Alice:
Easy-to-Learn 3D Scripting for Novices

A Dissertation

presented to

the Faculty of the School of Engineering and Applied Science

at the

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Matthew J Conway

December 1997

Alice: Easy-to-Learn 3D Scripting for Novices

© 1997 Matthew Conway All Rights Reserved

Approval Sheet

This dissertation is submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy (Computer Science)

____
Author

This dissertation has been read and approved by the Examining Committee:

____
Dissertation Advisor

____
Committee Chairman

____
Committee Member

____
Committee Member

____
Committee Member

____
Committee Member

Accepted for the School of Engineering and Applied Science:

____
Dean Richard W. Miksad
School of Engineering and Applied Science

Table Of Contents

Chapter 1 Acknowledgements 7
Chapter 2 The Three Stages of Media 10
Chapter 3 Executive Overview 20
Chapter 4 Alice System Overview 26
Chapter 5 System Description 51
Chapter 6 The Alice Main Loop 74
Chapter 7 Empirical Evidence 94
Chapter 8 Traditional APIs for 3D 112
Chapter 9 Objects, Parts and Children 124
Chapter 10 The Death of XYZ 131
Chapter 11 Rotation 156
Chapter 12 Sizing and Scaling 172
Chapter 13 Color, Texture and Visibility 183
Chapter 14 Programming Language Issues 194
Chapter 15 Future Work 206
Chapter 16 Previous Work 227
Chapter 17 Conclusions 234
Chapter 18 References 238

To Mom, Dad, Katie, Jessica and Paddy.

This report, by its very length, defends itself against the risk of being read.

Winston Churchill

It is easy to work when the soul is at play
Emily Dickinson

Acknowledgements

Alice would not exist but for the tireless efforts of every member of the University of Virginia User Interface Group. Without the UIgroup, this work and the larger Alice effort would have been impossible. Past and present members can be found at http://www.cs.cmu.edu/~stage3/ and through the *About* option on Alice's *Help* menu. I would also like to extend thanks:

To Randy Pausch, whose insight and guidance shaped this research from the beginning with a belief that fun is a critical component of any research venture.

To Tommy Burnette, Kevin Christiansen, and Dennis Cosgrove, for their infinite patience and brilliant insights in constructing the core of the PC/Windows 95 Alice implementation. They worked extremely hard to respond to the user testing that this work reflects and their insight in the design process was invaluable.

To Rob DeLine, for his Master's thesis on the first Alice implementations and his design of the Alice architecture. Rob's hand is still evident in Alice, a testimony to his

outstanding design skills.

To Rich Gossweiler, Shuichi Koga, Jim Durbin, Chris Long, Steve Miale, and George Williams for their design, implementation, development and maintenance of the original (now dormant) Silicon Graphics implementation of the Alice rendering engine, known in its day as Diver [Gossweiler]. Abandoning that code base for the PC world was a difficult decision, given Diver's outstanding stability and flexibility.

To Ken Hinckley and Jeff Pierce, Joe Shochet, and Brian Stearns for their help in the odious task of documentation writing.

To James "The Man" Patten for his work on the GUI tools he created for the Alice development environment, those that made it and those that didn't.

To Kristin Monkatis, Steve Audia, Chris Sturgil, David Staack for their phenomenal artistic input. Their skill brought a new burst of life at just the right time in the Alice development process. They helped remind me that what we were working on was in fact extremely cool. VR researchers forget the content side of the equation at their own peril, even if what they're doing is "just research."

To George Robertson and Microsoft Research for the patience and time they gave me in finishing this work.

To the Defense Advanced Research Projects Agency (DARPA) for funding this research.

To the members of the UVA CS department for their personal support and their generous support of the UIGroup, through the years.

To Jeni Willoughby and the teachers, administrators, and students in the Lynchburg Virginia School Systems for the courage and leadership it took to deploy the beta version of Alice in their high school.

And finally to Cassi Sendroff, age 11, and her sister, Eireann age 13, for the wonderful Alice worlds they created and for their perseverance with Alice, especially in the face of inadequate documentation. Their accomplishments with Alice were wonderful and surprising, and an unexpected validation of this work.

The content of any medium is always another medium
Marshall McLuhan

Chapter 1
The Three Stages of Media

The Alice project traces its roots back to early 1991, when the first Virtual Reality systems were being built at the University of Virginia. Starting with a desire to build new interaction techniques for immersive, head-tracked Virtual Reality (VR) simulations, we quickly found that the tools that were available for this work were extremely hard to use for exploring a new user interface design space. Realizing that this itself was a reasonable research agenda, we set out to build programming tools that were easier and faster to use, not just for ourselves but for as many people as we thought reasonably possible. This effort, though it started with several tentative prototypes under a variety of names[1], soon settled into what is now called the Alice project. We took as our charter the task of making interactive 3D graphics much more accessible to 19 year old "non-programmer"

1 Alice is the lastest system in a series of prototypes that we built starting in the early 1990s. Previous systems were named RPVR, Strobe, and Joe. We never had a system named Dave.

undergraduates, an audience that had been traditionally denied access to 3D graphics[1] because of the technical knowledge and mathematics needed to understand the tools. Why this audience? Why non-programmers? One way of understanding much of the Alice research agenda is to look to the past, to the invention of another technology-dependent visual medium: motion pictures.

For purposes of this discussion, what is interesting isn't so much the development of motion picture technology, but the development of the *content* over the first half-century of its history. I argue that film-making, as a form of expression, passed through three stages of development and that these stages are not unique to film, but coarsely describe the way that most new media pass from the laboratory to widespread use[2] [Eisenhart]. Interactive 3D graphics is in the process of passing through these same stages, and understanding these stages helps understand the Alice research agenda. The three stages of new media are :

1.1 Stage One: Novelty

In the earliest stage of a new medium, the developers of the supporting technologies are often the only ones in a position to explore what is and is not possible in the fledgling

---

1 Interactive three dimensional graphics is defined as any three dimensional computer graphics that can be rendered in real time at interactive frame rates (>12 fps). This definition therefore captures traditional "helmet and glove" VR as well as less exotic displays such as desktop monitors. This chapter applies to all forms of interactive 3D graphics, though some forms (e.g. head tracked genres of interactive 3D) may in fact have unique qualities that qualify it as a distinct medium of its own.

2 I arrived at this analysis of media evolution independently, but have since found corroboration in the more detailed treatments offered by Eisenhart and others.

medium. Because the medium is new, audiences are often forgiving if the technology pioneers don't say anything interesting with the new medium; the novelty itself is interesting. Until, of course, the novelty wears off.

Stage One Films [1]

In 1899, eight years after the patent was issued for the invention of the motion picture camera, came the age of the nickelodeon [Karney]. The most advanced "films" of the day were called *actualities* because they depicted extremely familiar scenes: men coming and going to work, men sneezing, pendulums swinging, and horses galloping were typical. Fortunes were made on these and on some of the more risqué titles, yet despite their popularity, nickelodeon[2] offerings were produced with single, fixed camera viewpoints, had no real stories, no plot, no discernable characters, indeed, they had virtually nothing that would remind us of the modern aspects of movie making.

Figure 1: Stage One Film. Several frames from *The Edison Kinetoscopic Record of a Sneeze* . Made on on January 9, 1894. This film was made by W. K. L. Dickson and depicts, Fred Ott, an Edison employee. This is the earliest surviving copyrighted motion picture.

1. An interesting collection of early American films, is held by the Library Of Congress on the web: http://lcweb2.loc.gov/ammem/papr/mpixhome.html
2. Also worth remembering that nickelodeons were single-viewer experiences, not the shared experiences that modern movie-goers enjoy.

Stage One Interactive 3D

Ivan Sutherland is credited with developing the first head-tracked display [Sutherland], a form of display device that would become iconic of Virtual Reality. Sutherland, like Edison before him, was a pioneer in a new medium, but Sutherland's efforts focussed on delivering the technology of the new medium, not on its content. Sutherland's first head-tracked display demonstrations presented a virtual cube hanging in space in front of the user[1], but unlike other computer-generated cubes of the day, this one seemed to occupy a position in space and allowed the user to get parallax information about its size and distance. This astounding display stands as one of the most dramatic examples of a Stage One effort – nothing more than a cube, but an important and pivotal first step toward more sophisticated content.

Figure 2: Stage One VR. Ivan Sutherland's colleague, Quint Foster wears the first HMD, circa 1967.

1.2 Stage Two: Emulation

Artists using the new medium to replicate the forms and content of previous media characterize this stage. Authors (or the more modern and distinctly more awful "content

---

[1] All fledgling VR systems seem to have their own favorite test object. We used a large block letter F, as it was asymmetric enough that it was useful as a debugging aid, and simple enough that its vertices could be generated by hand.

creators") do this either because they don't fully appreciate how to exploit the new capabilities of the new medium or they fear that the audience will not be able to understand how to interpret the new medium. Sometimes, the lack of creative expression can be blamed on technical or economic barriers; using more than one camera in early television was often very expensive, even after the problems of coordinating multiple cameras had been solved through on-the-fly direction. Sometimes coming up with the idea isn't good enough; the support mechanisms need to be in place and the economics need to work in your favor.

Stage Two Film: Filming Theatre

Film makers quickly saw that their medium could deliver dramatic material, but did this by emulating the most familiar form of dramatic presentation at the time: directors made films of stage plays with little or no camera motion.

Stage Two Interactive 3D : Animation

Early on, interactive 3D graphics was sometimes used to create virtually interactionless 3D movies displayed in a head-tracked display. Our own early VR efforts were very much of this style, owing mostly to the fact that building a VR simulation using the glove technology of the day (a Mattel Powerglove™ from the 8-bit Nintendo™ video game deck) was difficult and error-prone. We had fallen into the trap of building what was easy to build, not what was cutting-edge. To remain vigilant, we reminded ourselves of the dangers of new media with the tongue-in-cheek and probably over-conservative Conway's

First Rule of VR: *if it doesn't have a glove, it's not VR.* Since that whimsical pronouncement, the VR community has not shown that gloves are the best way to interact with a virtual environment, but the spirit of the law remains: interaction via direct co-spatial manipulation is an important part of what makes VR different from any of the media that preceded it.

Along these "stage 2 lines," I note a tendency that many VR research labs seem to have: given the opportunity to model any virtual space at all, the researchers often choose to model their own lab space.

Interactive 3D still borrows a great deal from motion pictures: both Disney's *Aladdin* VR attraction [Pausch96] and ID Software's *Quake* are highly influenced by film metaphors (indeed, Aladdin's backstory[1] is a full-length animated film), but neither of these 3D experiences is exactly like film[2].

1.3 Stage Three: Maturity[3]

A medium reaches stage three when expressions in the new medium exhibit unique behavior, allowing people to communicate in ways that previous media could never support.

---

1. Backstory: a storyline that sets up action that an audience is about to see, meant to frame the context, characters, and the rules governing the new story. An interactive experience like the Disney's Aladdin VR attraction gets its backstory from Disney's full length feature film of the same name. By putting the VR game in a familiar place with familiar characters, the player already knows a great deal about the virtual world with which he is expected to interact.
2. Some have argued that Quake is more like a Stage 2 example from the related medium of 2D/sprite based "scrollers" of the Sonic The Hedgehog variety. Indeed, Sonic is in some ways more complex than Quake because there is more to do in Sonic's world than just shoot-to-kill. Still, I believe the camera control and sense of spatial exploration in Quake and other first-person shooters of their genre, set it apart from the 2D scrollers.
3. Historical note: In mid 1996, we renamed our Virtual Reality studio/lab space "Stage 3" as a reminder to ourselves to strive for "stage 3" experiences in the interactive 3D experiences we were designing.

The new communication mechanisms may require that people develop skills that were previously unknown. Perhaps one test of whether a medium has reached stage three is the degree to which new technical roles get names of their own with their own traditions and craft: in film, as in stage we have writers and lighting directors but filmcraft adds to the list gaffers, best boys, and foley artists. In the design of interactive virtual worlds, the names of the necessary roles are still tentative, but it seems clear that a good interactive 3D experience requires the skills of programmers, modelers, painters, and often sound/musical talent.

Stage Three Film

D.W. Griffith's notorious but technically impressive *Birth of a Nation* (1915) was the first feature-length film, taking a year to produce compared to all previous films which were always short and hastily done [Karney]. Not only were films that followed *Birth of a Nation* held to this higher standard of length and craftsmanship, they were also held to a more sophisticated vocabulary. Griffith is credited with introducing film audiences to the cut, the close up, the pan, the crosscut, and action shots involving moving cameras. Note that it took nearly *two decades* after Edison's first patents before anyone conceived of these now "obvious" techniques. Reaching Stage Three is neither easy nor certain.

Stage Three Interactive 3D

I believe it is too soon to tell, but there are a few furtive steps taking interactive 3D graphics in the direction of new visual and interactive vocabularies. Eventually, the new

medium of interactive 3D graphics will have a rich and distinguishing vocabulary, one that is visual, kinetic and interactive. At first, this vocabulary will seem strange to new us, but in we will find them familiar and comfortable, just as the first film audiences became familiar with the visual techniques of cut and fade. The research is beginning to point toward what some of that interactive vocabulary might be like, in the form of novel 3D interaction techniques. Some notable examples: The *Worlds In Miniature* (WIM) work [Stoakley][Pausch95], Mark Mine's ISAAC system [Mine], the set of *Head Crusher* interaction techniques [Pierce] and Brown's *Sketch* modeler [Zeleznik]. Each of these research prototypes allow users to interact with 3D spatial data in novel ways that have no real analogue to previous forms of expression, not even film. These are only first steps, of course. The truly new forms of expression in interactive 3D graphics are waiting to be invented.

1.4 Passing the Torch

The innovators in the content of the media were rarely the technologists who created the machinery that made film possible. Georges Méliès, a stage magician, toyed with his cameras and learned that by cranking the film backwards and shooting new footage over the old, he could get a superimposed, ghostlike effect. In doing so, Méliès invented the art of cinematic special effects.

Movies developed into their modern form in part because the technology of movie making was put into the hands of people who were more interested in expressing an idea than in the technology for its own sake. If, instead, cameras and projectors remained hard to

use and expensive to operate, they would have likely remained in the hands of engineers who all too often, remain stuck in Stage Two.

This research is driven by the belief that 3D interactive graphics should be brought into Stage 3, and that this cannot happen unless the technology is made easier and more accessible to people outside the current community of programmers[1]. Current barriers to the technology are simply too high; requiring mathematics skills that are usually seen late in a science, math or engineering course of study. As long as matrix algebra, vector mathematics and C/C++ programming skill remain the price of admission for creating content in interactive 3D graphics, two things will remain true:

Homogeneity of Expression: The people who create content will remain a self-selected group of a relatively homogeneous set of skills, interests, experiences and backgrounds. This will mean a narrower variety of subject matter for 3D interactive programs.

Poor Focus: The majority of the time and effort required to create content will remain inappropriately focussed on implementation concerns ("how do I drive the tool?") and away from the presentation itself ("what is the message").

---

1 Some of the animators responsible for creating the dinosaurs in *Jurassic Park* [Knep] claimed that they felt that for the first time, artists were driving the content of the film, not the technologists, largely due to the presence of specialized input devices used to aid in the creation of the stop motion animation. *Interactive* 3D has yet to make this sort of advance in ease-of-expression.

The research results presented here show that ease-of-learning of a 3D API can allow more people to create content in the medium of interactive 3D graphics than was previously possible. I will also argue that many of the design decisions that make this possible have a positive influence on the performance of experts as well, underscoring my belief that the tradeoff between power and simplicity is often a false one. Experts and novices alike will need to have more effective authoring environments than the ones that currently exist if interactive 3D graphics as a medium is to achieve Stage Three.

I don't have any solution but I certainly admire the problem.
Ashleigh Brilliant

Chapter 2
Executive Overview

2.1 Contributions of this Work

The central thesis of this work is that 3D interactive graphics programming does not need to be as hard to learn as it is today. It seems clear to many observers in the field that in order to control the behavior of 3D graphical objects, a person needs to possess sophisticated mathematical skills (e.g. vector and matrix algebra, trigonometry) and that this requirement represents an impenetrable barrier to all but the most technically inclined people. My work represents a partial solution to this problem in the form of a prototype programming system named Alice, which makes 3D graphics programming easier than other commonly available tools. Alice's ease-of-use claim can be substantiated by showing that a new class of people, 19 year old non-engineering undergraduates with little or no programming experience, can effectively use Alice to create interactive 3D graphics scripts. This is a class of user who is clearly outside the reach of the presently available tools. The

Alice work shows that the need for advanced mathematical skill is not inherent to the domain of 3D graphics, but is simply a reflection that existing tools are not as simple as they could be or should be.

My work shows that the Application Programmer's Interfaces (APIs) can be improved by observing pairs of people constructing computer programs using the API. This two-person talk aloud protocol is a well-known technique for the evaluation of Graphical User Interfaces (GUIs) but this work shows that the technique has broader applicability.

Finally, my work shows that existing APIs depend too heavily on exposing the underlying representation of the graphics state (typically a 4x4 homogeneous matrix) or of the hardware (typically the polygon pipeline). Alice demonstrates that 3D APIs can effectively hide the implementation details behind higher-level, more powerful abstractions without sacrificing power. Some of Alice's most powerful abstractions were inspired, in part, by the research results of other fields, notably spatial understanding (psychology) and linguistics. This cross-disciplinary strategy of looking far afield for inspiration was a valuable one, and I believe serves as an example to researchers who will follow.

2.2 What Alice is Not

Alice is not a modeling package. Creating 3D worlds clearly requires the presence of 3D models to manipulate but both modeling and manipulation were each seen to be sufficiently difficult research thrusts. Research efforts continue elsewhere to make 3D

model-making simpler [Zeleznik], while the Alice effort focuses exclusively on the problem of behavior specification through programmatic control.

Alice is also not a keyframe animation system. Alice scripts are interactive, which keyframe animation systems typically are not. Neither is Alice a multimedia system, though there is some limited support for "bitmap" movies and AVI files: bitmapped images that can be texture mapped onto 3D polygonal shapes in the 3D scene and some support for playing back and looping audio files in the popular WAV sound file format.

2.3 The Research Strategy

The research that has gone into exploring the above issues has been based on a two-pronged strategy:

Usability Engineering: Gather data from members of the target audience by directly observing them interact with the system and through the use of exit interviews. This will uncover mistakes and design errors in the Alice implementation and will effectively drive the Alice system away from inadvertent design mistakes.

Spatial Understanding Literature: I have harvested research from the psychology community that describes the abstractions people use to describe 3D scenes, and the techniques people use in 3D wayfinding tasks and navigation. There are times when I use these abstractions directly in the API, and there are other times when this literature has been more inspirational than directly prescriptive. The hope is that a programming environment

and an API designed in this way will be easier to use and to remember because the API maps well to knowledge that the user already has about spatial tasks.

