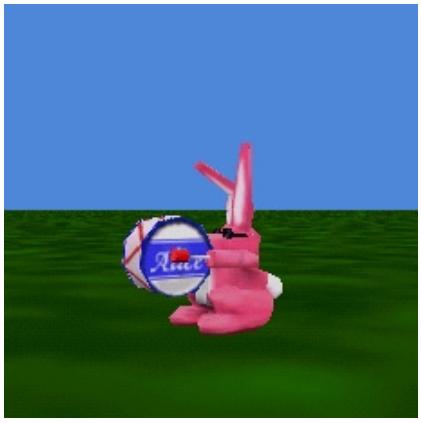

Figure 5: The Wonderland After Loading a Bunny

This tells the Wonderland (w is the name for your Wonderland in the script tab) to create an actor using the bunny.mdl file. By default Squeak Alice will try to name the actor using the file name; in this case it will create an actor named "bunny". If an actor with this name already exists (for example, if you already created an actor using the bunny.mdl file) Squeak Alice will try to name the actor bunny2, bunny3, etc.

As a public service the CMU Alice team has made their library of 3D objects available to the Squeak community free of charge. You can download these objects from:

http://www.cs.cmu.edu/~jpierce/squeak/SqueakObjects.zip

The only restriction is that you must acknowledge that these objects are copyright CMU if you choose to use them.

You can also create actors from other file formats by using the methods **makeActorFrom3DS:** and **makeActorFromVRML:**. One thing to remember when loading 3DS or VRML files is that these formats specify the size of 3D objects using an abstract "unit", so your 3DS model of a car might be, for example, 200 units tall. However, Squeak Alice specifies size in meters rather than some abstract unit, so if you load your car into Squeak Alice it will be 200 meters tall. Luckily you can quickly bring your 3D object down (or up) to size using the **resize:** method that you will encounter later in this chapter.

Simple commands

One of the key lessons we learned from the Alice project is that *vocabulary matters*. Many 3D authoring tools use commands like "translate" and "rotate" to manipulate objects. Unfortunately, these are not the best commands for novices. Consider that, for our target audience, translate is

usually thought of as something you do to a language (e.g. translating from English to French) rather than a 3D object.

Thus, instead of translate and rotate, Squeak Alice users “move” and “turn” objects. However, this only solves part of the problem. Other authoring tools usually also require users to specify directions in terms of the X, Y, and Z axes, yet most people generally do not think in terms of moving objects in the X direction. In Squeak Alice we instead allow users to specify directions using left, right, forward, back, up, and down. Each object then defines its own coordinate system (e.g. the bunny has built in forward, left, and up directions). While not all objects have an intrinsic forward direction, we note that no object has an intrinsic X direction. If an object has no intrinsic forward direction then we assign one arbitrarily.

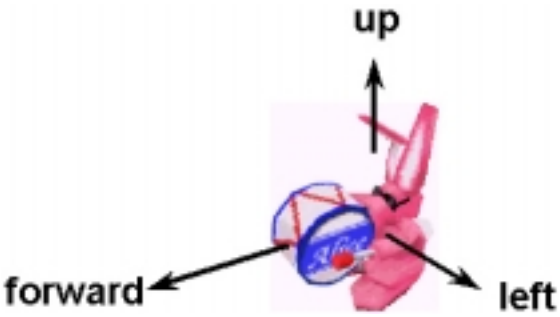

Figure 6: The Bunny's Built-In Directions

Type the following commands into the Wonderland script window and execute them to try moving and turning an actor.

bunny move: forward

bunny turn: left

Both of these commands provide a default amount to move or turn. By default actors will move one meter or turn a quarter turn. Users can also specify the desired amount themselves. For the move: command users specify the distance in meters, while for the turn: command they specify the number of turns. This is another lesson we learned from Alice: novices do not think about turning objects in terms of radians or degrees. Instead, they prefer to think about quarter, half, or full turns.

Try the following commands for yourself, and see what happens when you change the distance or number of turns.

bunny move: forward distance: 2

bunny turn: left turns: 1/2

Working with the mouse

Rather than type commands to objects, users can also manipulate them using the mouse. Simply clicking and dragging with the left mouse button on an actor moves the actor parallel to the ground plane. To move the actor up or down, the user first left clicks on the actor and then holds down the Shift key before dragging. Left clicking and then holding down the Control

key and dragging will turn the actor left or right (around the scene's up vector). Finally, the user can left click on the actor and then hold down both the Shift and Control keys to rotate the actor without any constraints.

Users can also access a menu of commonly used commands. To access this menu the user first left clicks on the actor's name in the object tree to select it, and then right clicks on the object tree. This will bring up a menu of useful commands that will turn the selected object around once, point the camera at it, make it grow, shrink, etc.