2.4 A Note About Evaluation and Design Improvement

The evaluation of Alice is one that explicitly avoids the pitfalls of testing one system directly against another in a "shootout" style confrontation. Instead, we choose to evaluate Alice against "whole results," meaning that it is better to test Alice against a certain standard of performance than against some other software artifact. In short, Alice is successful if it can be understood and used by a significant number of 19 year old undergraduate non-programmers, a level of human performance that is considered to be a significant improvement over the current state of affairs.

This mechanism avoids the question of generality: testing Alice against other systems would only reveal whether Alice was better than a rival for that particular task. It is exceedingly hard to generalize in any useful way from "point design" involving specific tasks. Instead, we choose to test Alice, and thus drive its design, using the same mechanisms that drove the design of the Apple Macintosh.

2.5 Outline of This Work

The following is an outline for the remainder of this work.

2.6 System Software Description

This section gives a broad overview of the Alice architecture, the software components that make Alice work, and a brief description of the Alice object interface. This isn't meant to be the definitive reference work for the Alice maintainers, but only enough background material as needed to understand the underlying API design ideas.

2.7 Empirical Evidence

This section describes the data gathering mechanisms used and the procedures employed in testing of Alice and Alice's support documentation. Alice is very much an empirical work with observations from 104 people forming the database from which my conclusions are drawn.

2.8 APIs For 3D Programming

This section examines several commands provided by traditional 3D APIs and contrasts them with the Alice versions. I also show several high-level Alice commands and API concepts that are usually not seen in traditional 3D APIs.

2.9 On Scripting vs. Direct Manipulation:

Programming 3D graphics animations often suggests a visual or direct manipulation interface. Alice's interface is very strongly geared away from this paradigm, and instead relies heavily on a lexical or script-based model for animation control. This section discusses the

strengths and weaknesses of both direct manipulation and imperative script to control the unfolding of animations and interactive applications.

2.10 User Observation for API Testing

Programmers APIs can be improved by the same mechanisms that application GUIs have been improved: through observation of real users engaged in the construction of real programs.

2.11 Spatial Understanding Literature

This section discusses a survey of the spatial understanding psychology literature, and the implications it had for Alice system design, leading to a discussion of where this could lead Alice system design in the future.

2.12 Previous Work

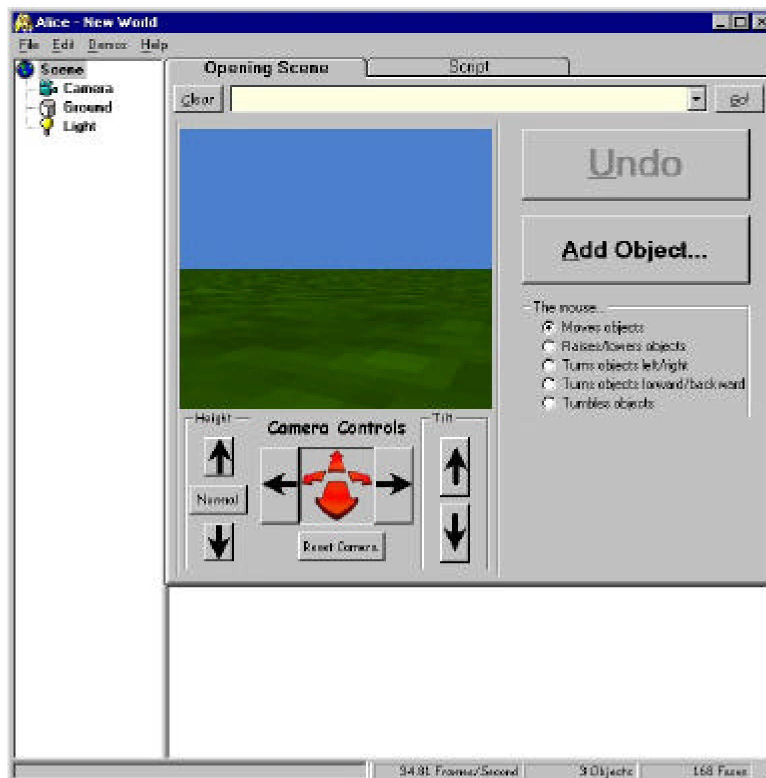
This section will compare Alice to related research and commercial efforts. I discuss previous works at the end of this document so that the differences and similarities of previous works can more effectively compared to Alice.

2.13 Conclusions

2.14 Future Directions

Chapter 3
Alice System Overview

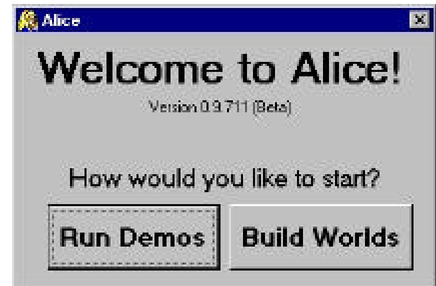
3.1 The Alice GUI



This extended example is meant to show what an average Alice interaction is like to our end user. Of course, a static presentation of Alice can't capture everything important about the Alice experience, but it can help make the explanations that follow more concrete. The following example shows

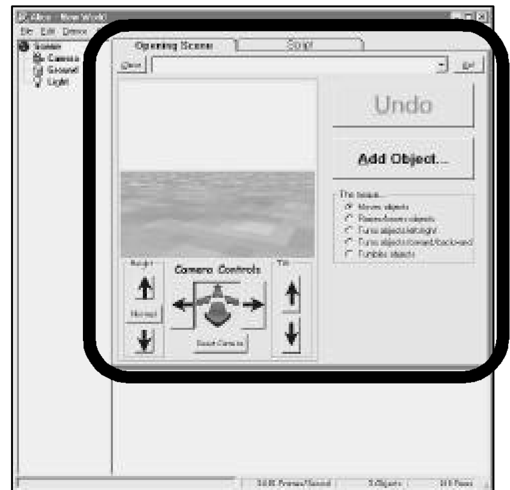
how a user might develop code for making a bunny hop using Alice version 0.9.771 Beta.

Step 1 : Starting Up



After we start Alice, we see a dialog box offering a choice between running demos or building worlds, and in this example, we choose to build a world. This panel exists purely for "marketing" reasons, so that users who are unfamiliar with Alice can quickly get an overview of the sorts of things that Alice is good at. Live demo programs give a fast and compelling answer to the question "why would I want to use Alice?" We learned the importance of motivating new users through live demonstrations during the SUIT project [Pausch] in the late 1980s, which also had ease-of-learning as one of its central goals.

This brings up the main Alice control panel. This control panel is a tabbed dialog box with two tabs, the first of which is the Opening Scene Panel.



Step 2: The Opening Scene Panel

Before the programmer can begin writing code that controls the behavior of 3D objects, he or she needs to set the stage by populating the world with objects, placing them in their initial locations, and pointing the cameras and lights in an

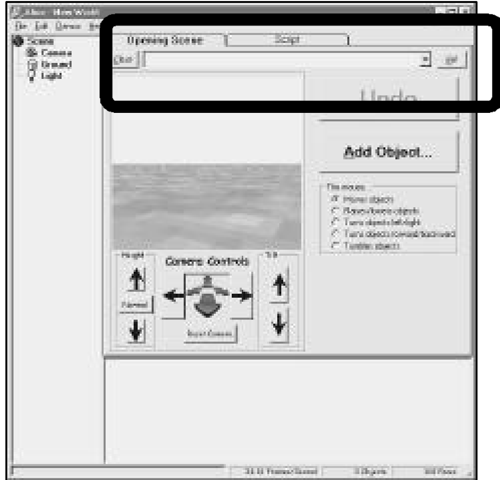
appropriate way. This set of initial conditions is called an Opening Scene, and is constructed through the Opening Scene Panel. The most prominent feature of the Opening Scene panel is the Alice rendering window which shows the 3D scene from the point of view of a synthetic camera pointing into a 3D virtual world. By default, the Opening Scene starts with one light, one camera, and a textured ground plane.

Step 3: Adding Objects

To begin populating the world we press the **Add Object...** button, which displays the Alice 3D-object gallery. In this example, we choose to add a bunny.

Once the bunny is in the world, we can move it with the mouse, picking directly into the rendering window, dragging and rotating the bunny into its starting position. In our example, we are prototyping a new hopping behavior, so it doesn't really matter where the bunny starts, as long as he's visible. Therefore, we can establish our opening shot by using the mouse to drive the camera in close to the bunny, as shown in the picture. A more complex opening shot would include more objects, possibly even more cameras and more lights. Having established our (very simple) opening shot, we can move on to experimenting with fragments of Alice script that we will use in our hopping behavior.

STEP 4 : Using the Alice Command Box



The key to inventing a new behavior is the ability to play around with ideas without having to endure long, enforced pauses before seeing the results of our experimentation.

Alice programmers do this through the Alice Command Box, a one-line typing area that allows users to evaluate single lines of Alice script at runtime.[1]

If, for example, we wanted to move the bunny by a precise distance (as opposed to using the mouse), we could click into the command box and type a single line of script:

```
Bunny.move (up, 1)
```

By pressing the GO button (or Enter) we can make the bunny move upwards (up, relative to the bunny) one meter, taking one second for the movement. All commands in Alice are animated by default whenever it is semantically reasonable.

If at any time we do something that we would like to reverse, we can press the Undo

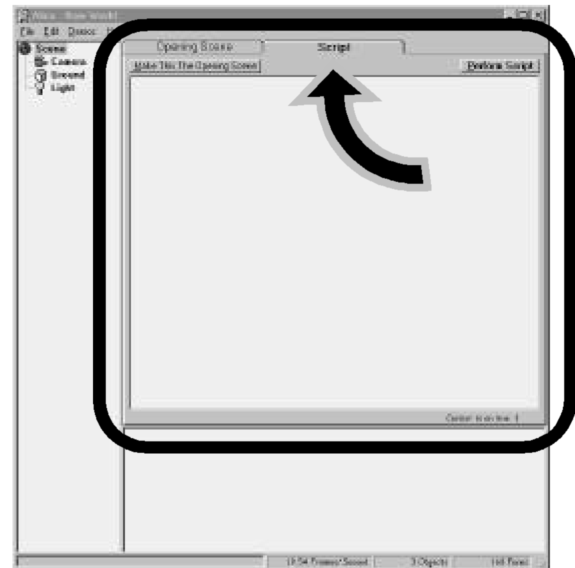
1 Alice script is a slightly modified version of an interpreted language called Python, and is the means by which end programmers specify the programmatic behavior of objects in the scene. Python and the modifications we made to the language are described in greater detail in Section 4.6., page 64.

button, which engages an infinite-level undo mechanism capable of undoing nearly any operation in Alice all the way back to the beginning of an Alice programming session. The Alice Undo mechanism is discussed in greater detail in section 3.8.

After some experimentation, we are ready to create an Alice script for this world.

Step 4: Creating a Script

Now we go to the second tab of the main interface, the Script Panel where we will write several Alice commands in a row and have the Alice system apply them to the scene in batch.



At present, the script panel is a just simple Wordpad[1]-like text editor, but we have developed advanced prototypes that provide support for context-sensitive coloring (command tokens in red, comments in green, strings in blue) of the Alice script which we hope will improve the readability of the scripts.

1. Wordpad is a very simple text editor that ships with Windows 95.

After some reflection, an important part of creating a behavior for Alice objects, or any programming effort for that matter, we conclude that a hop is made up of the following steps:

1. recoil the bunny scrunches down
2. leap the bunny rises into the air while unscrunching
3. fall the bunny comes back to the ground
4. bounce the bunny scrunches down, then rebounds to normal shape

To do this, we write the follow lines of Alice script:

```
doinorder(
            # recoil
            bunny.resize(toptobottom, 0.5, likerubber),

            # leap
            dotogether(
                  bunny.resize(toptobottom, 2, likerubber),
                  bunny.move(up, 1)
            ),

            # fall
            bunny.move(down, 1),

            # bounce
            bunny.resize(toptobottom, 0.5, likerubber),
            bunny.resize(toptobottom, 2, likerubber)
)
```

And then we run the script. This causes several things to happen: Alice resets the world to the same state as the opening shot (resetting the state of the simulation, which may

have been altered from earlier runs of the script or from mouse gestures). Alice then saves the script to disk (into a world file, which is just a combination of the opening shot and the script), and finally, Alice executes our script. If we wish to make a change to our script after seeing the script run, we can make alterations and then run the script again (which again resets the world to the state of the opening shot before executing). Once we are satisfied with the results, we can save our work in a world file and exit Alice.

3.2 Design Rationale: Why Shots and Scripts?

The creation of a 3D interactive simulation can be broken down roughly into the following phases of development:

1. The initial placement of three-dimensional objects for the beginning of the simulation.
2. The placement of some number of cameras in the scene (Alice provides one by default, so this step is usually skipped.)
3. The definition of the behaviors of the three dimensional objects.
4. The definition of how behaviors are changed in the face of user interactions (mouse and keyboard commands) or as a function of time.

The earliest versions of Alice made no distinction between these steps; all these tasks were accomplished through function calls executed from the end-programmer's Python script. For example, a simulation script in an early version of Alice might start out like so:

```
bunny = MakeAnObject("animals/bunny")
bunny.MoveTo (3, 0.232, 2)
bunny.TurnTo (232, -2, 55)
camera.MoveTo (0, 3, .2)
camera.TurnTo (0, 90, 0)
bunny.Move (Forward, Speed=2)
```

Notice how the script is responsible for everything: making objects, placing them in the scene for their initial locations and orientations, and making the simulation start. The numbers in the script come from experimentation; the programmer moves objects around either with the mouse or through evaluating code in the command box, noting positions and orientations by printing them out occasionally. When he or she decides that the opening scene's configuration is acceptable, the programmer can use the AnObject method GetPosition() and GetOrientaton() to query the state of the objects, thus generating the "magic numbers" we see above. While it seems obvious now that this workflow is too complex, too tedious and too error-prone to inflict on novices, there was a time when this workflow was a huge step forward from the compiled language 3D-programming mechanisms of the day. The ability to simply move objects interactively and evaluate code on the fly was such an improvement over the edit-compile-repeat paradigm of the day that the nuisance of having to generate the starting positions numbers "by hand" was one we were willing to endure for a long while. Of course, our novice users, not conditioned by the unpleasantness of previous tools, found this state of affairs still unacceptable.

The solution to this problem came from the observation that Steps 1 and 2 (from the

list on page 32) are setup steps often done more easily through direct manipulation while steps 3, 4, and 5 are definition steps, which are more accessible through textual/scripting means[1]. I made this partition in the workflow explicit by introducing the notion of an opening scene and a script.

3.3 A More Detailed Look at The Alice GUI

The sections that follow go into more detail and discuss some of the more subtle issues behind the scenes of the typical Alice session we just saw.

Inserting Objects

At the point when the user selects an object for inclusion in the 3D scene, Alice does several things:

1. The specification of behavior, of course, could also be done with direct manipulation among other techniques. I discuss the advantages and disadvantages of alternate paradigms starting on page 116.

1. **Loads polygons** into the graphics database from the geometric data stored in the file. Alice reads 3D Studio, OBJ, and DXF file formats. When storing new objects, Alice employs a proprietary format called A3D.

2. **Moves the object** so that it sits on the ground. This comes from the observation that novice users are very confused by seeing objects appear in their worlds floating in air. Previous research has shown that the presence of a textured ground plane provides a very strong depth cue [Dunn][Bajcsy].

3. **Creates a Python object** which is strictly speaking an instance of class AnObject. Alice then creates instances of all the parts of that object.

4. **Marks the objects** : the top-level node for the object is marked as being a "first class" object. and the Python instances for the parts are left unmarked, denoting that they are "parts."

5. **Gives the object a name** so that the object can be manipulated through Python. Object names are derived from the name of the file that stored the object, so in this example, choosing a file called bunny results in the object being named "bunny." See the illustration on page 36. The name of the object is available as a variable that can be manipulated in the Python script later. If the user creates more than one instance from the same A3D file, these objects have a number appended to their names: bunny2, bunny3, and so on. Alice used to prompt the user explicitly for a name, but this caused confusion in novice programmers, and so this feature was turned into an option that expert users can enable as desired.

6. **Names the object's parts.** Python names for parts are derived from the names of the parts stored in the file. Python objects are allowed to have "dotted-attributes" which are accessed with simple OOP syntax : bunny, bunny.head, bunny.head.left_ear, etc. Each of the attributes is an instance in its own right and is therefore allowed to have dotted attributes of its own. In this way, the assignment of Python names to object parts is performed recursively over all parts.

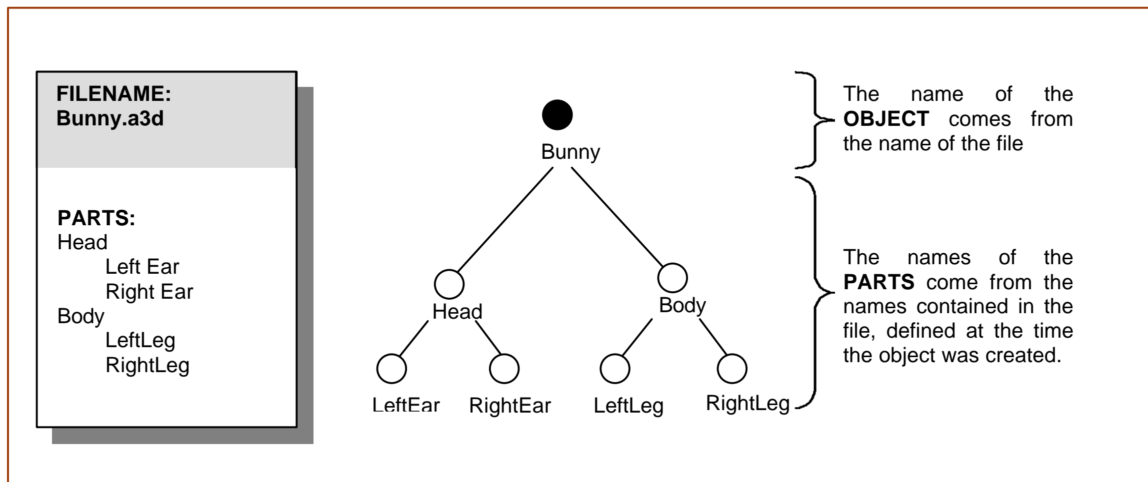
Alice does not allow for the creation of geometry interactively but instead reads several popular 3D file formats. Not being able to make one's own objects may seem like a strange lapse of functionality for a 3D authoring environment, but Alice was never intended to solve the entire interactive 3D authoring problem, only the problem of behavior specification. Simplifying the creation of 3D models was deemed to be beyond the scope of this research and so we depend on 3D models that are created by skilled artists using

standard (and unfortunately hard to use) 3D CAD programs.