Squeak Alice also makes it possible for users to create new responses for actors when users click on them. We will cover how to change an actor's responses later in this chapter.

Camera controls

By default every Wonderland contains a camera, and users can give this camera commands using the commands for actors. For example, try the following:

camera move: back distance: 3

camera turn: left

Users can also maneuver the camera around the world using the mouse. To do this they first need to show the controls for the camera. You can do this by evaluating:

cameraWindow showCameraControls

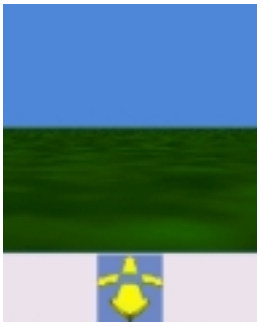

Figure 7: The Camera Window and Camera Controls

To hide the camera controls again, evaluate:

cameraWindow hideCameraControls

Showing the camera controls will show a small morph beneath the camera window that depicts four arrows. To move the camera around the scene you left click in the camera controls morph and then drag the mouse. Moving the mouse up relative to the center of the controls morph will move the camera forward; the further the mouse pointer is from the center the faster the camera will move. To move the camera backward you move the mouse down from the center, while to turn the camera left or right you simply move the mouse pointer left or right of the center. Note that you do not need to release and then re-click to change direction. While holding the left mouse button down you just move the mouse pointer to a different position relative to the center of the camera controls morph.

You can change the way the camera moves by holding down one of the modifier keys. To move the camera up or down hold down the Shift key. To force the camera to only rotate left or right, hold down the Control key. Because Squeak interprets a left click while you are holding down the Control key as a different command you will need to click on the camera controls first and then press the Control key. Finally, you can tilt the camera up or down by holding down both the Shift and Control keys and then left clicking on the controls and dragging the mouse.

Ubiquitous animation

If you are experimenting with Squeak Alice as you read through this chapter you have no doubt noticed that the commands to move and turn the bunny animated over time. All commands in Squeak Alice animate by default over one second whenever it makes sense to do so. This is actually based on a psychological principle: people take a second or two to assimilate any instantaneous change. Given this fact, we can make use of that second to animate the command so that users can watch the command unfold. We find that this tends to make scripts easier for users to debug. For example, rather than instantaneously disappearing off the screen an actor might move out of view to the left, letting the user know where the actor went.

Squeak Alice also allows users to specify durations other than one second. Any command that animates by default also allows the user to explicitly set the duration. For example:

bunny move: left distance: 2 duration: 4

bunny turn: forward turns: 1 duration: 10

There is something else interesting about animations in Squeak Alice. Specifically, these animations are *time* based rather than *frame* based. Users specify how many seconds an animation lasts, rather than the number of frames. There are two reasons for this. First, novices intuitively think about duration in terms of seconds. The notion of a *frame* requires a more detailed understanding of how computer graphics works. Second, the actual amount of time that an animation specified in frames takes depends on the speed of the computer you run it on. On a computer that can render at 30 frames per second, a 30 frame animation will last one second. However, on a computer that can render at 60 frames per second that same 30 frame animation will only last half a second. Animations with a duration specified in seconds, by contrast, will last the same amount of time on both computers.

Zero duration animations and rightNow

Squeak Alice allows you to create an animation with a duration of zero seconds by setting the duration to 0. However, this does not actually cause the command to happen instantaneously. When you evaluate a command with a zero duration Squeak Alice still creates an Animation object for that command.

bunny move: forward distance: 2 duration: 0

This means that Squeak Alice will not evaluate the command until the next time it processes animations, which will be slightly later. This can

cause problems if the next line in your script assumes that the bunny has already moved.

To make Squeak Alice execute a command instantaneously, you need to use the **rightNow** primitive. This tells Squeak Alice to execute the command right away, without creating an Animation object. You use rightNow like this:

bunny move: forward distance: 2 duration: rightNow

Styles of animation

By default Squeak Alice animates commands using an ease-in / ease-out (also known as slow in / slow out) animation style. Therefore if you tell the bunny to move one meter over one second it will not move at a constant speed; instead, the bunny will accelerate to a maximum velocity and then decelerate.

Although this is the default animation style, you can also specify other animation styles using the **style:** keyword. In addition to the default style (known as *gently*), you can also use the **beginGently**, **endGently**, or **abruptly** animation styles. The beginGently style will accelerate to the maximum velocity but will not decelerate gradually, while the endGently style will start at the maximum velocity and then decelerate smoothly. The abruptly style will cause the animation to happen at a constant velocity.

bunny move: forward distance: 2 duration: 2 style: abruptly

bunny move: forward distance: 4 duration: 8 style: endGently

Ubiquitous undo

Part of the goal of the Alice project is to encourage users to explore the possibilities of 3D graphics. Before users will explore, however, they need to feel safe. Specifically, they need to feel like actions will not have irretrievable consequences, so that they can eliminate any changes they do not like. One of the ways we accomplish this is by providing an ubiquitous undo mechanism, so that users can always roll back to a safe state.