Alice's file format for 3D data contains a polygonal description of the object, the colors and textures applied to those objects, plus hierarchy information, if any, that specifies the parts and sub parts of an object.

Names of Objects and Parts

The person building the 3D model assigns a string name to each part of the object. These part names are particularly important because Alice will use them to generate attributes at runtime for the objects that are added to the scene.[1] For example, when our bunny object was modeled, the parts of the bunny were assigned names by the creator of the



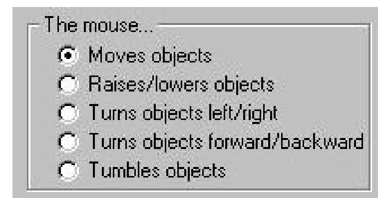
1. The use of the word *object* in this context is potentially confusing: Alice objects have a 3D graphical representation and are accessed by Alice programmers through Python objects in the "object oriented programming" sense of the word. These Python objects can have attributes associated with them that are accessed through the fairly standard OOP notation of setting the attribute off from the object with a period. In the context of this discussion, bunny is the object and head is the attribute.

model. That person chose to name the bunny parts head, body and drum. When we added the bunny to the scene, Alice created an object called bunny (this name derived from the name of the object file) and then created objects for each of the parts of the bunny. These parts are given names found in the file and are *bound to the bunny object as attributes* of the Python object called bunny. For example, the head of the bunny is accessed through the name bunny.head, while the head's parts are accessed through the names bunny.head.left_ear, and bunny.head.right_ear. Adding an object to an Alice scene not only creates a new graphical object, but also populates the namespace of the Alice programming environment with all the handles needed to manipulate each of the independent parts of the object being added to the scene.

3.4 Using the Mouse

The mouse can be used to move and rotate objects, to drive the synthetic camera around in the 3D scene and to invoke context menus over the 3D objects. This overloading of functionality through a mouse with only two buttons poses interface challenges that we address in the following manner:

Moving Objects With the Mouse

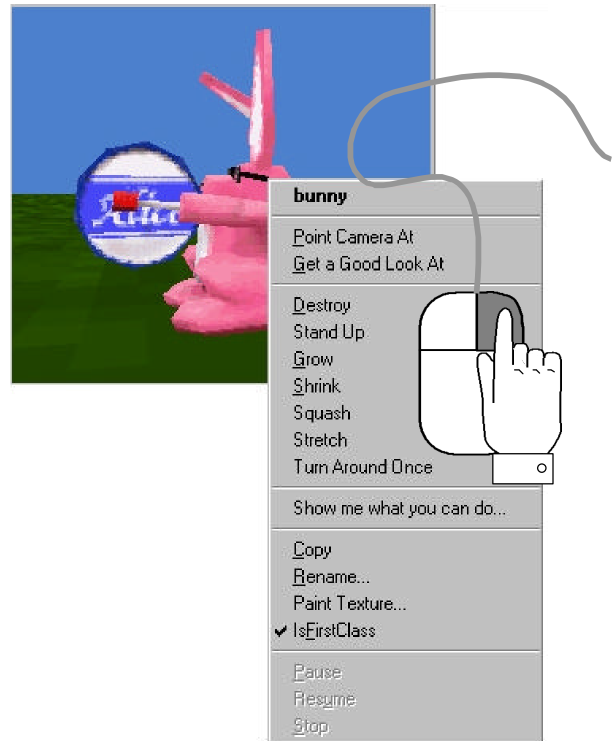


While the default behavior of the mouse is to move objects in a plane parallel to the ground, there are other operations that users can also perform through direct manipulation. Rather than only

providing the extra modes behind modifier keys (drag=move, shift-drag=rotate, etc), we decided to also make them explicit by a set of radio buttons. While this moded interface is far from fluid, it has the advantage of being extremely simple and self-revealing.

The Mouse Context Menu

Objects in the scene can also be manipulated by holding down the right hand mouse button. This action is analogous to other operations in Windows 95, and brings up a context menu for the given object. The items on this menu are some of the more common operations that a user might want to perform, including:



- point the camera at the object (*Point Camera At*)
- move the camera to a position that is above and off to one side of the object (*Get a Good Look At*)
- spin the object in place for simple examination (*Turn Around Once*)
- Show a list of methods that this object responds to *(Show Me What You Can Do).*

The implementation of this last command is a hand-coded scrolling list of useful commands, with full text examples of each. Future versions should have a similar mechanism that can grow as the repertoire of commands available to an object grows. The

context menu also provides functionality for controlling the animation of the object. If the object is static, as is the one in this picture, the options for stopping and starting the object (Pause, Resume, Stop) are disabled.

3.5 Moving the Camera with the Mouse

Below the rendering window is a set of camera controls for driving the viewpoint through the scene in a simplified “architectural walkthrough” style (i.e. no camera controls for “rolling” the camera, all camera motion is planar.) There are clearly other camera-motion paradigms including:

Flight simulator controls including roll, pitch and yaw. Our target audience is likely to include a fair number of game players, which argues for inclusion of these controls for those game/application domains that demand such full control. Note that a great many environments do not require this complex style of navigation, which is why the Alice implementation does not supply it as a default.

DOOM or Quake-like controls from popular first-person, interactive 3D games. This set of keyboard and mouse interactions will let us take advantage of a highly-refined human skill set that has been cultivated by many members of our target audience and which allows people who have this skill to navigate very fluidly and efficiently in architectural-walkthrough type environments. Alice currently has a working prototype of the DOOM/Quake navigator tool, but it has not been tested or shipped with the beta software.

Click-to-fly allows users to click on objects or other “Points of Interest” and fly ballistically to them in constant time along an interpolation curve, as per [Mackinlay].

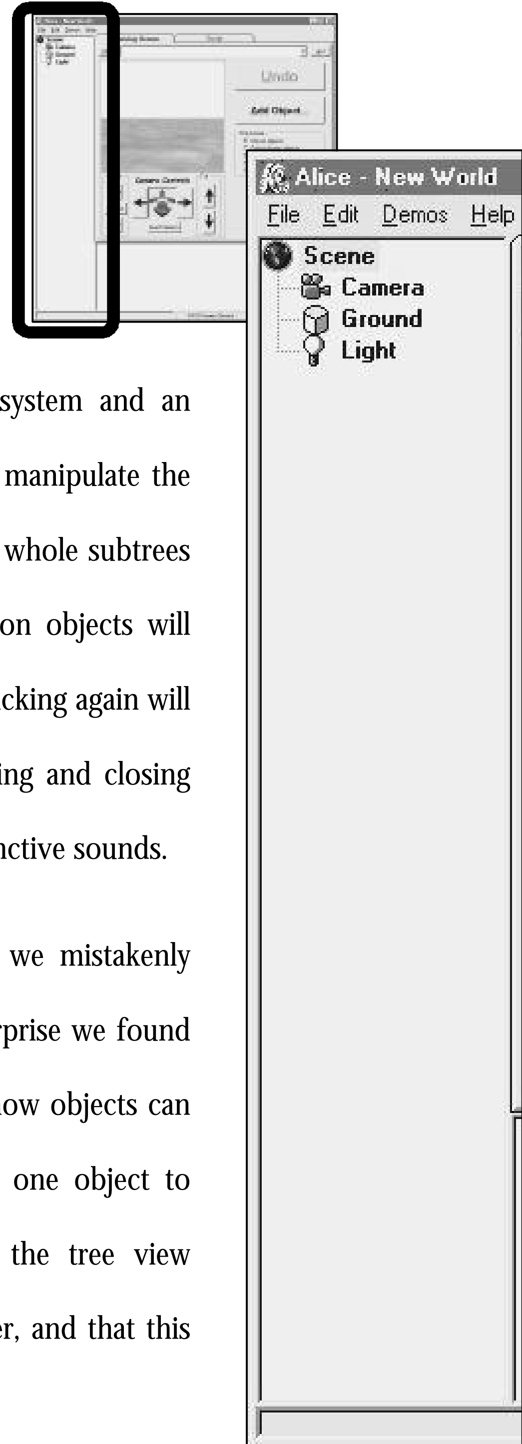
Examine object allows users to click on an object, which points the camera at the

selected object and then modes the mouse so that subsequent mouse motion is converted into camera rotations around the center of the selected object.

Fly-to-named-landmark is similar to click-to-fly, but uses a textual list of names as the selected destination instead of the objects themselves. VRML 2.0 worlds often feature a vehicle with this navigation feature. The list of named destinations is prepared by the designer of that world and allows users to travel along complex (pre-authored) paths, freeing the user from needing to locate them along a free line of sight, and from having to manage the navigation by hand.

Each of these modes is a legitimate navigation style, and warrants its own set of APIs and GUIs – the one that an application writer chooses is entirely dictated by the nature and demands of the 3D task at hand. There is a more exhaustive taxonomy of virtual environment navigation styles given in [Bowman], which may also serve as a useful source of API navigation packages.

3.6 Restructuring Objects with the Mouse



The vertical column on the left of the Alice panel is a tree view display of the Alice hierarchy of objects in the scene. Alice scenes are organized in a traditional PHIGS-like tree structure [VanDam] where each object has a unique parent which acts as a local coordinate system and an unlimited number of child nodes. We can directly manipulate the nodes in this tree display with the mouse, dragging whole subtrees of objects from one parent to another. Clicking on objects will “open them” revealing the children of the object, clicking again will collapse the tree, hiding the children. These opening and closing operations are each accompanied by their own distinctive sounds.

The object tree was a GUI element that we mistakenly predicted would cause much confusion. To our surprise we found that our users had no problems in understanding how objects can have sub-objects that can be “re-parented” from one object to another. Several users mentioned explicitly that the tree view looked something like the Windows 95 file explorer, and that this made object hierarchies easy to think about.

The tree view is not just a "parent/child" visualization tool, it is often the only way to conveniently point to an object in the scene. Objects in a 3D scene can be occluded by other objects, can be behind the camera view point, or can be too distant to be picked with the mouse. Given this, we have made sure that the right-mouse button menu (page 38) also works when the user right-clicks over an object in the tree view.

3.7 The Command Line and an Animated API

All API calls in Alice launch animations of one-second duration whenever it is feasible to do so. Very early on, we were skeptical of the power of this technique, and believed that expert users (such as the Alice development team) would require a toggle switch that turned off the automatic animation of commands in the Alice command box. We were very wrong. Without animation, operations can be very difficult to understand, especially when a command that moves an object or moves a camera causes things of interest to move completely off-screen, out of the view of the camera. At least when commands animate, users can see what direction the errant object(s) went, with some hope for correction. Teleportation is very confusing, even to experts.

Most commands can be animated, but there are some commands that by their very nature are not animatable (see sidebar). Take for example the BecomePartOf command that moves an object in the tree from one parent to another. There seem to be no animation semantics for this operation, and so this operation executes instantly, with a duration of one frame time, not the default of one second. Most of these non-animatable commands (see

sidebar) fall into one of several categories:

- commands that are logical in nature, not spatial
  (e.g. RespondTo)
- commands that require transaction semantics
  (e.g. GetPosition)
- commands that have a discrete value for which multiple-frame interpolation has no meaning
  (e.g. changing rendering styles from WireFrame to Gouraud shading)

Just because it appears at first that a given command cannot be animated, doesn't mean that we can't invent a way to make it *appear to be* animated. For example, it took the Alice design team many months to realize that even though the Delete method was implemented as a single-frame, transaction-like operation, we could display the deletion to the user as if it were an animation. Thus, deleting an object in Alice causes its parts to fly off into space, rolling and tumbling, fading from opaque to transparent, taking one second from

Non-Animated Alice Commands

DistanceTo, GetPosition (*or any call that queries an object's state. Generally, any call that starts with the word Is or Get.)*
BecomeParentOf
BecomeChildOf
BecomePartOf
BecomeFirstClass
BecomePart
SetChildren
MakeTransparentToInput
AttachCamera
SetVertices
DoToEach
Start
Stop
Pause
Loop
Wait
Do
PlaySound
RespondTo
AddResponse
RemoveResponse
Resume
SetName

Commands That Could Be Animated But Are Not (in Beta)
ShowBoundingBox
CastShadow
StopCastingShadow
SetTexture
SetTransparentTextureColor
Show
Hide
SetFillStyle (points, lines, filled)

beginning to end[1]. The actual delete operation removes the object from the rendering and simulation databases at the end of this short animation. On those occasions where the deletion is supposed to be appear instantaneous as well, the programmer can turn off the animation explicitly by supplying a keyword parameter to the function call:

```
Obj.Destroy(Duration=0)
```

3.8 Issues in Supporting Undo in an Animated Environment

Alice's Undo mechanism is animated, meaning that undoing an operation does not instantaneously restore an object to its previous state, but animates the object back to its original state, taking a constant one second to do so. This isn't just an implementation trick meant to make Alice more appealing; it is an explicit design decision meant to exploit the human perceptual system and our need for smooth motion to maintain object constancy [Robertson].

Users do not seem to be confused by the fact that an operation that takes several seconds to execute in the "forward" direction will only take one second to execute in reverse under the undo command. Perhaps users are forgiving as a matter of pragmatism, not wanting to waste time to see their "mistakes" unfold in reverse, but it is interesting to note that the temporal qualities of the animation are not coded in the user's cognitive

1 See section 6.9 for user reactions to this feature.

understanding of the operations that are being undone. I have never observed a user being confused by the fact that undo executes in reverse far faster (usually) than the running times of the original animations that are being undone.

Currently, an undo animation must run to completion before another undo animation is initiated. This lockout happens to prevent undo animations from interfering with one another.

Undo interacts in interesting ways with steady-state animations like:

```
obj.turn (Left, Speed=2)
```

This starts the object turning to its own left at a speed of two revolutions per second. The animation causes the object to spin forever until either the user or the script causes it to stop. If the user presses the Undo button while the object is spinning, the object will (1) stop spinning and (2) will rotate back into the state that it was in before the steady-state animation was launched.

3.9 Infinite, Animated Undo

Pressing the Undo key in Alice engages a sophisticated undo mechanism that is:

Infinite in extent meaning that the user can execute all previous actions in reverse, all the way back to the beginning of the session, within the limits of virtual memory.

Animated, implying that the operations being undone are not applied to the scene

instantly, they are presented to the user as smooth transitions, animated over an interval of one second, regardless of the duration of the original operation.

To see how this works, it is helpful to see an example. Suppose we would like to rotate an object 1 / 4 turn (90 degrees):

```
obj.Turn (Right, 1 / 4)
```

Before executing this command, Alice queries the state of the object that is being manipulated and uses that state to push a triple (object, inverse operation, state) onto an undo stack. In this case, the implementation does this:

```
old_orientation  = obj.GetOrientation()
PushOntoUndoStack( obj,  TurnTo, old_orientation)
```

When it comes time to undo the operation, the first triple on the undo stack is pulled off, assembled into a command, and then executed which in this case, calls TurnTo on the object obj, sending it back to its old_orientation.

Another way of thinking about undo is in terms of states. If an Alice command takes an object from some starting state S_{old} to a new state S_{new}, then an undo operation is one that takes the object from S_{new} back to S_{old}. Animated undo does the same thing, but takes S_{new} to S_{old} through a series of intermediate states $\{S_1 \ldots S_N\}$ for an N frame animation (always one second in duration, so the number of frames N will depend on the frame rate).

Undo and State Sequences

It is interesting to note that the states $\{S_1 \ldots S_N\}$ displayed during the undo animation **may not be exactly the same states** that the object passed through during the forward execution of the animation. This can happen for example when the object is being turned – in the case of a 359 degree rotation, it leaves the object only one degree out of alignment from its starting position. The undo operation that restores the object to its old orientation will take the short way around and will therefore only rotate the object one degree, not 359 degrees in the opposite direction.

Users notice this discrepancy and have often complained to us that this seems to be a bug, and even though the end state is correct, it takes time and effort to realize that this is so. I note that some operations do not show this effect: animating a SetColor and its undo operation through different paths is something that users seem to be less sensitive to.

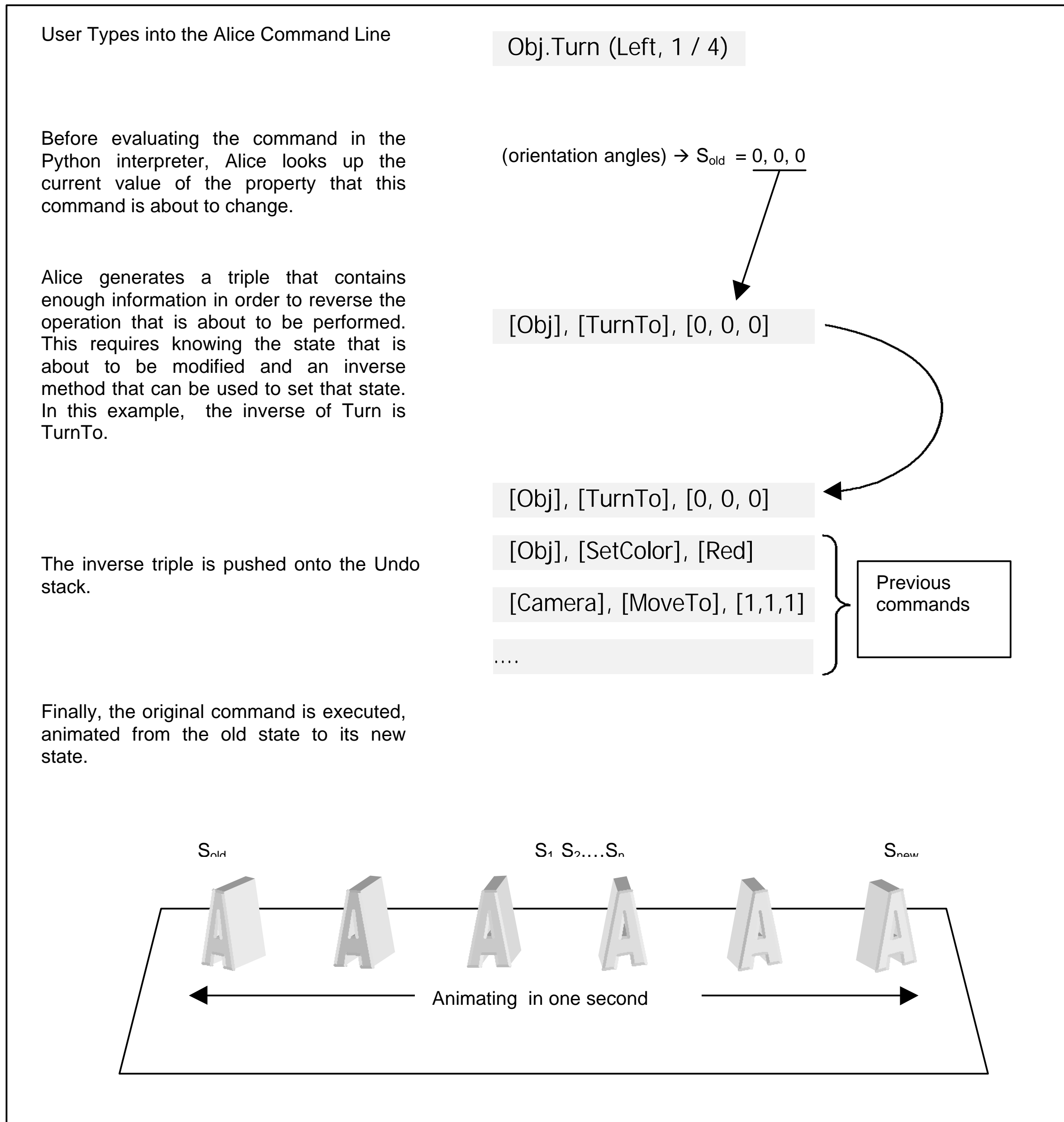
Invisible Undo

In the limit, not encoding a path through state-space leads to undo operations that have no visible effect on the screen. For example, if we were to rotate an object by 360 degrees, we would find that the old orientation state pushed onto the undo stack was exactly the same as the final state of the object: $S_{old} = S_{new}$. This means that when the user presses undo, the object does not rotate. Users notice this and are convinced that it is a bug ("Alice " , "Why doesn't it turn backwards?"). In the case of orientation changes (rotation), it seems that it isn't the state of the objects that users are concerned with, so

much as they are concerned with the path that objects take during undo operations. The animation is just as important to users as is the final state.