Squeak Alice provides a big, green Undo button in the Wonderland Editor. Each time you click on this button Squeak will undo one action or command, starting with the most recent. The Wonderland actually keeps track of all of your previous commands, so if you click the Undo button five times Squeak will undo your last five actions. In keeping with our philosophy of ubiquitous animation Squeak Alice animates the undo operation over one second as well.

Figure 8: The Undo Button

Parent – child relationships

Alice actors are hierarchical objects, which means that they are divided up into different parts with a parent-child relationship. This *parent-child* relationship is important because commands that you give to the parent actor can affect its children. For example, because the head is a child of the bunny, the head will move if you move the bunny.

In Squeak Alice the *object tree*, located underneath the Undo button in the Wonderland Editor, depicts the parent-child relationship between the different actors. The **scene**, located on the top left, is the parent of all the actors in the Wonderland. The **ground**, **camera**, **light**, and **bunny** are the immediate children of the scene, so Squeak Alice displays them one level down and to the right of the scene. The bunny actor is also composed of different parts (a head, body, and drum) that are in turn built of more parts.

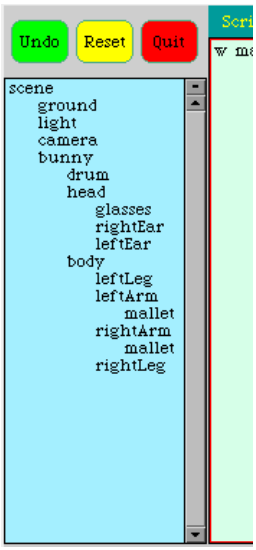

Figure 9: The Object Tree

Because these constituent parts are actors we use the same Alice commands that we do with the whole actor. Try out the following commands:

bunny turn: left

bunny head turn: left turns: 1

bunny drum move: forward distance: 2

The effect of some commands on an actor's children depends on whether a given child actor is a *first class* or *part* object. The easiest way to think about the distinction is a first class object is a complete, separate entity like a table or a book, while a part is a piece of an object (e.g. a table leg).

The reason we created this distinction is that we needed a way for users to be able to decide whether or not property changes to an actor would affect the actor's children. Thus a book might be a child of a table, so that the book moves when the table does, but if the user changes the color of the table the book should remain the same color. The user can create this

behavior by making the book a child of the table and setting both actors to be first class. You can see for yourself how this works by changing the first class or part status of an actor. Try the following:

bunny setColor: green

bunny head becomeFirstClass

bunny setColor: red

bunny head becomePart

bunny setColor: blue

The first class or part status of an actor also affects how that actor handles events, as we will see shortly.

In addition to changing the first class or part status of an actor, you can change the parent of the actor. The following makes the bunny's head a child of the ground:

bunny head becomeChildOf: ground

Now if you click on the bunny and drag you will move the bunny's body and drum, but his head will remain in place. The bunny's head is now a part of the ground. This also causes the name of the head actor to change. Because the head is a child of the ground, the actor's name changes to **ground head**. You can give it commands using this new name:

ground head turn: left

Advanced commands

The Move and Turn commands are both the simplest and the most commonly used commands in Alice. Squeak Alice also provides some useful higher level commands for users to work with.

- Squeaks Alice provides an animated **destroy** command that removes objects from the Wonderland. Like all other Alice commands users can Undo this operation if they destroy an object accidentally.

  bunny destroy

  bunny destroy: 4

- The **resize:** command allows users to change the size of objects. Many 3D authoring tools refer to this operation as scaling, but we found that users tend to associate the word "scale" with the notion of weighing. The basic version of resize allows users to specify an amount and duration. More advanced versions allow the user to specify non-uniform resizing and a volume preserving resize.

  bunny resize: 2

  bunny resize: 1/2 duration: 4

  bunny resizeTopToBottom: 2 leftToRight: 1 frontToBack: 3

  bunny resizeLikeRubber: 2 dimension: topToBottom

Figure 10: The Bunny After a resizeLikeRubber

- We found that users intuitively understood using **turn:** to specify yaw (left/right) and pitch (up/down). However, there was no direction that users associated with Turn to describe roll. We implemented the **roll:** command for this operation.

  bunny roll: right

- Squeak Alice provides **moveTo:** and **turnTo:** commands that allow users to specify motion to an absolute position or orientation. The user can specify an absolute position or orientation using either a numerical triple or a reference object. The triples use a {Left. Up. Forward} notation and describe positions or orientations in the actor's *parent's* reference frame. For example, for the bunny {0. 2. 0} is the point two meters above the origin of the scene's coordinate system, while for the bunny's head this same triple is two meters above the bunny's origin.

  bunny moveTo: {0. 2. 0}

  bunny turn: left

  bunny turnTo: {0. 0. 0} duration: 3

  bunny moveTo: camera duration: 2

  bunny turnTo: camera

- Although users can tell an actor to align with another actor using the **turnTo:** command, we also provide an **alignWith:** command.

  bunny alignWith: camera

- Sometimes rather than aligning one actor with another you want to turn one actor to face the other. The **pointAt:** command provides this functionality: the command rotates the actor so that it is pointed at the specified target. Note that you can specify either another actor or a {Left. Up. Forward} triple as the target.

  camera pointAt: bunny

  camera pointAt: {0. 0. 0} duration: 3

- The **place:** command moves actors to a specific position relative to another actor. The positions that Place supports are

inFrontOf, **inBackOf**, **onTopOf**, **onBottomOf**, **toRightOf**, **toLeftOf**, **onCeilingOf**, and **onFloorOf**.

light place: onTopOf object: bunny

light place: inFrontOf object: bunny duration: 4

- After you have been turning and rolling an actor or the camera you occasionally want to realign the actor with the Wonderland's up vector. This is especially true for the camera, as we found that users quickly become disoriented when the camera is rolled too far left or right. We created the **standUp** command to provide this functionality.

  camera standUp

  camera standUpWithDuration: 4

- The **nudge:** command moves actors in multiples of their length, width, or height.

  bunny nudge: forward

  bunny nudge: up distance: 2 duration: 2

- Occasionally users need to temporarily hide an actor in their Wonderland. The **hide** command will stop Squeak from drawing an object, while **show** makes Squeak start drawing it again.

  bunny hide

  bunny show

- The **playSound:** command will make an actor play a specified WAV file. Squeak Alice creates an Animation object that controls the playback.

  bunny playSound: 'bangdrum.wav'

This is of course not all of the commands that Squeak Alice provides for actors. For a more comprehensive listing you can look at the Quick Reference tab in the Wonderland Editor, or you can look at the WonderlandActor implementation itself.

Beyond time dependent animations

Although the majority of the commands in Squeak Alice create time dependent animations, you can also use Squeak Alice to create more persistent actions. First, you can use the **speed:** keyword to cause actors to move at a constant rate. When you add the **speed:** keyword to the **move:** command, Squeak Alice will move the object at the specified velocity in meters per second. If you specify a **distance:** the actor will move that distance and then stop; if you do not then the actor will move at the constant rate forever (or until explicitly stopped). The **speed:** keyword works similarly with the **turn:** command, but the associated quantity is rotations per second.

bunny move: forward distance: 5 speed:1

bunny move: forward speed: 1

bunny turn: left turns: 2 speed: 1/2

bunny turn: left speed: 1/2

We found that using speed to create a persistent action involving moving or turning made sense to our users. Unfortunately, not all commands naturally involve a speed. We needed a way for users to be able to establish simple constraints like making the bunny's head always point at the camera. For this type of action we created the **eachFrame** parameter. You can use **eachFrame** as the duration to create an action that occurs every time Squeak redraws the Wonderland.

bunny head pointAt: camera duration: eachFrame

In addition to allowing users to supply **eachFrame** as the duration, Squeak Alice also provides **eachFrameUntil:** and **eachFrameFor:** keywords that users can add to commands. The **eachFrameUntil:** command allows users to provide a block context that returns true or false; the command will repeat until the block returns true. The **eachFrameFor:** keyword makes the command repeat for the specified number of seconds. The following command makes use of another parameter, **asIs**, to constrain the bunny to only move along the ground for 10 seconds.

bunny moveTo: {asIs. 0. asIs} eachFrameFor: 10

The **asIs** parameter tells Squeak Alice to leave the current value "as is". Note that this does *not* prevent the value from changing at all, it merely prevents *that command* from changing the value. Thus while the previous command is active the user can left click on the bunny to drag it around (the **moveTo:** command will not cause any change in the bunny's left or forward position). However, if the user holds down Shift while dragging to move the bunny up or down then the **moveTo:** command will always reset the bunny's up position to 0.

Frames of reference

By giving each actor its own reference frame we make it very easy to talk about moving actors forward, up, etc. However, another way that people talk about moving objects in the real world is relative to other objects, e.g. move it to your left. Users can move actors in Wonderlands relative to other actors using the **asSeenBy:** keyword. The following command will move the bunny one meter to the *camera's* left:

bunny move: left distance: 1 asSeenBy: camera

You can also move actors to an absolute position relative to another actor. The following command moves the bunny to a position one meter in front of the camera and one meter above it:

bunny moveTo: {0. 1. 1} asSeenBy: camera

Controlled Exposure to Power

One of the basic design principles we used in the creation of Alice was the notion of *controlled exposure to power*. This principle in essence states that commands should have sensible defaults, so that the user can work with them in their simplest forms and still be able to accomplish useful work. Then as the user becomes more sophisticated he learns how to explicitly specify values rather than relying on these defaults. The user continues to work with the same commands, but those commands advance with user.