As a matter of future work, the Alice undo mechanism should encode a path through state space, and not just end state, so that we can animate undo operations through a series of states $\{S_{new} \ldots S_{old}\}$ that more closely matches the series of states that brought the object to the state S_{new} in the first place.

3.10 Quick Overview of Undo



Our life is frittered away with detail…Simplify, simplify
Henry David Thoreau

Chapter 4
System Description

4.1 Development History

The Alice project is a large scale, multiyear effort that began in 1992 with the vision of making an interactive 3D graphics system to support exploratory programming [Ungar] for the creation of programmatically controlled 3D object behavior. We specifically did not seek to create breakthroughs in programming languages, new 2D GUI toolkits, or fast 3D-rendering algorithms. Instead, we took for granted that these technologies would exist, and instead focussed our energies on the problem of ease-of-authorship.

Our development strategy was to look for the best "off the shelf" solutions for each of Alice's components while not becoming wedded to any one solution. Our goal was to keep the interfaces between layers small and well-controlled, allowing us extreme flexibility: during Alice's development time, we changed the rendering engine three times, the 2D user interface toolkit four times, and the scripting language twice. A visual summary of the

technologies used in the Alice project over its lifetime appears below.

	1992	1993	1994	1995	1996	1997	1998 (projected)
Language	Tcl; Python					Modified Python	
Operating System	SGI Irix		MS Windows 95			MS Windows NT 4.0	
Alice Process Distribution[5]	Two Process			One Process			
Rendering Library	SGI gl			OpenGL	Direct 3D Retained Mode (D3DRM)		
User Interface	Unix Shell	GNU Emacs[1]; SUIT[2]	Tkinter [3]			Visual Basic [4]	
Hardware	Sun Microsystems (computation) + SGI (render) [5]		Single or Multi-CPU SGI	Intel x86			

Notes:

1 : Early Alice versions ran on a variant of Emacs called Lucid Emacs which allowed us to provide the programmer with pulldown menus for some functions. Where more GUI support was needed, Alice could launch separate GUI tools which we implemented in SUIT and then later in Tkinter.

2 : SUIT : The Simple User Interface Toolkit [Pausch], A GUI library developed at the University of Virginia, and in many ways a predecessor to Alice

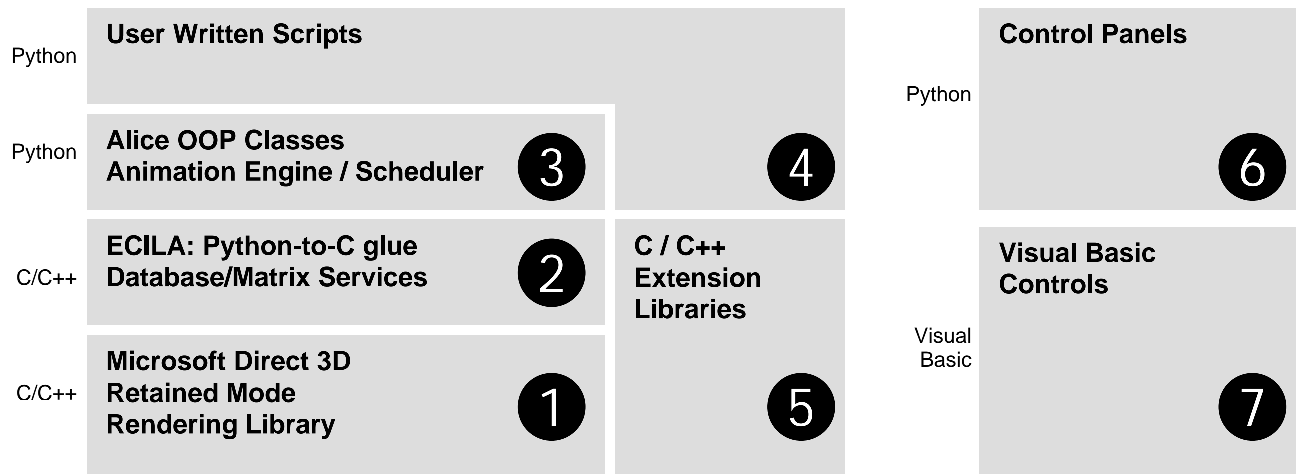
3 : Tkinter is a library that makes the Tcl/Tk toolkit callable from the Python scripting language. During the course of Alice development, we assisted Guido Van Rossum and Steen Lumholdt in the implementation, development, and documentation of this module, which is now in the standard Python distribution.

4 : Alice exploits a standard Python module that allows two-way communication between the Visual Basic elements of the Alice GUI and Python: the controls are Python-callable and are allowed to have Python callbacks.

5 : The earliest versions of Alice used a distributed software architecture that separated the rendering process from the other computations in the virtual world. Each process ran on its own CPU, coordinated over a network connection. This distributed two-process version of Alice was converted into a single CPU, one-process version so as to make it more compatible with commodity PCs running Windows 95. For more information on the distributed system, see [Gossweiler].

4.2 System Architecture

Despite the radical internal changes of its basic software components, the Alice architecture has remained more-or-less stable over the years. In terms of the latest implementation, the architecture is shown here:



Direct 3D Retained Mode (Layer 1)

At the lowest level (layer 1), Alice depends on the Direct 3D retained-mode (D3DRM) graphics library from Microsoft. This layer provides basic services for managing the 3D database of objects and their attributes, texture mapping and lighting the scene, and maintaining the hierarchical tree of coordinate systems for the 3D objects in the scene. Direct 3D also supplies utility services for identifying and using 3D hardware when present. At this implementation level, we also use Microsoft's Direct Sound and Direct Play libraries.

Ecila (Layer 2)

At the next level up (layer 2) is Ecila[1], which implements the glue code that makes the Direct 3D library (a C-callable DLL) callable from Python. Ecila also includes utility support for creating new scenes, performing pick correlation, initializing and cleaning up the database, and querying the rendering layer for information about frame rate and the number of objects in the simulation. Ecila also provides services to the Alice implementation layer above for calculating transformations in arbitrary coordinate systems as well as transformations of points and vectors into and out of the image plane of any camera. Both Ecila and Direct 3D are intended as implementation service layers, never to be seen by the Alice programmer. Ecila is implemented in approximately 25,00 lines of C.

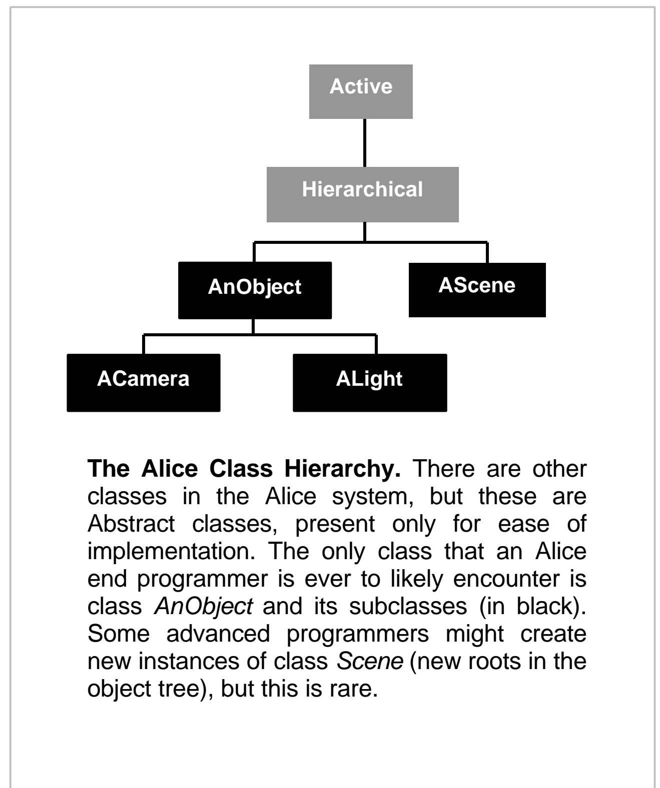
Alice (layer 3)

Above the Ecila layer is the Alice layer (layer 3). This layer contains the three most important components of the Alice system: the OOP Class Definitions, the Scheduler, and the Animation Engine which taken together are implemented in a total of 15,000 lines of Python, 4,000 lines of C and 10,000 lines of Visual Basic .

1 Ecila is Alice spelled backwards, through the looking glass, as it were. Dennis Cosgrove, one of the Alice developers, suggests that this layer is more properly spelled *ecilA*.

The OOP Classes (Component of Layer 3)

The OOP class definitions are the primary set of APIs that are visible to the Alice end programmer. This set of object-oriented classes (AnObject, ACamera, ALight, etc.) give Alice programmers access to Animation control, movement and rotation, and control of parent/child attachment relationships between objects.



The Alice Class Hierarchy. There are other classes in the Alice system, but these are Abstract classes, present only for ease of implementation. The only class that an Alice end programmer is ever to likely encounter is class *AnObject* and its subclasses (in black). Some advanced programmers might create new instances of class *Scene* (new roots in the object tree), but this is rare.

The most interesting classes in the Alice hierarchy are shown in the illustration. From top to bottom, these classes are:

Class Active

This class defines the basic animation API. These methods include stop, start, pause, and resume. Together, these methods define a protocol for talking to the scheduler in a way that doesn't directly expose the scheduler to the end user. The commands are available on an object-wide basis:

```
Bunny.turn (left, rate=1)
Bunny.stop()                  # bunny stops all animations,
                                    # not just the turn command
                                    # that was executing.
```

As well as on an animation-wide basis.

```
spin = bunny.turn (Left, rate=1)
whiz = bunny.move (forward, duration=10)

spin.stop()      # bunny no longer spins
                              # but whiz continues to execute
```

Class Active also implements the methods for reacting to user input (RespondTo).

Class Hierarchical

This class defines and implements an API for manipulating the object tree in Alice (BecomeChildOf, BecomeParentOf). Parent-child relationships define the structure of objects in Alice, forming a traditional PHIGS-like tree [VanDam] of nested coordinate systems. To the end-programmer, this appears as an attachment relationship, though this is a curious one-way attachment by which a parent node moves and carries its children along, but when child objects move, they leave their parents behind. This arrangement allows end-programmers to define movements of parts in relation to their parent object, a technique that has proven useful over the years.

In addition to the standard Parent-Child relationships, this class also provides the API for managing object Parts (BecomePartOf, GetParts, IsPartOf). Alice introduces the notion of object Parts, which are children especially flagged as *belonging* to a parent, not merely

attached to it. More about parts and children is discussed in Chapter 9.

Class AnObject

This class defines the widest part of the Alice API, and is the primary source of commands that Alice programmers have to know. Speaking loosely, this is the only class an Alice programmer needs to know about, which, I would argue, is the source of much of Alice's simplicity, and a characteristic that distinguishes it from the more class-rich APIs in existence (Direct3D retained-mode programmers, by contrast, need to know the interfaces for 16 commonly used classes). Because so much of the Alice functionality is provided through this single class, the system tutorial teaches only that there are Alice commands, and effectively side-steps the issue of teaching classes altogether. Note that the abstraction is that of a single class, though the actual functionality is implemented in several superclasses of class AnObject, none of which the end programmer ever sees. Novice users might make calls to BecomePartOf, never knowing or caring that this function is not (directly) an AnObject method. OOP becomes an implementation detail, a decidedly good thing for our target audience.

AnObject methods fall into several general categories:

- **Geometric Manipulation :** Move, MoveTo, Turn, TurnTo, Nudge, Place, Pan, PointAt, AlignWith, StandUp, SetPointOfView, SetSize, SetScale, MoveToInPicturePlane, MoveInPicturePlane.
- **Property and State Query :** GetPosition, DistanceTo, GetAngles, GetPointOfView, GetBoundingBox, IsHidden, IsCastingShadow, BoundingBoxIsShowing, GetScale, GetTexture, GetTransparentTextureColor
- **Shadows :** CastShadow, StopCastingShadow
- **Textures** : SetTexture, SetTransparentTextureColor
- **Coloring and Rendering**: Get/SetColor, Get/SetVisibility, Get/SetShininess, Get/SetHighlightColor, Get/SetEmissiveColor, Get/SetShadingStyle, Get/SetLightingStyle, GetFillingStyle
- **Vertex Manipulation** : GetVertexPosition, SetVertices, GetVertices, GetFaces, GetVertexCount, GetFaceCount
- **Miscellaneous :** Show, Hide, ShowBoundingBox, GetFilename, Destroy, Store, AttachCamera, MakeTransparentToInput

The semantics for these calls can be found in the online reference manual. Some are discussed in detail later in this dissertation.

Class ACamera

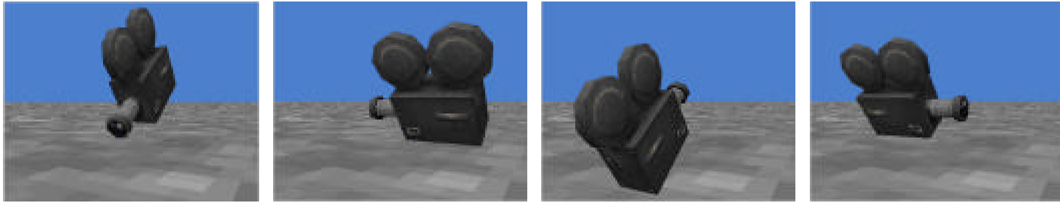
This class defines the API for cameras, which is very much like that of objects, except that cameras have methods for changing their field-of-view (SetVerticalViewingAngle, SetHorizontalViewingAngle)[1] and for managing the output of several cameras into one on-screen window (SendToBack, SendToFront). These functions are all fairly sophisticated in nature, and are not taught to novice users in the introductory tutorials. There are also

1 There exists a prototype Alice API for managing camera field-of-view that makes an analogy to camera lenses (setting FOV in terms of wide angle, telephoto lenses, etc), but this API does not exist in final form at the time of this writing. The current SetFrustum method takes angles as parameters, which is useful at times, but not as familiar or accessible as it could be.

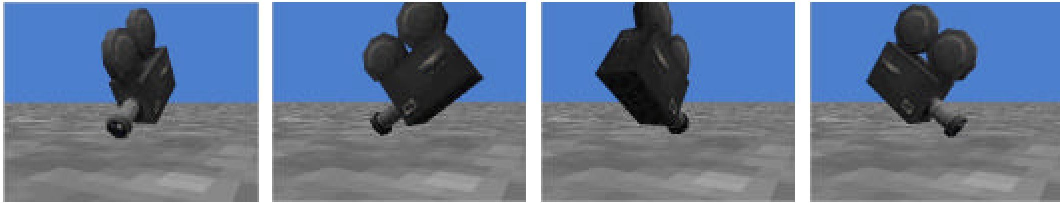
implementation reasons for keeping this class apart from the others, as it is helpful to know when cameras are being created and destroyed, for the purposes of managing system resources.

Cameras support Pan, a rotation command that is somewhat like turn except that the pan command does not rotate the camera around the camera's own up vector, but around the ground's up vector instead.

This is essential for controlling camera motion when the camera is pointed up in the air or down toward the ground. The semantics of Pan are designed to emulate the constraints seen in camera tripods, and to some extent, in the freedom-of-motion seen in the human neck.



camera to end up pointing up in the air and rolled onto its side.



PANNING the camera. The camera rotates around the ground's UP vector, centered on the camera. This keeps the lens pointing at the ground, and more closely emulates the motion afforded by a real camera tripod. The Pan command was inspired by cameras, but is so useful that it is now supplied as a generic method for all objects.

Class ALight

This class implements that part of the Alice API that manages light-specific methods that differ from AnObject: (e.g. TurnOn, TurnOff, SetBrightness). Alice supports several kinds of lights: Spotlights, PointLights, FillLights (what other APIs call AmbientLight) and SunLight (other APIs call this directional Light). The kind of light created is chosen via a parameter to the constructor as opposed to creating four distinct subclsses for users:

```
My_light = ALight ()

            # the default is to create a PointLight
            # though you can create others kinds too…

My_fill_light = ALight (FillLight)
My_spotlight= ALight (SpotLight)
```

Most of the methods for lights are not dependent on the kind of the light (another reason we chose not to use subclsses) though there are a few that do. One such example is a Spotlight-specific method for controlling the width of the beam. SpotLights throw a bright inner beam and a somewhat dimmer outer beam, which have traditionally been called an Umbra and Penumbra respectively. In Alice, these structures are referred to as the Inner and Outer beams and are controlled with methods called SetInnerBeamAngle and SetOuterBeam angle, both given in degrees.

We also found that lights needed a physical embodiment in order to be accessible through direct manipulation. The notion of wrapping a 3D model of a light bulb or a spotlight around the location of a light is not a new idea, but not all 3D programming APIs do this by default. In Alice, the default is to show the geometry, and to let the user call the hide() method on the light if he or she does not want the model to show.