For example, the **move:** command can take all of the following forms:

bunny move: forward

bunny move: forward distance: 1

bunny move: forward distance: 1 duration: 1

bunny move: forward distance: 1 speed: 1/2

bunny move: forward speed: 1/2

bunny move: forward speed: 1/2 for: 5

bunny move: forward asSeenBy: camera

bunny move: forward distance: 1 asSeenBy: camera

bunny move: forward distance: 1/10 duration: eachFrame

And that is only *some* of the forms **move:** can take. Even though users can start out by simply specifying a direction, there's a lot of different ways they can use **move:** as they become more advanced users.

Animation Methods

When you give a command to an Actor in a Wonderland Squeak creates a WonderlandAnimation instance that governs the time dependent behavior of that command. You can assign this instance to a variable and access the methods defined on animations. Four useful methods that animations provide are **pause**, **resume**, **stop**, and **start**. **pause** will temporarily stop an animation until you **resume** it. **stop** will stop that animation altogether, while **start** will run the animation again from scratch.

spin := bunny turn: left turns: 20 duration: 40

spin pause

spin resume

spin stop

spin start

You can also cause an animation to repeat using the **loop** command. If you supply a number with **loop:** the animation will repeat that number of times. Otherwise the animation will repeat forever until explicitly stopped. If you want the animation to finish the current iteration before stopping you can use **stopLooping**.

flip := bunny turn: forward turns: 1 duration: 2

```
flip loop: 2
flip loop
flip stopLooping
```

Composing Animations

Squeak Alice provides a set of primitive commands (e.g. **move:** and **turn:**) as well as a set of more advanced commands (e.g. **pointAt:** and **alignWith:**). During the initial design of Alice we realized that these commands alone did not provide the user with the control and flexibility that we wished. We therefore provided a way for users to compose primitive animations to create more complex animations. There are two ways to compose animations: **doTogether:**, which causes the animations to run at the same time, and **doInOrder:**, which runs the animations one after another. These commands are defined by the Wonderland itself, rather than by the WonderlandActors. With the **doInOrder:** command we can, for example, create an animation the causes the bunny to hop up and down.

```
jump := bunny move: up distance: 1 duration: 1/2
fall := bunny move: down distance: 1 duration: 1/2
hop := w doInOrder: {jump. fall}
hop start
```

You can also compose the animations that you create with **doInOrder:** and **doTogether:** with other animations.

```
spinJump := w doTogether: {hop. bunny turn: left turns: 1 duration: 1}
spinJump loop
```

Setting Alarms

Because commands in Squeak Alice are animated over time, each Wonderland needs to keep track of the passage of time. To do this, each Wonderland has a Scheduler instance responsible for keeping track of the passage of time and for updating the animations of commands. When you create a Wonderland the scheduler sets the time to zero and then updates this time every frame. You can find out the current time (in seconds) in a Wonderland by printing the result of:

```
scheduler getTime
```

Squeak Alice makes use of the fact that Scheduler keeps track of the passage of time by allowing you to set *Alarms*. An Alarm is some action (a BlockContext) that you want the Wonderland to execute at a specific time or after some specified time has elapsed.

The two commands the Alarm class provides for this are **do:at:inScheduler:** and **do:in:inScheduler:**. The first takes a BlockContext, the time to execute the action, and the scheduler to add the alarm to.

Alarm do: [bunny turn: left turns: 1] at: (scheduler getTime + 10) inScheduler: scheduler

The second command takes a BlockContext, how time much to wait (in seconds) before executing the action, and the scheduler to add the alarm to. Both of these commands return an Alarm instance.

myAlarm := Alarm do: [bunny turn: left turns: 1] in: 5 inScheduler: scheduler.

The Alarm instance has a **checkTime** command that you can use to determine when the alarm will go off, and a **stop** command you can use to stop the alarm before it goes off.

Making objects react to the user

All of the commands that I have covered so far are useful for creating behaviors for objects, but they do not make the Wonderland interactive. To create interactive behaviors for the actors in a Wonderland you need to use the **respondWith:to:** or **addResponse:to:** methods.

These methods allow you to specify what you want the actor to respond to, and how you want it to respond. Actors can respond to **leftMouseDown**, **leftMouseUp**, **leftMouseClick**, **rightMouseDown**, **rightMouseUp**, **rightMouseClick**, and **keyPress** events. Responses then take the form of a BlockContext that accepts a single parameter. This parameter is a WonderlandEvent instance that Squeak generates to encapsulate data about the event (e.g. what key the user pressed).

The **respondWith:to:** method tells the actor to respond to the event with only the specified response; the actor ignores all other previously defined responses for that event. The following command will make the bunny spin around once every time you click on it with the left mouse button. Notice that clicking and dragging the bunny will no longer move it around.

bunny respondWith: [:event | bunny turn: left turns: 1] to: leftMouseClick

The **addResponse:to:** method will add the new response to any other defined responses for the specified event. The method returns the new reaction, so that you can later remove the reaction using the **removeResponse:to:** method.

newReaction := bunny addResponse: [:event | bunny turn: left turns: 1] to: leftMouseClick

bunny addResponse: [:event | bunny move: forward distance: 2] to: leftMouseClick

Now you can click and drag the bunny around, and when you let up on the left mouse button the bunny will move in a small circle. If you now remove the first response you added, the bunny will only move forward when you let up on the button.

bunny removeResponse: newReaction to: leftMouseClick

Helper Actors

Most of the time users work with actors that have geometry (a polygonal mesh that Squeak Alice uses to create the visual depiction of the actor). When writing scripts, though, users will occasionally need to describe the motion or behavior of an object relative to some arbitrary position or orientation. Consider as a simple example trying to make the bunny orbit around the scene coordinate {0. 0. 2}. The simplest way to do this is to create an actor without any geometry (informally dubbed a *helper actor*):

```
helper := w makeActor
```

Move that helper to the desired position:

```
helper moveTo: {0. 0. 2} duration: rightNow
```

And then rotate the bunny around that point:

```
bunny turn: left turns: 1 asSeenBy: helper
```

Multiple Cameras

Although Squeak Alice provides only a single camera by default, you can create new cameras to provide multiple views into the Wonderland. There is a drawback to creating new cameras: because your processing resources are finite, each new camera decreases the overall frame rate for all camera views. If your frame rate with one camera is F, then your frame rate with N cameras will be approximately F / N.

Despite this drawback, users often find it useful to provide multiple views onto a scene. For example, multiple views can make it easier to find and manipulate actors in the scene. The syntax for making a new camera is:

```
w makeCamera
```

When you execute this command Squeak Alice will create a new camera (named camera2 for the second camera you create, camera3 for the third, etc.) and add it to the scene. New camera windows all start in the same default position, so you may need to move one camera window to see the other. You can either move the camera window morph with your mouse, or you can actually issue Squeak Alice commands to the camera window morphs themselves. The distance units for morphs are pixels.

```
cameraWindow move: down distance: 50 duration: 2
```

Squeak Alice actually represents the position of a camera in a Wonderland using a 3D camera model. Although with a single camera you will never see this model, with two or more cameras you can actually position one camera to view another. You can even left click on a camera model in a Wonderland to move that camera around with the mouse like any other 3D object.

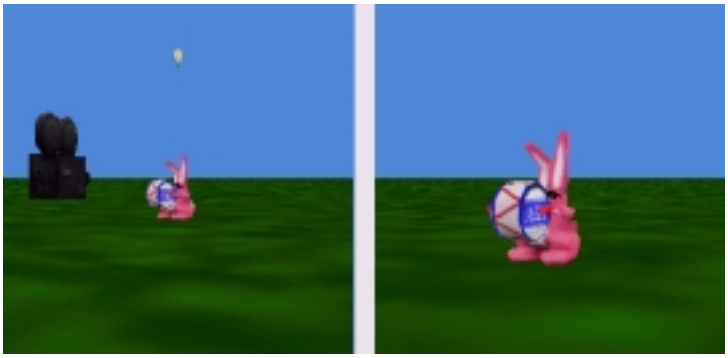

Figure 11: Multiple Cameras Providing Multiple Views

Blending 2D and 3D

Unlike other 3D authoring tools (including the CMU version of Alice), Squeak Alice allows you to smoothly integrate a 3D Wonderland with other 2D content in Squeak. To do this you simply turn off the background in the camera window:

cameraWindow turnBackgroundOff

You may also want to hide the ground to focus more on composing the actor with current Project.

ground hide

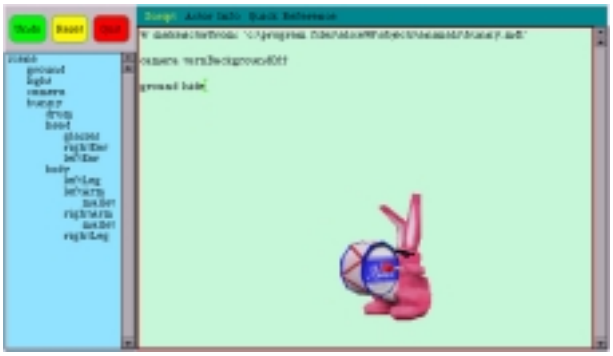

Figure 12: The Bunny Blended With the Script Editor

Assuming you pointed the camera at an actor in the Wonderland, you should now see that character blended smoothly with the Project. To move the actor around the Project you need to move the camera window, not the actor itself. Try the following one at a time to see the difference:

cameraWindow move: right distance: 50

bunny move: right distance: 5

Now that you have created a 3D character integrated with your 2D Project, you can make that actor interact with the Project. Squeak Alice provides commands for converting 2D points to 3D points, as well as commands to change the Z ordering of the camera window morph in the Project. Thus you can, for example, make the bunny's head watch your cursor:

bunny head doEachFrame:

[bunny head pointAt: (camera transformScreenPointToScenePoint: (Sensor mousePoint) using: bunny) duration: rightNow].