Class AScene

This class implements the Scene nodes, a top-level node in the Alice world, to which all other things are attached. Alice worlds can have several disjoint scenes, each of which is an instance of this class. Scenes cannot be moved or rotated, and cannot be made a child of

anything else. These restrictions required that Scene objects be instances of a distinct class of its own. Other methods that Scenes support: SetColor, which changes the color of the background fill (often interpreted as “sky”, so we make the default Scene color a sky blue) and BecomeDefault, which flags a scene as being the tree to which new objects are added when the user inserts new instances of AnObject into the world.

4.3 The Scheduler (Component of Layer 3)

The classes in Alice define a set of commands that objects can respond to, but the scheduler is the element that determines when those actions are executed. The Alice runtime system is designed as an infinite loop architecture that runs commands in a round-robin scheduling policy. Commands are introduced to the scheduler either by hand (the rare case, usually undertaken by expert programmers) or by running an animated Alice command (one that has a duration parameter which is greater than zero). Running an animated command creates an “animation object” on behalf of the user, one that manages the interpolation of the object’s state over the duration of the animation. This is meant to relieve the end-programmer from the workaday implementation details of interpolation, managing the clock and the sequencing of animations. More about the Alice Animation engine is described in the next chapter.

4.4 Extending Alice in C and C++ (layer 5)

Users can provide any special-purpose functionality through C or C++ extension

libraries (layer 5). The Python scripting language is extensible in C and C++, allowing functions implemented in these languages to be called from Python through a standard programmatic interface that can be generated semi-automatically by filtering the C/C++ header files with software tools that come with Python. These software tools generate C code that translates C and C++ data structures into their Python counterparts and back again, allowing inter-language support through Python's built-in C API. Alice ships with a short example module that shows how this is done, for the benefit of the advanced Alice programmer. For a larger and more realistic example, Alice also ships with Microsoft's DirectPlay Library, originally written in C and made callable from a Python script using this extension mechanism.

4.5 GUI Support (layer 6)

Users who wish to create their own 2D GUI support panels for their 3D applications can use Alice's "Control Panels" package. The functionality of this package is limited, but has the advantage of being extremely simple and that all the logic behind the control panel is written in the same scripting language (Python) that controls the 3D screen objects. More ambitious users can build a Visual Basic tool and arrange for it to be Python-callable (indeed, this is how the Alice main control panel works), but the procedure for doing this is not documented, as it depends on creating and registering COM servers, which is prohibitively

difficult to explain and support.

4.6 The Python Scripting Language

Alice uses an interpreted language for control of 3D objects in the scene. This has been seen by many to be an unusual design choice, given the high-performance demands of interactive 3D graphics. There is nothing particularly strange about choosing an interpreted language for a high-performance system, so long as the design carefully respects the fact that some parts of the system (e.g. the rendering functions) have higher performance demands than others (the simulation and animation control). Some previous systems, such as VEOS [Bricken] made the design mistake of implementing all of the system in a very-high-level language (LISP) which had very serious performance repercussions. Alice, in contrast, uses a scripting layer for the high-level abstractions and codes the performance-critical parts of the simulation in C. This mixed language design allows Alice to maintain adequate runtime performance while remaining flexible enough to allow on-the-fly code changes.

Part of the tension that interactive 3D programmers face is the need for both high speed and high runtime flexibility. As is often the case in system design, these two things trade-off with one another, and traditionally, 3D graphics systems have opted not to compromise at all on speed, probably because for many years even the highest performance systems weren't as fast as researchers might like them to be.

Now, with 3D hardware becoming more ubiquitous and with general purpose CPUs

becoming more powerful, we find that overall system performance is more than adequate to render 3D scenes interactively, yet we as an engineering community too often build systems as if machine performance was still the most important thing to maximize. I believe this places far too much emphasis on machine performance without adequately attending to the more important issues of human performance, ease-of-learning, ease-of-teaching and runtime flexibility. Said another way, it isn't how fast the machine seems to the person, so much as how fast the person is with the machine.

The issue of human performance is particularly important when the task at hand is that of exploring ideas in a new medium. The last thing an explorer needs is a tool that imposes long delays (say, waiting for a compiler) or exacts other penalties for asking exploratory "what if" style questions. A programming language/environment that does not offer this agility is therefore an inappropriate design choice for an interactive 3D graphics development system.

In principle at least, it is possible that compiled languages like C or C++ could deliver on both runtime speed and runtime flexibility, but such languages would need to be accompanied by a sophisticated runtime environment capable of fast incremental compiling, runtime linking and name resolution. While creating such an environment might be interesting and challenging, doing so lies outside the scope of this research. With this in mind, we decided to sacrifice some runtime speed in order to gain the runtime flexibility we needed, and this implied that we should choose an interpreted programming language.

Rather than waste time and effort building yet-another new scripting language, we chose a general-purpose, freely available scripting language called Python [van Rossum]. Python is available via the World Wide Web (http://www.python.org) and is a portable, object-oriented language which features very-high level data structures (lists, dictionaries, strings) and excellent support. A few snippets of Python appear below.

```
# this is a comment.
print "this is a print statement"

# make a list
muppetlist = ["cookie", "kermit", "seymour", "pepe"]

# put them in order and print them out
# note that lists support a built-in sort method
muppetlist.sort()
for critter in muppetlist:
		print critter

# define a new function
# note lack of begin/end pairs
def movethis(x):
		print "moving object", x
		x.move(forward, 1)

# get the Alice module and all its functions and constants
from Alice import *

# make an Alice Object
bunny = AnObject("animals/bunny")

#call the function
movethis(bunny)

# call an AnObject-defined method of bunny
bunny.move(forward, 1)

# call same method again, using optional keyword parameter
bunny.move(forward, 1, Duration=3)
```

As you can see, Python is not a 3D graphics language, but a general-purpose

programming language in the spirit of Perl or Tcl. Unlike these languages though, Python's syntax is extremely clean, carrying fewer noise characters than other languages, having no Begin/End block markers (curly braces in C) or end-of-statement markers (semicolons in Pascal). Python was also an attractive choice because it is extensible in C, meaning that performance-critical functions, such as rendering and database operations, could be coded for speed and then made callable from Python. Like any programming language, Python is not without defects. I discuss some of these flaws in section 6.10 along with a discussion of some of the language features missing from Python that would be most helpful.

Python was not the first programming language we used with Alice. For the first six months of Alice's history, we used the very popular language called Tcl which, like Python, is embeddable and extensible in C and therefore seemed ideal for the core language of an interactive scripting environment. Unfortunately Tcl proved untenable in the long run because of idiosyncrasies of the language syntax, in particular the rules governing the quoting and evaluation of expressions. Keeping track of the various quoting rules and the places where whitespace was and was not allowed proved too cumbersome for the development team to bear and we believed would be far too hard to teach to our target audience. Ironically, Tcl has some interesting features that recommend it for novice use: Tcl is space delimited, with an extremely simple and regular syntax for simple commands with access to advanced features through keyword parameters. Unfortunately, this cleanliness of design is not carried into the design of the syntax for expressions, if statements and while loops, which can be anything but simple [Conway].

4.7 Modifications Made to Python

During the course of development, we tried very hard not to modify the Python language. Making modifications to Python would mean that as the language implementation changed, we would have to repeatedly apply our local modifications, a clearly undesirable way to spend one's time and limited resources. This decision to stay with Python "as is" caused us to steer the Alice design in certain directions, avoiding solutions that required language features that Python did not have (e.g. operator overloading on assignment, for example). Unfortunately, we were ultimately forced to abandon pure Python when testing revealed two flaws in the language that made Python unusable by our target audience: integer math and case sensitivity.

Integer math

Curiously, Python defines mathematical operators between integers to perform integer math, which states that if a division between two integers results in a fraction, the fraction is truncated, leaving the integer part as the result. For example, the numerical expression 1 / 2 (one divided by two) evaluates in Python to be the integer 0. In observing novice users, we noticed a preference toward typing fractions in (numerator / denominator) form, as opposed to decimals. This, of course, was a pedagogical disaster to the novices who encountered this problem who then needed a (painful) explanation as to why 1 / 2 was really 0.

Rather than teach the concept of integer truncation to novices, we altered the Python implementation so that expressions involving integers would engage the floating point routines, and thus return floats when appropriate, integers when possible. The "loss" of the integer math "capabilities" has never been noticed either novices. In those rare cases when a programmer needs the semantics of integer truncation (as was indeed the case in the Alice implementation itself), programmers can use Python's `divmod()` library function.

Case Sensitivity

Python is case sensitive. While we, as programmers, were comfortable with this language feature, our user community suffered much confusion over it. At least 85% of users[1] who were observed using the Alice tutorial made a case error at some point during the experience. While explaining the case rule was simple enough ("upper and lower case mean different things to Alice"), this was not sufficient to instill a "case aware" sense in our users. Of the users who had problems with case, most *continued* to type case-incorrect tokens in their programs for a short period. Coming to terms with case sensitivity is a difficult skill for many to learn, a fact that can often be lost on experienced programmers. Case sensitivity is an artificial rule that fights against older knowledge that novice users have, namely that while `forward` and `FORWARD` may *look* different, they should at least *mean* the same thing. A case-sensitive language refutes this intuition.

---

1 Out of a sub-pool of 65 users. The case sensitivity issue was one that came to our attention late in testing and so I do not have data for the full set of Alice users in the tutorial observation sessions.

To make matters worse, the specific stylistic case rules for the Alice API were very confusing to novices. We employed a rather standard set of rules that most programmers would find familiar: constants in ALL_CAPS, class names in LeadingCapsMixedCase, method names in lower_case_and_underscores. This kind of consistency helps programmers understand, read and navigate through large programs, but to a novice, these rules are arbitrary and confusing. To explain that FORWARD is a constant and that move is a method makes no sense to a person who does not know or care what a constant or a method is. To the novice, these are both just "words" that are used in the Alice command language; words that play more-or-less equal roles in the novice programmer's mind. As such, novices perceive the case rules not as evidence of organization, but of deep inconsistency.

Eventually, novices who wish to gain more programming skill will have to learn these concepts, but until that time, it is pedagogically unwise to burden the novice programmer with such details. By using a case sensitive language, we had inadvertently turned understanding of constants, methods, and classes into prerequisites to doing even the simplest things in Alice. We came to believe that the case sensitive nature of Python would prove to be an impenetrable barrier to most of the people we were trying to reach and therefore changed the Python implementation to casefold all input before interpreting it. This change turned out to be fairly minor, and has been contributed to the Python Software

Activity[1] for inclusion as a runtime option in the next major release of Python.

I note with some embarrassment that Hypercard, Pascal and LOGO were designed for novice or infrequent programmers and each was case insensitive. It may be that Microsoft's Visual Basic programming environment provides the best of both worlds by allowing the user to type in a case-insensitive way, while the programming environment applies the proper case to the program text on behalf of the user whenever possible.

The Value of Informal Observations

Some circumstantial evidence is very strong, as when you find a trout in the milk
Henry David Thoreau

Both integer math and case sensitivity show how informal user observations can be used to alert a development team to the existence of a problem in an interface without having to perform long and expensive usability tests. Fred Brooks cautioned the user interface community [Brooks88] to measure the right things at a confidence level that is appropriate to the task, and it is in this spirit that I claim that informal observation can be one of the most powerful techniques that a user interface designer employs.

Once a designer observes a problem, there are several choices:

Fix the system to remove the problem. This a designer's first choice. In the case of

---

1. The PSA is a consortium of Python software developers led by Guido van Rossum: http://www.python.org

the integer math problem, this is what we did despite the fact that doing so required changing one of the lowest-level components of the system

Teach around problem by working the problem into a pedagogical moment in the tutorial. For example, we could have chosen to make part of the Alice tutorial a lesson in integer mathematics, explicitly leading a student through an exercise that demonstrates the bug (e.g. "see what happens when you move an object ½ meter…")

This technique is appropriate when the bug cannot be fixed, but neither can it be avoided by a novice user. Unfortunately, the cost in taking this course of action is that it imposes yet one more lesson on the beleaguered user. Said another way, the user's cognitive abilities should be thought of as a finite resource to be used with great care, as it is limited in both quantity, quality, and in duration. In the case of Alice design, every lesson we choose to teach about integer math is (at least) means the tutorial can contain one fewer lesson about 3D behavior programming.

Leave the bug in and document it is an option of last resort to be done when a bug is either too hard or too expensive to fix. Users have understandable resistance to reading documentation, and so taking this course of action is very much like doing nothing at all.

4.8 Chapter Summary

- The Alice research agenda focuses on the specification of behavior. There are certainly many other interesting 3D graphics research questions, but we leave these for others to pursue, and choose to incorporate the latest technology into the Alice implementation whenever feasible. This research strategy requires that the Alice design be sufficiently modular that we can remain free to swap out different technologies (languages, rendering libraries, etc.) as the field develops.
- Alice currently uses Direct 3D for rendering, though it has used other rendering libraries in past implementations.
- Alice uses a modified version of Python for parts of the implementation and as the scripting language exposed to the end programmer. Using this language allows us to support an "exploratory programming" style that favors the programmer's ability to try new ideas, at some expense in performance.
- Alice has a small collection of classes (Objects, Lights, Cameras, Scenes) that are exposed to the end user, though most 3D object functionality comes through the AnObject class, a fact that allows us to side step most of the complexity that comes with object-oriented programming when first teaching the system to new users.
- Experienced programmers can create Alice programs that call out to C and C++ libraries.
- Novice programmers can create simple GUI control panels using Alice.
- Case sensitivity is a bug in programming languages intended for novice users. We removed it in the version of Python that ships with Alice.

Turning and turning in the winding gyre…
W B Yeats

Chapter 5
The Alice Main Loop

5.1 Introduction

This chapter describes the main loop that governs the behavior of objects in Alice. Each turn through this loop defines one frame in the Alice simulation, an analogy to the frames in motion pictures. During each frame, Alice gathers input events, updates counters and clocks, and most importantly, runs the animation engine, which increments the state of every object undergoing a state change (moving, rotating, changing color, etc). Finally, at the end of the loop, Alice redraws the contents of the window. All of this has to happen fast enough so that the user's perceptual system fuses the individual renderings into an illusion of smooth motion, no slower than once every 1/10 of a second. This defines a realtime deadline: taking longer than this will cause the animation to "break up" into a series of flashed still images and the system fails to perform correctly. Alice's main loop, in

pseudocode[1], looks like this:

```
while (TRUE):
        ComputeAnimations()
        ComputeReactionsToAlarms()
        ComputeEachFrameActions()
        GetAndProcessUserInput()
        Render()
```

5.2 Elements of the Alice Main Loop

ComputeAnimations() is responsible for managing the animations in Alice. An Alice animation is any state change that is supposed to take place over the course of some interval of time (always expressed in seconds). These time-based state changes usually result in the movement of graphical objects but can also be a color or opacity change. This part of the Alice main loop is covered in greater detail in section 5.3.

The next phase of the main loop is ComputeReactionsToAlarms(), which manages all the calls to Alice.SetAlarm (some_function, time_interval). This method allows Alice programmers specify an arbitrary Python function (some_function) to be called at some number of seconds in the future (time_interval). Alarms like this are sometimes useful in animations where the start time of an animation is more easily expressed in terms of time (e.g. perform action X three seconds from now) than in terms of some other condition (e.g. perform X when Y is finished). Alarms are kept in a priority queue, with the closest deadline

1 The main loop for Alice is implemented in a combination of Python, Visual Basic and C, and does not contain functions with precisely the names seen here. This code listing is simplified for pedagogical purposes.

kept on top of the heap for fast access. Each frame of the simulation, this alarm is examined to see if its deadline has expired, and if it has, the system calls the function associated with that alarm. This implementation is not perfect, of course, only guaranteeing that the user's function is called *no sooner* than the time specified. In the worst case, an alarm can be late by nearly a full frame time, which is typically about 1/10 second, but could be arbitrarily late in a pathological case. Under these conditions, the interactive and real-time nature of the simulation is lost anyway, indicating that the application has more serious problems than simply missing a desired alarm.

There are more effective and sophisticated ways of handling real-time deadlines, though the use of these techniques is not yet widespread. One system that employs a more sophisticated treatment of realtime constraints is the research prototype known as *ReActor* [Ball]. In ReActor, animators can specify certain "critical times" – moments at which a frame must be rendered, even at the expense of rendering other frames. Take for example, an animation of a bouncing ball on the floor. To make this animation look compelling, we want to make sure that the ball is shown in contact with the floor. The simulation state that has the ball on the floor is noted as a "critical time", which guarantees that this frame is rendered even if doing so requires dropping a frame or two before the critical moment of contact. ReActor does this by estimating the current frame rate and using it as an indication of how much time is left before the critical time deadline. Should ReActor predict that rendering the current frame would mean being late for an upcoming critical time, ReActor chooses not to render, saving time so as to meet the more important deadline.

ComputeEachFrameActions() implements the "back door" into the Alice scheduler letting programmers specify arbitrary functions that should be executed once per frame. This is done through the Alice Do method and the EachFrame option, as in: Alice.Do(some_func, EachFrame). Calling a function in this way is handy for maintaining a constraint throughout the simulation (objects do not interpenetrate when they collide, for example).

The functions in the scheduler list are evaluated in the order in which they appear in the list, an order which is determined by the order in which animations are started, but this fact is not generally guaranteed by the Alice implementation. It is possible to create simulations where the order of evaluation is important (say where one spring needs to push on a second spring, which in turn presses on the first) though in practice, we find this case is not as common as one might think. There is clearly a question of cause and effect here that is difficult to untangle: would scripts with cyclic data dependencies be more common if Alice encouraged or supported the writing of such scripts?

On those occasions when order does matter, there are two options in the current Alice implementation. The first option is for the programmer to use an API that the Scheduler provides for obtaining, reordering, and setting the list of functions manually. The second option is for the Alice programmer to introduce a "manager" object that takes care of the inter-object dependencies manually. This design reflects the philosophy that it is often better to provide a mechanism that is overly simple than a mechanism with a lot of sophisticated features. Both simple and sophisticated systems can fail, but when simple

mechanisms fail they are are easier to understand and therefore easier to debug than sophisticated systems precisely because the sophisticated systems are complex. Of course, sometimes complexity can help avoid error conditions altogether, but unless the mechanism is guaranteed to "do the right thing" in all conditions, it is often better to have no mechanism at all than one that is hard to debug.

GetAndProcessUserInput() gathers input events from the keyboard and mouse, dispatching the events to either a 3D Alice object or to a Visual Basic GUI element. Events that happen over Visual Basic elements are handled internally to those widgets, those that happen over 3D Alice objects are dispatched to the event handlers (Python functions) that are associated with those objects, should they have any. Programmers can set and unset event handlers to objects unset using obj.RespondTo(event, function), which takes two parameters: an event to react to (one of: LeftMouseUp, LeftMouseDown, LeftMouseClick, RightMouseUp, RightMouseDown, RightMouseClick, MouseMove, KeyUp, KeyDown) and a Python function to call when that event happens. Passing the Python constant None as the function to call causes the handler for that event type to be removed.