This example introduces the **doEachFrame:** method, which allows you to send a snippet of code to an actor to execute each frame, and the camera **transformScreenPointToScenePoint:using:** method, which converts a 2D point to a 3D point using the bunny to determine the relative depth of the 3D point.

Two other useful methods are the **sendInFrontOf:** and **sendBehind:** methods provided by the camera window. These methods allow you to (for instance) make a 3D character wander around (in front of and behind) a 2D morph.

Active textures

Andreas Raab implemented a method of drawing the contents of a morph on a 3D object in a Wonderland that he calls active textures. To create an active texture you need to open the camera window morph to drag and drag, enable active textures for a particular 3D object, and then drop a morph on it.

To open a camera window morph to drag and drop you need to show its menu by holding down Control and left clicking on the camera window morph. Select "open to drag and drop" from this menu.

Next create a 3D object that you want to draw the morph on. It's usually easiest to use a 2D plane. Wonderlands provide a shortcut for creating a simple flat square:

w makePlaneNamed: 'myPlane'

Now you need to enable active textures for your object. Hold down Alt (option for Mac users) and left click on the plane to show its halos. Click on the red halo to show the plane's menu and select "enable active texture". Repeat these steps to show the plane's menu again, but this time select "auto adjust to texture".

You can create a sample morph to use as your active texture by left clicking in your project to show the World menu and selecting "new morph". In the new menu select Demo→BouncingAtomsMorph. Squeak should create a new morph and attach it to your cursor. Now just left click on the plane to drop the morph on top of it, and you should see the bouncing atoms morph appear on the plane inside your Wonderland.

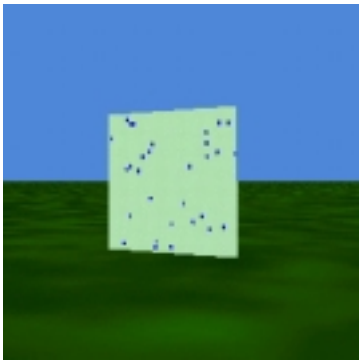

Figure 13: An Active Texture in a Wonderland

Quitting a Wonderland

Squeak creates classes that are specific to each Wonderland and are only used within that Wonderland. As a result, when you want to delete a Wonderland you need to make sure Squeak removes those classes correctly. To do this you click on the red Quit button in the Wonderland Editor; Squeak will delete the Wonderland for you and clean up after it properly.

Instead of deleting a Wonderland you can instead pause it by exiting the project containing that Wonderland. You can also reset the Wonderland to its initial condition (but keeping the script you have written in the Editor) by clicking the yellow Reset button.

The Squeak Alice Implementation

This part of the chapter provides a brief overview of how I implemented Alice in Squeak. My intention is to provide a high level understanding of how Squeak Alice works. Advanced Squeak developers who want a more detailed understanding of how Squeak Alice works should find that this discussion at least provides a framework for reading and understanding the actual code.

Balloon3D

Balloon3D is an *immediate mode* 3D renderer written by Andreas Raab. Balloon3D provides the necessary primitives (lighting, shading, texturing, mesh and matrix transformations, compositing operations, etc.) that are needed to draw a single frame of a 3D scene. However, Balloon3D does not provide any continuity between frames, nor does it have any notion of a hierarchical, persistent scene. Put simply, Balloon 3D knows how to draw triangles, not objects.

Squeak Alice uses Balloon3D to create a scene graph-based *retained mode* renderer. This means that Squeak Alice knows how to draw objects: it creates a 3D world that persists from frame to frame. Squeak Alice organizes the objects in the 3D worlds (lights, cameras, and actors) in a scene graph. A scene graph is a hierarchical structure that describes the relationship between objects in the scene and their properties (color,

lighting, texture, position, etc.). By incrementally modifying the properties of objects in the scene graph each frame Squeak Alice can animate the 3D world.

Scheduler

Each Wonderland has a Scheduler instance. The scheduler keeps lists of active Animations, Actions (e.g. BlockContexts that should be executed each frame), and Alarms. The scheduler causes time to pass in Wonderlands by iterating through these lists.