Render() accesses the underlying Ecila layer to update the screen out of the retained mode database being held in the Direct 3D layer. The performance of most Alice simulations is bound by the speed of this function, though this is likely to change as 3D accelerators become more common (and more powerful) on the PC platform.

5.3 A Closer Look At the ComputeAnimations() Function

The ComputeAnimations() function is the first phase in the Alice main loop and is the heart of the Alice Animation engine. This engine is a simple interpolation system that tracks the progress of all currently running animations, manages their start and stop times, and interpolates the time-varying state of the animations until their completion.

Time Based v. Frame Based Animation

There is one thing that differentiates the realtime animation from batch animation – realtime animation cannot make *a priori* assumptions about the instantaneous rate at which individual frames will be presented to the user. Batch animation, as used in the construction of computer generated movies such as Pixar's *Toy Story*, are animated in such a way that every individual frame represents 1/24 of a second of action. This time interval is constant, and can be relied upon for purposes of pacing the action of the film. Every image is always 1/24 of a second away from the preceding frame. This is not true in interactive 3D graphics.

In the realm of interactive 3D, the images are not going to be written onto a constant speed medium like film, but are to be played to the screen in real time. The individual frames of animation may vary in complexity, and therefore will vary in the amount of time they will take in order to render. This means that at every frame of the simulation, the Alice animation system needs to query a runtime clock to determine what fraction of an animation has expired. While a batch animator can rely on 1/24 of a second of action per

frame, an Alice animator needs to compute this number.

Why not just use frames? We could just allow Alice animators to specify that animations are to take N frames rather than N seconds. This would be a seriously flawed approach because it would imply that animations would run faster on better hardware. By forcing animations to be specified in seconds, we guarantee that an animation that takes one second today will still take one second to execute on tomorrow's hardware (will be smoother). Early video games for the PC made the mistake of using frame-based animation and as a result many popular games became unplayably fast when PC clock speeds rose.

In contrast, Alice uses time-based animation, meaning that users specify the number of seconds an animation should take, and the implementation renders an animation of the appropriate length by using a call to gettime() to pace the frame-by-frame progress of the animation. On a fast or lightly loaded CPU, the time between frames is small, leading to smoother motion; on a slower machine, frames will take longer to render and therefore Alice renders fewer frames for the same animation, leading to choppier motion. In either case, the animation takes the same amount of real-world time (possibly exceeding that time by the duration of one frame.)

By default, commands in Alice execute as one-second animations when there is a sensible way to do so. If a user wishes an animation to take longer than a second, he or she can supply a new value through a Duration keyword added to the command.

Consider a user script that contains the following code:

```
Bunny.Move (Forward, 1, Duration=5)
```

This command actually does not move the bunny directly; it launches an animation that, in turn, does the movement, a little bit each frame of the simulation, over a span of 5 seconds (in this example).

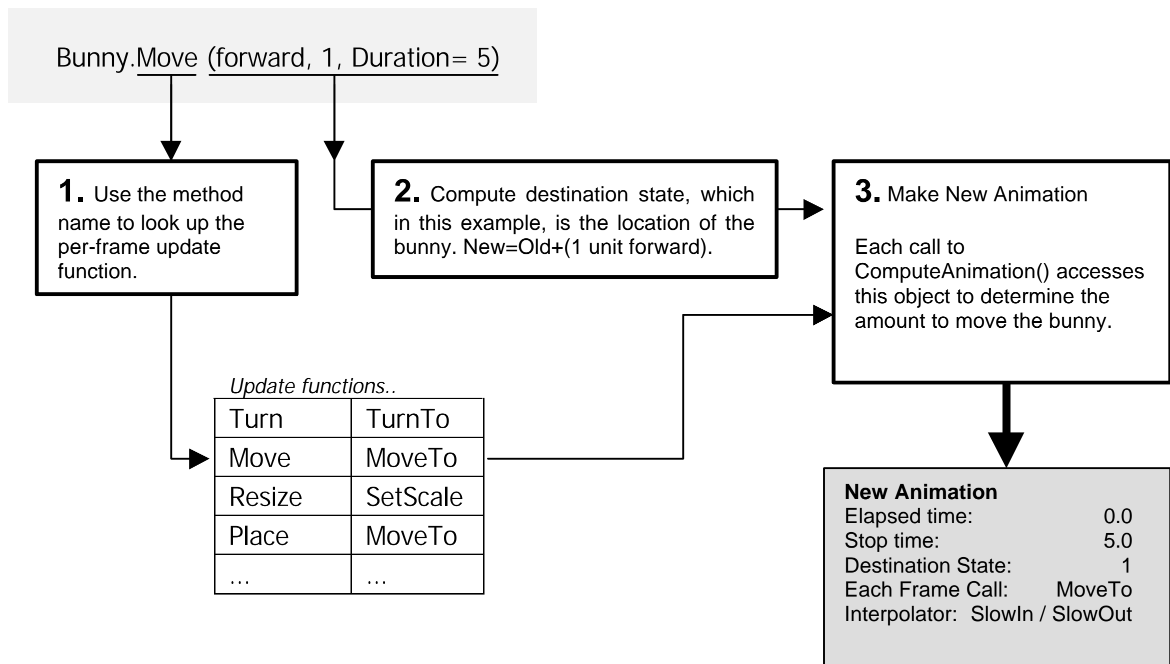
For this to work, each animatable method in Alice has to have associated with it a lower level "update" function that is called once per frame during ComputeAnimation.

In the bunny example above, the user has called the move method on the bunny. At the time that this code is evaluated, the Alice system creates an animation object that encapsulates within it the following information::

- The object being animated: bunny
- The method to call each frame: UpdateMove (a private update function to move)
- The total distance to move: 1 meter
- The total execution time 5 seconds

This Animation Object is registered with the system and will persist in the "current list of animations" for the duration of its run (5 seconds in this case). Each frame of the Alice main loop, this object is accessed by ComputeAnimations() which computes the amount of time that has elapsed since the last time through the loop, and adds this to the total elapsed running time of the animation. If this time is greater than the total execution time, the animation is advanced to its final state and is removed from the list of currently running

animations. If the animation is not over, Alice computes how long the animation has been running as a fraction of its total duration, and uses this fraction as an interpolation quantity (e.g. if 2 seconds have elapsed, the 5 second animation is 2/5 = 40% of the way through the animation.) The process is outlined in the diagram below[1]:



By default, Alice uses an interpolation curve that gently accelerates objects and then gently decelerates them to a stop. Alice animators can override this default with the Style keyword[2] and the name of an alternate interpolation function or an interpolator of their own creation. Of course, this is something we would expect only from an expert user. The Alice

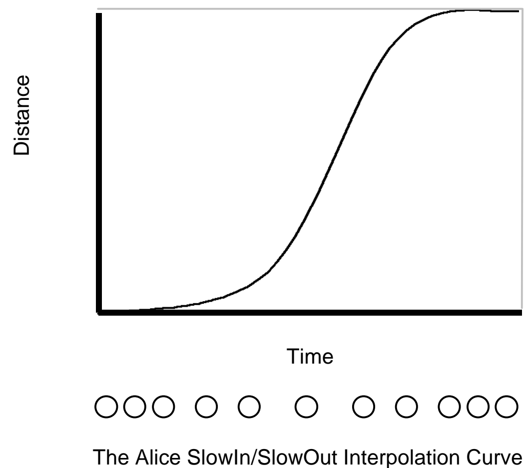
---

1. This diagram is slightly simplified – the global table of update functions does not actually appear in the implementation, but is helpful for purposes of explanation.

2 Standard linear interpolation is done like so: bunny.move(forward, 1, Style=Abruptly)

documentation contains a careful pedagogical treatment of animation interpolation.

Note that all of this is automatic, that the user has only written a single line of code, `Bunny.move(forward, 1, duration=5)`, and the rest of the details managing the animation are taken care of by the Alice implementation, which contributes greatly to the compactness of Alice scripts and distinguishes Alice from much software in its genre.



The Alice SlowIn/SlowOut Interpolation Curve.

5.4 Compound Animations

Simple animations (single animated Alice commands that have a duration>0) can be combined to form more complex animations through the use of two commands: **DoTogether** (which launches two or more animations at the same time) and **DoInOrder** (which causes two or more animations to be sequenced, launching one after another.) These commands each create Animations themselves, so that the results of **DoTogether** and **DoInOrder** can then be re-used in more complex animations:

5.5 DoInOrder

```
myanimation = DoInOrder (
                    bob.move(forward, 1, Duration=1),
                    bob.turn(left, 1, Duration=1)
                   )
```

DoInOrder

The code example shown above will make the turn command happen after the move command is finished. The entire DoInOrder animation is considered done when the last animation in the list is finished. This example has two commands in the DoInOrder, though in practice, this command can have an arbitrary number of parameters.

5.6 DoTogether

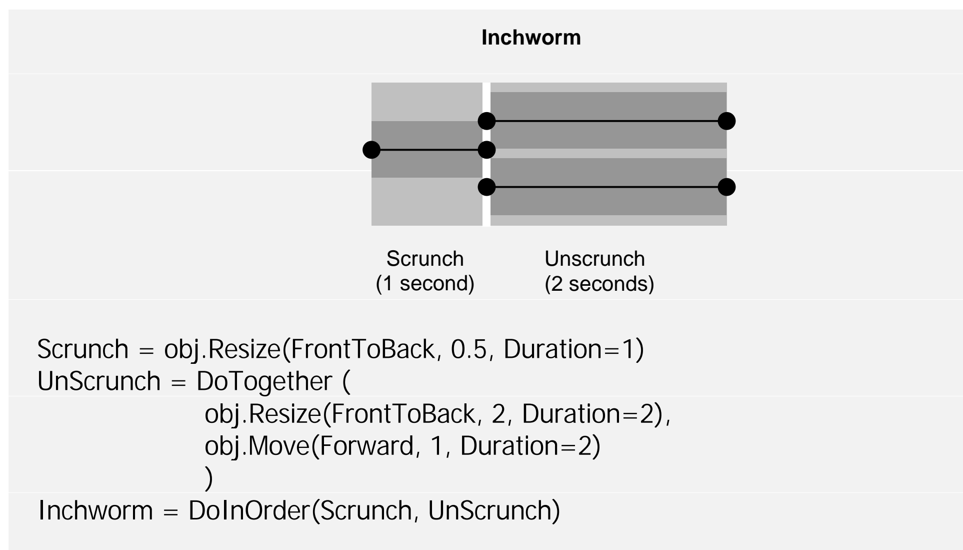
```
myanimation = DoTogether (
                    bob.move(forward, 1, Duration=1),
                    bob.turn(left, 3, Duration=2)
                    )
```

DoTogether

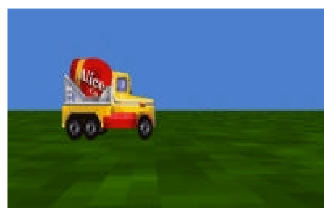
This code example will cause the move command and the turn command to start at the same time. Notice that each command placed inside a DoTogether expression can have a different duration, so even though the individual animations start together, they might not stop together. The DoTogether animation is considered done when the longest running animation inside it is finished. This example has two commands in the DoTogether, and has a total duration of three seconds.

5.7 Composing DoTogether and DoInOrder

Like all animated Alice commands, both DoTogether and DoInOrder return values of type Animation, allowing them to nest freely inside each other. A simple example of this appears below: an inchworm animation made up of two parts: scrunching up followed by the parallel animation of unscrunching while moving:



Dump truck starts at left.



Dump truck contracts to 0.5 of original length.

Dump truck resizes back to full length while simultaneously moving forward.

Dump truck stops moving one meter to the right from its starting position.

5.8 A Word About Syntax and Evaluation of Parameters

The syntax for DoInOrder and DoTogether deserves a word of comment. Note that the actual parameters passed to DoInOrder and DoTogether are Alice expressions. As with any function call, expressions should be evaluated at the point of call and should therefore launch animations, one per command inside the parentheses of a DoInOrder or DoTogether. After user observation, we discovered that this behavior would be extremely confusing, as it would cause the definition of an animation to be executed as the animation was being defined. To keep this from happening, it might appear at first that we would need to suppress the evaluation of arguments that are passed to the DoInOrder and DoTogether functions. While it is possible in some languages to do this (e.g. LISP and many of its dialects like CLOS [Steele]), usually in the form of passing closures[1] it is not possible to do this cleanly in Python.

The expressions being passed to DoInOrder *are*, in fact evaluated at the point of call! The arguments are evaluated, and as usual, they create animation objects which are then passed into the DoInOrder call, at which point the DoInOrder calls Pause() on each of the parameters it is passed. This is what keeps the animations from actually executing before their time: DoInOrder stops them in the same frame in which they are started. Once the animation is defined, Alice launches the finished compound animation. To prevent Alice

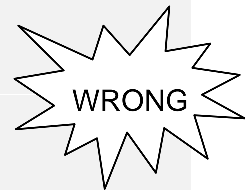
1 A *closure* is a single object that collects together a function (any token that represents a callable object) and enough of the calling context so that its arguments can be evaluated, usually at some later point in time.

from doing this, the programmer can pass Start=No to DoTogether or DoInOrder.

One clear drawback of this approach is that the arguments to DoInOrder have to be instances of class Animation (so that Alice can call Pause()). This restriction prevents a user from inserting arbitrary code (e.g. an if statement or a debugging print statement) into a DoInOrder. To get around this shortcoming, Alice supplies a command called Do() which takes a single function and wraps it in an animation object, suitable for passing to DoInOrder or DoTogether.

For example, this code example does not work as intended:

```
DoInOrder (   truck.move (forward, 1),
              print "just moved the truck",
              Truck.turn (left, 1 / 4)
              )
```



The print statement inside the DoInOrder is a Python command, not an object of type Animation. As such, evaluating this command raises an exception and an appropriately worded dialog box which explains the problem.

To fix this, the user must create a function and convert that function into an animation object with the Do command:

```
def myprint():
            Print "just moved the truck"

DoInOrder (   truck.move (forward, 1),
                     Do(myprint),
                     Truck.turn (left, 1 / 4)
                     )
```

This is clearly a potential problem for novice users.

5.9 Alice Animations and Threads

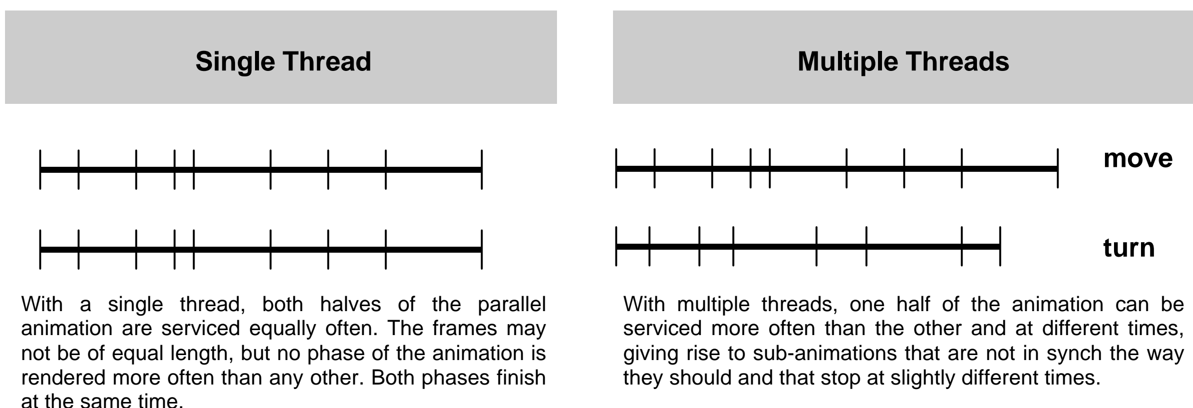
The Alice Animation API is a mechanism for launching parallel threads of control and so it would seem logical to implement this mechanism on top of the threading services offered by the Windows 95 operating system. As a proof of concept prototype, we designed and built a threaded version of Alice, but the results were surprisingly unsatisfactory. To understand why, it helps to look at the simple inchworm animation again:

```
Scrunch = obj.Resize(FrontToBack, 0.5)

UnScrunch = DoTogether (
                        obj.Resize(FrontToBack, 2, Duration=2),
                        obj.Move(Forward, 1, Duration=2)
                        )

Inchworm = DoInOrder(scrunch, unscrunch)
```

In the animation called UnScrunch, the Resize and Move commands will run in parallel and together will take two seconds to execute. During those two seconds, a single-thread scheduler will execute functions that incrementally Resize and then incrementally Move the object, once for each frame of the animation. Each time through the simulation loop, the object gets exactly one update to its size and exactly one update to its position. This relative

frequency-of-service (Resize is updated exactly as often as Move) is guaranteed because the scheduler is implemented in a single thread of control and the updates happen in order each time a frame is rendered.

A threaded implementation of the scheduler would typically use one thread for each branch of the parallel UnScrunch animation, so one thread runs the Resize interpolation and another thread runs the Move interpolation. The implementation hands over frequency-of-service concerns to the operating system's thread scheduler, which sometimes services one thread slightly more often than the other. This means that in any given time interval during the two second duration, the operating system may decide to update the object's size more often than its position, or vice versa, giving rise to an uneven and jerky looking animation. While this would seem to be a small effect, it turns out that humans are remarkably senstitive to the effect.



With a single thread, both halves of the parallel animation are serviced equally often. The frames may not be of equal length, but no phase of the animation is rendered more often than any other. Both phases finish at the same time.

With multiple threads, one half of the animation can be serviced more often than the other and at different times, giving rise to sub-animations that are not in synch the way they should and that stop at slightly different times.

We found that although Windows 95 does a generally satisfactory job of load-

balancing the threads, it doesn't do a perfect job. Seeing this, we abandoned the multithreaded implementation for a more directly controllable single-threaded version of Alice, and implemented our own scheduler to manage simultaneous animations. [1]

The Illusion of Threads

Even though Alice's implementation is single threaded, the Alice experience from an end-user's point of view remains one of having multiple threads of control. Alice launches an independent "thread of control" (animation interpolation and updater) every time an animation is started, but unlike other systems, Alice doesn't put the burden of managing that thread (forking, exec'ing, terminating) on the back of the programmer. Instead, the Alice design reflects the belief that multiple threads of control are the rule, not the exception in interactive 3D programming, and so the system should make this sort of thing so easy to program that is becomes the default behavior. The behind-the-scenes launching of threads is a significant contribution of this work.

As a side note, the Disney Imagineers who built the Aladdin *Player* animation engine noted that this policy of thread management was one of the most attractive differences between Alice and *Player*.

1. Some have speculated that threading on a per-object basis might be more successful.

5.10 Sequencing Actions: When Do Animations Start?

Animations in Alice are launched as soon as they are defined. Thus, the script below:

```
Cube.move(forward, 1)
Cube.turn(left, 1)
```

will launch a move animation, followed immediately by launching a turn animation. Because both animations will be started in the same frame of the simulation, both of these one-second animations will end at the same time too. Thus, Alice scripts have the unusual property that sequential lines of code do not necessarily execute in sequence. Animations always launch in parallel.

There are times when an animation is not meant to be executed, but is simply being defined so that it can be explicitly started or stopped at some later time using the `start()` and `stop()` methods. For example, in the `Inchworm` example given above, the `Scrunch` and `UnScrunch` sub-animations are not really meant to be executed, they are meant to be run in the context of the `Inchworm` motion. Whether the `Inchworm` animation itself executes immediately is a decision best left to the programmer. It might be a command worth running, or it might simply be an animation definition, intended to be run later. We can leave this decision with the programmer, but as API designers, we have to decide: what is the default behavior? Do animations launch as soon as they can, or must they wait for an explicit starting signal from the script writer?

One thing we do know: when teaching programmatic animation to novice users,

students are confused by the notion of animations that do not run once they are defined. Definition without execution is seen as a kind of silent failure that has to be "fixed" with a "workaround" of explicitly having to call start() on the newly defined animation. Given that animation is hard enough as it is, we decided to make all animations launch by default. To define an animation without actually running it, a programmer can add the Start keyword parameter to the function call:

```
Inchworm = DoInOrder(scrunch, unscrunch, Start=No)
```

This decision would seem to favor novice behavior and expectations over those of experience, but I would argue that a programmer who is experienced enough to realize that delayed execution is a feature worth having is probably experienced enough and is clearly ready to learn that a keyword parameter is necessary to get delayed-execution semantics. By setting the default this way, we have shifted the pedagogical burden from the inexperienced to the experienced programmer, freeing the novice to concentrate on more immediate issues while also respecting the Law of Least Astonishment and the principle of Controlled Exposure to Power. Note that if we had an assignment hook (see future work section) we could detect when an animation was being created and being assigned to a variable name, and therefore we could create those animations in a paused state. Unfortunately, Python does not easily afford us this power and so we have to live with this minor system flaw.

Ideally, we would also like to service the needs of the long-term Alice user who might find these novice-directed semantics somewhat burdensome. Future work might

explore the benefits of language support for the building of APIs where the functional defaults (starts=Yes) are settable by the *client* of the API, not the designer, on a function-by-function basis. This raises questions of finding ways that such a programming language/environment can support the typical activities of program writing, reading, debugging, and the understanding of legacy code.

5.11 Summary

- Alice is implemented as a single loop with different phases that manage the various parts of constructing the state for each frame.
- Alice can handle Alarms, but does so in a straightforward way. There are more sophisticated models.
- All animations in Alice are time-based.
- Most commands in Alice are animated by default.
- Animated Alice commands create Animation objects that the implementation uses in order to manage the on-screen action. Animations obey a simple algebra that allow them to be run in sequence or in parallel with other Animations.
- Not all commands in Alice create Animations. To make such commands into animations, the programmer uses the Do command.
- Threaded animations sound like a good idea, but require fine-grain control over the frequency-of-service for each animation thread to make sure that threads are evenly balanced in terms of CPU time. Creating one thread per object might also be a viable design direction.
- Alice runs animations in parallel by default, which relieves the programmer from the burden of having to launch and manage separate threads by hand but has serious usability concerns.

You can observe a lot just by watching
Yogi Berra

Chapter 6
Empirical Evidence

6.1 Introduction

One thing that distinguishes my work from previous attempts to create 3D APIs is the extent to which these ideas were tested against members of our target audience: 19 year old undergraduates outside the engineering disciplines. The testing technique used is largely informal and qualitative in nature, meant to detect the worst problems in the GUI, the tutorial and the API design. This chapter discusses the nature of my subject pool, the two techniques used when observing these subjects interacting with the system and the nature of some of the evidence that was gathered. Subsequent chapters will discuss the specific observations made from these sessions.

6.2 The Primary Subjects

The user observations of the Alice system happened in two large waves, a small one

in 1994, and a much larger effort in 1996-7. The early observation session was a graduate level graphics class of 20-25 students that was given the opportunity to use the SGI version of the Alice system. This set of users were graduate students and exceptional undergraduates, most of whom were in engineering, and as such, this group falls well outside of our target user base. Observing this set of near-experts (relatively speaking) was extraordinarily helpful, in that this group was better equipped to talk about their experiences than a similarly sized group of novices. Many of their suggestions caused radical changes in the Alice API, including the notion that the scale operator (later renamed to resize) should not have the side effect of scaling space for an object.

The second set of users was drawn almost exclusively from our population of target users: people who had no computer programming or linear algebra training. These subjects were recruited over the course of several months via email lists, local USENET news postings, and paper flyers posted around the University of Virginia grounds and the local Charlottesville community. In all, nearly 200 people responded to the call for volunteers. Some were rejected because they were deemed to be “too knowledgeable” and others rejected because they were unable to invest the 60-90 minutes required to finish the observation session. In the end, we observed a total of 104 subjects in this second wave of subjects.

As a final note, other researchers have reported the importance of making the development team participate in the usability testing process [Wixon]. Doing so can sensitize

the developers to the mindset of their users.

6.3 Two Person Talk Aloud Protocol

All users (except ten, see below) from this second group were asked to visit the User Interface group lab space in the University of Virginia Computer Science Department. Here they were given a brief introduction to the motivations behind the Alice research, asked a few preliminary background questions, and then given a paper copy of the Alice tutorial and asked to sit side-by-side in front of a Windows 95 PC running Alice, two subjects per computer. A small subset of subjects (ten in all) were not tested in the UIgroup lab space, but were approached at a local coffee shop, *The Mudhouse*. This shop had a high-performance PC in the corner and the owner allowed me to use the facility for user observation on an occasional basis. This testing was done in this fashion for opportunistic reasons; the computer was available and the people in the coffee shop were generally members of the Alice target audience.

The two person talk aloud protocol is a standard and well-known technique for testing the usability of end applications [Nielsen]. In this protocol, pairs of test subjects are observed in the pursuit of a specific task, with the observers either sitting nearby or watching the interaction remotely from behind one-way glass. The goals of such observation sessions is to uncover the problems that users are most likely to encounter in the pursuit of a given task with the software. Often times the severity of usability problems can also be assessed.

There are two important characteristics of this form of observation:

(1) The observer not interact too aggressively (preferably not at all) with the subject being observed.

(2) The subjects are observed in pairs in a problem-solving task. This is vital as it gives the observer the ability to eavesdrop on the conversation between the two subjects as they proceed through their task (in this case, a printed tutorial.) This strategy of observing pairs of subjects has several advantages over using a single subject talk aloud protocol. With a single subject, the observer must rely on reports from the subject about the “internal conversation” they are having, a report that often has to be coaxed out of the subject by interrupting the test with admonitions like “what are you thinking right now,” and “did that do what you expected?”. This can be jarring and disruptive, and clearly biases the data gathered. Also, some subjects find it difficult to talk out loud when they are the only one being observed either because it requires introspection skills they may not have, or because talking to oneself is somewhat awkward or embarrassing. In either case, two-person talk aloud protocol works well in overcoming the silence problem often seen with single-subjects.

The difficulties with one person talk aloud protocol has been commented on before in the usability testing of end-user application GUIs, but I believe that one person talk aloud protocol is even more damaging in the domain of API testing, due to the highly cerebral nature of programming tasks where interruptions and proddings from an outside observer

can be even more disruptive to the task, thus biasing the tests.

At the risk of painting with an overly-broad brush, I would note that on average, women in our test subject pool were somewhat more forthcoming with data (and therefore more valuable) than were the men. Women, on average, tended to talk more freely on their own and to their test partners about their own mental states, and the problems they were encountering. In contrast, men on average, tended to be somewhat more stoic, preferring not to talk until they had mulled a problem over or explored options silently on their own. These observations are clearly generalizations, as there were men who were quite articulate and talkative, and women who preferred to remain silent, but the overall behavior of the test subjects adhered to these general stereotypes.

Once users were done with the tutorial, they were asked to leave the computer and to sit with the observer for another 15-30 minutes in a debriefing session during which they were asked questions about their experiences and about future experiences they might have with Alice. The content of this debriefing changed over the course of the observation sessions, as sometimes the answers that users provided in the debriefing sessions were so overwhelmingly lopsided in favor of one solution that it was deemed unimportant to continue asking the same question. The union of all questions asked of all subjects appears at the end of this dissertation.

6.4 Characterizing the Subjects

In this section, I will not include the original graduate graphics class subjects in the statistics I give, nor will I include the subjects that were observed in the critical incident reporting (discussed later) or the children who were home schooled who gave us feedback as well. The table below summarizes the population that experienced the tutorial-1 observation sessions, with a 100-member sub-sample where I've thrown out the two oldest and two youngest subjects as outliers.

	Total		Without Outliers
Number of Subjects	104		100
Ages	High: Low: Mean: Standard Deviation:	71 15 27 14.36	41 18 22 7.36
Sex	Female Male	58% 42%	58% 42%
Computer Experience	90% self-described as using email, some www browser, some word processor, no programming.		

6.5 Other Subjects

In addition to the two primary populations, we made several other observations of users interacting with Alice, through what is known colloquially as "friends and family" usability testing. Informally, we would use subjects who are convenient and available at the time but who may or may not have been formally solicited for user observation. These subjects are often tested alone, not in pairs, or may have a closer, one-to-one contact with

the observer, not the more detached observer attitude present in the more formal two-person sessions described above. There were dozens of such observations made over the Alice development cycle, each providing food-for-thought if not exactly hard data on which to act decisively. So long as one avoids the temptation to make rash design changes in the face of one or two of these informal sessions, the data gathered in this way can help inform future decisions in a helpful way, and can keep the software designers on the project appropriately sensitive to the needs of the target user. A programmer who never comes out of his or her cubicle will never create easy-to-learn software.

6.6 Lynchburg Public Schools

In addition to these serendipitous observation sessions, Alice has been deployed at Lynchburg Public High School in Lynchburg, Virginia, as part of an experimental program for exceptional students. The high point of this interaction was a day-long field trip in which I observed eight high school students[1] learning Alice and subsequently interacting with Alice to build their own virtual worlds, an observation session which provided valuable informal feedback about the usability of the system.

This rare interaction came about thanks to the industrious and energetic support of several teachers and faculty members within that school, and for their efforts Alice has

1 High school students are somewhat younger than our target audience, but these students were exceptional students, and highly motivated, and we were interested in pushing the limits of the system. None of these eight users have been included in my usability testing figures.

improved immensely. Our first encounter with the Lynchburg school system was at one of our monthly open house presentations. It was there that several visiting teachers from Lynchburg were convinced that deploying Alice in their up-and-coming public computer lab would be an interesting experiment. To that end, we met with these teachers several times and ultimately invited them out to our lab to show them Alice in a hands-on way, and to teach them enough about Alice so that they could return to their classes with a curriculum design of their own.

In the course of showing the teachers Alice, it became clear that the “out of the box” experience with Alice was far too painful for inexperienced programmers. Even though users could instantiate objects through the menus, and move objects with the mouse, Alice had no facility for capturing the state of the resulting scene. The work was “lost” unless the programmer explicitly captured the state of the world by querying the positions and orientation of every object, and using the resulting numbers in their scripts.

The gulf of experience between the programmers of the UIgroup and the non-programmers of the Lynchburg school teachers had clearly resulted in a different set of expectations of how easy a programming environment should be. We had been perfectly happy to move objects around with the mouse then to capture the positions and orientations of objects to generate the necessary information for the script. The teachers found this state of affairs unacceptable, or at least unteachable. It took the shock of demonstrating the system to new users to make us realize something that now seems obvious to us in

hindsight: Alice's workflow for the first 3 years of her existence was not geared toward the novice. The result of this interaction was the Opening Shot and Script abstractions that were outlined in section 3.1.

6.7 Home Schooled Students

Finally, and in some senses, most remarkably, Alice is being used in a home-schooling environment in at least one place that we know of. The Sendroff family is using Alice to teach programming and critical thinking skills to their two daughters, Cassi (age 11) and Eireann (age 13). We remain in close email contact with the Sendroffs and provide them with frequent updates of the software. In return, the Sendroffs have sent us several of the scripts that they have written to show us the sorts of worlds they have worked on, along with insightful email telling us what they found most difficult about the various projects they have undertaken. This exchange has been extremely valuable as well as sobering to the Alice design team, as none of us would have thought back in 1992 that interactive 3D graphics programming could be made even remotely accessible to 11 year old school children, regardless of how gifted the student may be. As anecdotal as this evidence may be, I consider Cassi's scripts among the most impressive and most surprising outcomes of the Alice research effort.

6.8 Critical Incident Reporting

Usability testing like this is useful for seeing the grave errors either in the tutorial or

in the GUI; the sorts of "show stoppers" that would so frustrate a new user as to cause them to abandon the tool completely. In the testing of an API, one also needs a way of delving deeper, a way of finding out what sorts of problems crop up after the first hour of system usage.

Toward that end, I employed a technique known as Critical Incident reporting [Fitts]. Originally used in the design testing of World War II fighter cockpits, this technique is still used widely in helping to understand what goes through the mind of an expert or near-expert as they exercise their expertise in their given domain.

Critical incidents are not necessarily bugs. A critical incident may be a commonplace, everyday event or interaction, but it is critical in that it stands out for the person involved. An incident is critical if it was problematic, confusing, highly typical or exemplary, a great success, or a significant failure.

A description of a critical incident generally includes some sort of context (what sort of goal the subject was trying to accomplish), as much detail of what happened as the subject can remember (what strategies were tried, what worked and didn't work), and reports of what the subject was thinking and feeling at the time and just after the incident

There were four people I drew upon for critical incident reports, three of whom were engineers, and one was Cassi, the aforementioned home schooled girl. Each had his or her own self-selected task, and each experienced significant problems in the course of

implementing their ideas. I treated their observations as valuable post-mortems which provided insight into the more serious Alice design flaws. These issues will be covered later in the section on future work.

6.9 The Worst and The Best

The last part of an Alice observation session was a 30 minute debriefing session and interview. While some of the questions asked during this session changed over the course of the 104 user trials, two questions remained constant: "what was your favorite thing about Alice" and "what was your least favorite thing about Alice?" Asking users to clearly identify things they don't like, (and forcing them to answer[1]) can be a valuable way of gathering the all important "walkaway" impression that one's software leaves after the fun is over. Asking people what part they liked best is less useful, though still informative in its own way.

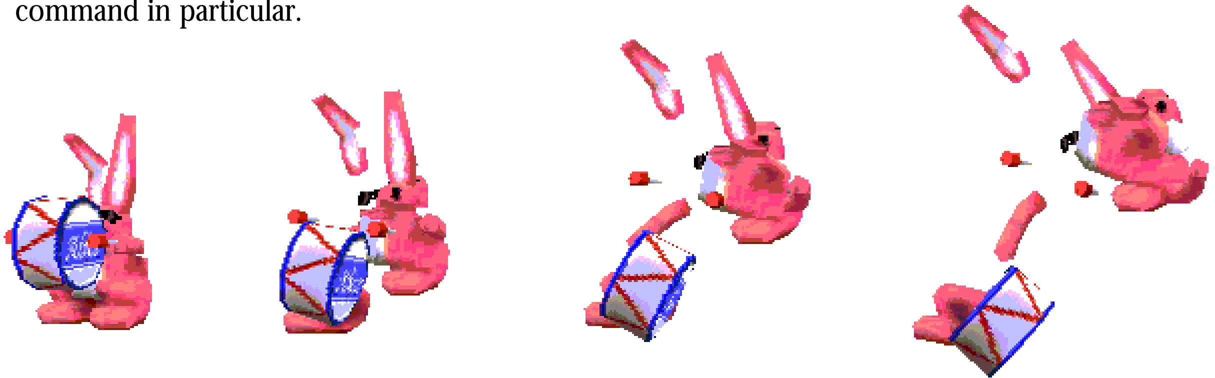
The lists below represent the percentage of people who mentioned a given topic as being one of their 3 most or least favorite things about Alice. In some cases the totals are greater than 100% because out of the starting pool of 90 people, some people chose to identify "best" things but not "worst" things and vice versa.

1 It is sometimes difficult to get people to be sufficiently critical as it is sometimes seen by users as impolite. It is vital that observers not acquiesce on this matter, even though one's own ego may be inclined to take "oh, no, I liked it all" as a legitimate answer. This is a subtle twist on Nathaniel Borenstien's concept of "egoless programming" [Borenstein].

The Best

▪ Destroying the Bunny	80%
▪ Sounds	30%
▪ Getting Objects to Obey Commands	10%
▪ PointAt	10%

By far the most popular aspect of the Alice tutorials was the chapter where users learned about the destroy command. Destroying objects, like most things in Alice, is animated. Animated object destruction entails pulling the parts off of the object, tumbling them away from the center of the blast[1] while gradually reducing their opacity to 0.0 (invisible) over the duration of the animation (default of one second). While this happens, Alice takes a whimsical turn and plays a random "death sound" from a collection of gurgling cries and groans[2]. Sounds in general came in a distant second place, followed by a general appreciation for the notion of commanding 3D objects and a fondness for the PointAt command in particular.



---

1 The rate of tumbling and the speed of movement is proportional to the size of the part's bounding box; smaller pieces move faster than large ones. For many objects, this gives a nice distribution of speeds, which can enhance the aesthetic effect of destruction.

2 The noises come courtesy of ID Software, makers of the 3D game Quake. Used with permission.

The Worst

▪ Typing	63%
▪ Syntax / Remembering Order of Parameters	45%
▪ Destroying the Bunny	25%
▪ Finding Lost Objects	12%

By far the worst part of Alice that people reported from the tutorial observation was typing. People simply hated typing. There are alternatives to typewritten script but our research agenda was to push the limits of the textual paradigm, given the problems with some of the other programming methodologies. I discuss the strengths and weaknesses of alternate programming paradigms in section 7.7.

Closely related to this, but mentioned separately by many, was difficulty with syntax. People who complained about this struggled with the textual patterns of programming in Python. As experienced programmers, it may be hard for us to remember that there was once a day when we first saw a list of parameters being passed to a function, and that those parameters were separated by commas. To novices, this was as much a part of their learning curve as the 3D commands themselves, and in fact was often the dominant source of difficulty in the Alice tutorial exercises. Said one user "the 3D stuff seemed pretty easy, but I

Many users also complained about not being able to remember the order in which the parameters appeared in Alice commands. Move(forward, 1) and move(1, Forward) were typical sorts of mistakes. I believe runtime tools can be most effective in helping users

through this problem. I have built a few prototype GUI tools to explore the efficacy of form-based programming, but as yet these prototypes have yet to be fully extended and incorporated into Alice, to be tested with members of our target audience. Until this happens, the best we can do through the system design is to make sure that the Alice Quick Reference card and copious example programs are easy to get to.