The scheduler updates itself as often as possible (using the Morphic **step** method). Each time the scheduler updates it first determines how much time has elapsed and what the current time in the Wonderland should be. The scheduler then processes the lists of active actions, alarms, and animations.

The scheduler executes any current actions and checks to see if the actions should be removed from the active list. The scheduler removes an action if the action's **for:** time has expired or its associated **until:** condition is true.

The scheduler next checks to see if any alarm times have passed. If the scheduler finds an alarm whose time has passed it executes the BlockContext associated with the alarm and then removes it from the list of current alarms.

The scheduler's last step is to update the active animations. Animations know their start state and time, end state and time, and the interpolation function to use to move between states. The scheduler thus merely needs to tell an animation the current time to cause the animation to update to the next intermediate state. The scheduler ends by removing any animation whose end time is earlier than the current time. Note that it does update these animations first to make sure that they actually reach their end state.

WonderlandActor

The WonderlandActor class is the most important Squeak Alice class. This class encapsulates the mesh and texture that comprise the visual representation of the actor, and defines the core behaviors that you use to interact with the actors. Internally the WonderlandActor classes uses 4x4 homogeneous matrices to represent the position, orientation, and size of objects. However, this internal representation is hidden from the user; part of the design philosophy for Alice is to present the user with an interface based on ease of use, not on implementation details.

The behaviors that the WonderlandActor class provides all work in a similar manner. If the user specifies that the behavior should occur **rightNow** then the effect of the behavior is instantaneous. Otherwise the behavior method actually creates an animation that encapsulates the current (start) state, the desired target (end) state, the duration, and the interpolation function to use (usually gently, or slow-in/slow-out). In either case the behavior method also creates an undo animation and pushes it on the WonderlandUndoStack instance for that Wonderland.

WonderlandCamera

The WonderlandCamera class defines a special type of WonderlandActor. In addition to possessing the same behaviors as actors, cameras know how to render a frame of the Wonderland from its current point of view. The camera creates an offset for drawing the scene based on its current position and orientation, and then tells the Wonderland to walk the scene graph. Walking the scene graph involves setting the background color and then telling the top-level actors (immediate children of the scene) to draw themselves. Each child actor draws itself (using its position, orientation, mesh, and texture) and then tells any of its children to draw themselves.

Each WonderlandCamera instance has a WonderlandCameraMorph instance that it actually renders into. You can access this morph using the **getCameraWindow** method on cameras. This morph uses the Morphic step method to re-render the view into the Wonderland as often as possible.

Wonderland

The Wonderland class is the container for the 3D world. This class contains lists of the cameras, lights, and actors in the world and provides methods for creating or loading these objects. The Wonderland class is also responsible for cleanly initializing a new Wonderland instance when you create it (e.g. creating a scheduler and undo stack), as well as cleaning up after a Wonderland instance when you delete it (by quitting the Wonderland).

The Future of Squeak Alice

Very little has changed with Squeak Alice since I completed my internship with Alan's group. While I believe that Squeak Alice has great potential, at least in the short term my time is focused on completing my Ph.D. However, when I next have time to turn my attention to Squeak Alice (or the next time another of the Alice team interns with Alan and his group) there should be many possibilities for its development. Andreas will have finished a new version of Balloon3D that is capable of taking advantage of hardware 3D acceleration, and the Alice team at CMU will have learned a whole new set of lessons about making 3D graphics easy for novices. In addition I hope that the next generation of Squeak Alice will be able to take advantage of feedback directly from its intended audience: novices to 3D graphics.

Further Reading

To learn more about the Alice project visit the Alice web page at http://www.alice.org. For more information on the lessons we learned from Alice, try these references:

Matthew J. Conway. *Alice: Interactive 3D Scripting for Novices*. Ph.D. dissertation, University of Virginia, May 1998.

Matthew Conway, Steve Audia, Tommy Burnette, Dennis Cosgrove, Kevin Christiansen, Rob Deline, Jim Durbin, Rich Gossweiler, Shuichi Koga, Chris Long, Beth Mallory, Steve Miale, Kristen Monkaitis, James

Patten, Jeff Pierce, Joe Shochet, David Staack, Brian Stearns, Richard Stoakley, Chris Sturgill, John Viega, Jeff White, George Williams, Randy Pausch. *Alice: Lessons Learned from Building a 3D System For Novices*. Proceedings of CHI 2000, pages 486-493.

About the author

Jeff Pierce is working on his Ph.D. in computer science at Carnegie Mellon University. During his Ph.D. student career he has worked on the Alice, consulted for Disney Imagineering on DisneyQuest, worked at Microsoft Research on handheld devices and 3D interaction, and implemented Squeak Alice for Alan Kay and his team. These days he is busily trying to complete his dissertation on novel 3D interaction techniques. Jeff can be reached at jpierce@cs.cmu.edu. For more information visit http://www.cs.cmu.edu/~jpierce.