That destruction made the list is interesting given that it was also on the list of things people liked most about Alice. This is possible only because some subjects who had an opinion about the best thing did not have an opinion about the worst thing – no subject mentioned it as both the best and worst thing in Alice.

Of the people who reported that Destroy was their least favorite part of the experience, 75% of the respondents were female, all of whom mentioned that the destruction sounds were extremely violent sounding, which was made all the more disturbing by the fact that the operation was carried out on a small, pink bunny rabbit. In reaction to this, I have developed a somewhat friendlier, less chilling destruction protocol that scales objects down to 0.0, makes a whimsical *poofing* sound and replaces the object with a cloud. This new protocol has not been tested yet, but I have high hopes that it will be received well by both males and females.[1]

1 In a head-to-head contest between Quake death sounds and a poof noise, it is not hard to guess which would be most popular with young males.

Finally, users often found themselves losing objects. Either they would drag objects through the ground or off the screen, or would fly the camera off into space. We solved this problem through a series of GUI elements: Alice objects have a context menu (activated by the right-hand button on the mouse), on which appears a command for "Get a Good Look At." The camera controls themselves also have a reset button for when the camera is hopelessly lost in space.

6.10 Some Commonly Seen Problems

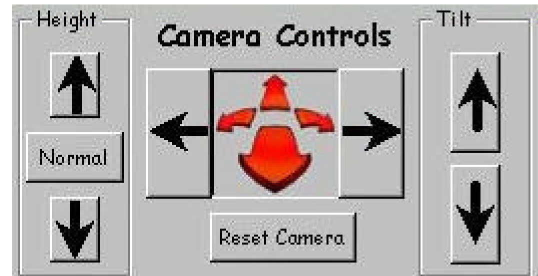
In the course of observing these 104 users, there were some common problems that I saw but were never reported in the "3 worst things" question. The compiled list of common problems appears below. This list is a good example of how user surveys are helpful, but no substitute for direct observation of users. Users often find it hard to fully articulate or clearly remember what it is they find hard about a system, to say nothing of the well-known phenomenon of subjects wanting to please the system designers by not saying anything derogatory.

Case Sensitivity

As mentioned in section 4.7, case rules were a huge problem. Few people mentioned it explicitly, often mentioning it in passing as a part of "the typing problem" but we saw so much difficulty with this particular part of Alice that we were driven to alter the programming language to fix it.

Mouse interactions and camera controls

This confused many people. Our first camera navigation interaction entailed mousing into the scene to fly, which invariably led to confusion with picking and manipulating objects, and in getting the camera into strange orientations and positions. Most everyone who tried to fly got lost. Even so, users generally found flying around a 3D world much more compelling than anything else that was happening in the tutorial, which, of course, is a pedagogical problem. We considered remedying this problem by incorporating flying into the lesson (*if you can't beat them…*) but we rejected this because the flying interaction, as it was designed at the time, was too hard to teach. Besides, flying was so much fun that everything else that appeared in the Alice tutorial we saw as boring by comparison. Never follow the animal act.



Instead, we redesigned the navigation controls by bringing the flight interaction out of the rendering window and into an explicit navigation vehicle attached to the lower edge of the rendering window, much in the style of navigation vehicles seen in VRML 2.0 worlds[1]. This helped break the ambiguity between mousing into the scene to fly and mousing into the scene to pick objects.

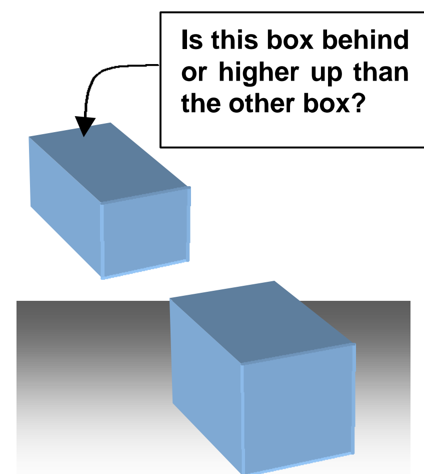
1 One important difference between the VRML 2.0 Vehicles and Alice's camera controls is that VRML vehicles are typically implemented as 3D objects, which makes them subject to lighting cues and occlusion with objects in the scene. Alice camera controls are never part of the simulation, being a 2D GUI control.

Motivation for scripting

Some users never quite understood why one would ever want to write a script in the first place. “Why can’t I just use the mouse?” was a common refrain. Our failure to motivate is a very serious problem, though one that might be fixable only through a more careful pedagogical treatment in the tutorials. The best pedagogical answer I have to the question of “why not just use the mouse” is that “you only have one hand, and the script is the way of getting the computer to do lots of things at once for you – more things than you can do with just the one mouse.” Further testing will bear out whether this approach is effective.

3D seen as Flat

There was a small, but significant percentage of people (5%), all of whom were female, who struggled to understand the Alice rendering window as a three-dimensional scene. For these users, Alice’s depth cues were too weak, which lead them to ask repeatedly how far back things were. Objects that were further in depth in the 3D world were often seen as floating above objects that were nearer, which implies that these users were not seeing depth in the render window at all, just a 2D image. All subjects thought that shadows might help, and for a few it did, but more study is needed to determine the nature of this interesting subset of the user community.



Confusing Motion and Scale

A few people (10%) confused the act of moving objects closer to the camera with making objects physically larger. This confusion came about regardless of whether the user moved objects via direct manipulation or with snippets of Alice script. Again, this group was entirely female, which may or may not be significant.

6.11 Summary

- Alice testing was performed on a total of 104 users.
- Two-person talk aloud protocol was used to test the API, the tutorial and the runtime programming environment.
- Alice is targeted for use by undergraduates, though we have significant anecdotal evidence that Alice can be used by exceptional students who are much younger than this.
- Alice is being used in the Lynchburg Virginia public school system as well as in some home-schooling settings.
- Destruction was listed as both the best and worst experience during the testing of the tutorial. Those who did not like the destruction protocol were largely female.
- Syntax, case sensitivity and other surface issues of the programming language were a significant impediment to the users we observed.
- The 3D display may not carry enough in the way of depth cues for some users. Designers should be aware that what seems 3D to some may not be seen as 3D by others.
- Typing is hard for novices.

Perfection is achieved, not when there is nothing left to add, but when there is nothing left to remove.
– Antoine de Saint-Exupery

Chapter 7
Traditional APIs for 3D

7.1 Introduction

This chapter describes the ways that traditional 3D APIs fail to support programmers, and then describes some of the alternative paradigms for specifying 3D object behavior

7.2 Non-intuitive semantics

Whether through oversight, poor design, or failure to hide an underlying abstraction, some 3D APIs can exhibit behavior that novices (and some experts) find surprising and unintuitive. For example: the scale command available in most 3D APIs changes not only the size of the object being scaled but the size of space as seen by that object (usually implemented as the multiplication of the object's modeling matrix by an appropriate scale matrix). This has the side effect that moving a scaled object in its own coordinate system

moves it by the scaled amount, not the requested amount. Alice used to have these same semantics (it is after all, easiest to build a system this way), but we long ago abandoned these semantics in the face of watching many users struggle to understand the very strange side effects that come from making an object larger or smaller. Perhaps experts tolerate this strange side-effect because they understand the implementation, whereas novices have no such background, and to them the side-effect seems strange and arbitrary.

7.3 Ignoring the Law of Least Astonishment

One design principle that holds across domains is *the law of least astonishment*, which suggests that a design ought to present the least surprising interface possible to its user. The designer should not confuse *least astonishment* with *most consistency*. Sometimes designing an unsurprising interface amounts to choosing a formal notation and being ruthlessly consistent in its application. This gives the API a high degree of predictability and tends to work best in highly artificial domains, such as network protocols. Other times, to be *unsurprising* means matching people's expectations, which are sometimes themselves inconsistent. I would argue that that in some cases a designer should strive to design an API that is as naturally inconsistent as people tend to be. This strategy works best when the user is likely to bring real-world intuition and prior experience to the problem domain that the API addresses, as is the case in describing the placement and motion of three dimensional objects in space over time.

One example of a 3D API element that violates the principle of least astonishment is

GL's rotation command. This command takes a single argument: an integer which is the number of tenths of a degree to rotate. No other command in the GL command set is given in terms of fractions of a natural unit, and in any case, it is unclear why tenths of a degree makes any more sense than whole degrees.

7.4 Weak Abstraction

This problem occurs in 3D APIs that do not fully insulate the programmer from the software and hardware implementations underlying the graphics library. Typical examples of this sort of API design flaw in 3D APIs:

- Exposing the 4x4 matrix representation of the underlying graphics, forcing the programmer to know matrix algebra to manipulate the graphics
- Exposing the polygon pipeline implemented in hardware, forcing programmers to manage the database themselves
- Failing to abstract away the rendering operation, forcing programmers to call draw() explicitly

Without effective abstractions, programmers spend precious cognitive resources on implementation details instead of application semantics.

7.5 Poor Separation Of Concerns

Many modern 3D APIs require programmers to create geometric objects directly in code. Programmers are not 3D graphic artists or model makers, and as such, this task should be lifted from them by having the API provide the ability to import finished CAD models in popular 3D file formats. Similarly, texture definitions for 3D objects should be left to artists,

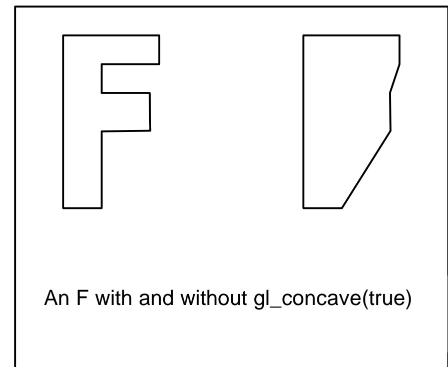
not programmers.

7.6 Poor Choice of Defaults

The GL libraries on SGI computers often set the default parameters of the rendering engine for maximum speed, even at the price of correctness of the graphics. This "go fast, be wrong" can be turned off, into "go slower, be right", but knowing where the problem is and how to fix it often requires knowing more about the programming system than a novice can easily tolerate, leading to confusion. For example, in the gl programming library, in order to render concave polygons correctly, the programmer must first execute the function:

```
gl_setconcave(TRUE)
```

This prepares the graphics hardware to use a slower but more flexible rendering algorithm that handles both concave and convex polygons. The default is to use a faster, but less general algorithm that assumes that all polygons will be convex. Attempting to render a concave polygon without calling gl_setconcave(TRUE) will result in a picture of a polygon that has "cut corners" or other strange anomolies.



An F with and without gl_concave(true)

A novice gl programmer therefore needs to know this strange implementation detail if he or she wants to render concave polygons. If the novice does not know about this function call, there is little hope of debugging a visual scene that happens to contain a

concave polygon. Ordinarily, when polygons are rendered incorrectly, the most likely place of failure is in the polygonal data itself, but gl opens up another possible failure mode, and one that can only be discovered through extensive reading of the gl programming manual. APIs should be designed so that their default behavior favors the programmer who isn't familiar with the entire API. As a real-world example, a student in our research group lost two full working days in hunting down a problem that was due to this usability flaw in the gl library.

7.7 Non-scripting Paradigms

At first it might appear odd that we make non-programmers use textual programming as the means for specifying the behavior of three dimensional graphical objects. Isn't there a better way for novices? Much has been written on the subject of programming by novices [diSessa][du Boulay][Green] with the conclusion that there is no "silver bullet" for making program writing a skill-free task, and Alice makes no claim to having solved this problem. This choice of scripting may seem doubly odd given that there seem to be non-scripting alternatives to programming for specifying the motions of objects on screen. Some well-known ones are:

Direct Manipulation (or keyframing) is a style of animation specification in which the user places objects into sequences of static positions using the mouse, recording the position and orientation at each step, then leaves the computer to interpolate between these positions during playback. The quality of the animation is dependent on the skillful choice of

positions (usually placed at the extremes of motion) and the details of the interpolation method used (manual, computer-assisted, completely automated).

Programming By Example (or motion capture) is an animation method in which the user starts the recording process, then drags the objects appropriately, during which the computer samples and records (with a timestamp) the object's position. During playback, the position information is fed back to the object at exactly the same rate that it was recorded. Motion capture is a special class of this kind of animation, and usually refers to methods that involve instrumenting a human actor's body with tracking technology that streams to a computer that records body part positions.

Constraint Based Programming controls object animation implicitly through declarative means by imposing constraints on the objects in the scene, usually based on metaphors found in the real wold (joints, hinges, springs, glue). The animation is initiated by providing some motive force: a simulated motor or crank in software or user interaction with the mouse. In this way, the user can achieve realistic motion without having to specify the logic behind the motion; it is often easier to specify the constraining relationships between objects and to let the system compute the ramifications.

7.8 Critique of Non-scripting Paradigms

In spite of the attractiveness of some of these alternate approaches, the Alice project has remained focused on the lexical/procedural paradigm for the specification of 3D object

behavior. The prime motivation for focussing on procedural animation comes from a desire to see how far the procedural paradigm could be pushed, but at least some part of the motivation was also due to the belief that the alternative paradigms listed above suffer from some significant drawbacks:

Logic : capturing the notion of an `if` statement is difficult in the direct manipulation and keyframing paradigms. This should not strike us as terribly surprising as this has a long-standing problem in the domain of graphical programming languages. Constraint-based animations on the other hand, have much (but not all) of their logic buried in the constraints themselves, or in the order in which the constraints are evaluated, which sometimes makes such systems hard to debug. For logic and choice-making that falls outside the domain of the constraint engine (if the package even allows this), constraint based systems suffer the same problem as direct manipulation and keyframing.

Interactivity : this is a special case of logic. Specifying what happens when the user clicks on an animating object is awkward using these techniques. Again, procedural programming makes specification of callbacks relatively straighforward.

Ease of Mastery / Quality of the Result: Keyframing even simple animations is a great deal harder than it appears at first; getting the proper visual effect is a skill that can take years to develop.

Editing : all three techniques are typically difficult to modify once animations have

been specified. Also, animations created by these methods are not straightforward to parameterize (making something go more slowly, or act on some other object.) Procedural progamming, in contrast, is comparatively easy to modify and parameterize.

Parallelism : keyframing and programming by demonstration both make it tricky to describe how two or more things are to happen at the same time. And because everything happens in parallel in constraint based systems, it can be hard to describe a strict sequence of events using constraints.

Size Efficiency : demonstration and keyframing systems typically require a great deal of information to specify motion. Scripting, in contrast, can be very compact, which is extremely important in distributed or networked environments. Pixar's classic computer generated short *Luxo Jr.* was several minutes of batch animation, but an Alice implementation of this animation can be specified in about 200K bytes, including all models, textures and animation specification data, a remarkably small size compared to the batch version.

Despite the drawbacks, it would be inappropriate to dismiss these approaches as unhelpful; each has its strengths for different classes of problems. Finding an easy way to integrate all these paradigms so that they work well together in a programming environment (one that is still easy to learn) is a topic for future research. Some systems already start to mix paradigms: the Disney Imagineering *Player* system (used in authoring the Aladdin VR attraction deployed at Epcot center in 1995-1996) and SGI's *Cosmoworlds* authoring system

both use uninterrupted animation snippets played out at “branch points” controlled through the **if** statements of a traditional text-based programming language.

7.9 Some Thoughts on Programming By Novices

Much of this work is concerned with the design of an application programmer’s interface (API). An API is a library of reusable commands that a programmer uses to help simplify the task of assembling a computer program. Alice is (in part) an API that supports abstractions and commands for defining the motions and reactions of 3D graphical objects.

It goes without saying that APIs, like graphical user interfaces (GUIs), are designed artifacts, but until now almost no work has gone into considering the process that goes into the design of APIs. Perhaps the lack of design guidelines has to do with the perceived size of the potential audience: many more people interact with GUIs than ever interact with an API. There will always be more users than programmers. Or will there?

One underlying assumption of this work is that programming is a more mundane task than it first appears, and that people who would never describe themselves as programmers might find themselves engaging in programmer like behavior. On the other hand, others have observed [Brooks75] that when it comes to programming, there is no silver bullet: “programming is hard.” Indeed, inventing an easy-to-use authoring tool will not totally obviate the need for programming. The specification and control of 3D object behavior is something that will still have to be specified by a sequence of scripted

commands, but there is ample evidence that there exists a level of programming that some people are willing and capable of engaging in. If the programming logic is straightforward, if the tools and programming environment are specific to the application domain, and if the applications are not terribly large in scope, people can and do engage in activities that are very much like programming, though they themselves might be surprised to hear themselves described as programmers. As evidence, I offer the following examples of novice programming:

Hypercard is widely used as an authoring tool by non-programmers in a wide variety of settings on the Macintosh platform. Tools were used here to obviate a lot of programming, and when users find that they need an “if” or “while” construct, they are highly motivated to learn the scripting language, but are not required to use the language before the need is apparent. This system exhibited a quality I refer to as the *controlled exposure to power,* which I discuss in the context of API design in section 13.1.

Smalltalk [Goldberg] was used by Alan Kay to show that school children were capable of learning object oriented programming if they were sufficiently motivated and the tools were sufficiently advanced.

Logo, [Papert] presented by Papert et. al. showed again that children were able to engage in more sophisticated programming than anyone thought possible. All that was necessary was a carefully-designed language that modeled a well-defined domain with powerful abstractions. Just as important as the language was Logo’s curriculum, which

revolved around the idea that programming was a powerful form of thinking that allowed children to explore notions like debugging and independent exploration.

Macros provided by most business software products can be "programmed by demonstration" by non-programmers who have goals other than programming in mind.

Spreadsheets represent constraint programming in a well-defined domain. Like macro users, spreadsheet users typically are motivated much more by their application program than they are by the programming task.

From these experiences, I would argue that programming by novices can be supported through three things:

Motivation: Users can accept writing programs ("scripting") if they perceive the need to do so, often driven by the demands of task-related need. This external motivation is not an all-or-nothing affair: Some students will understand simple keystroke capture (or macros) if it is seen as the natural way to avoid tedious sequences of steps. Not until those steps require repetition is there enough motivation to explore the notion of loops, and not until the script requires decision making will it be possible to effectively teach if statements.

Tool Support: People concentrate on the application task better if they have a powerful environment that helps ease them gradually into the task of scripting.

Safe Exploration: As shown by Logo, a programming environment that encourages exploration and asking "what if" is a powerful aid in getting non-programmers to behave like

programmers. To this end, an Undo facility is vital in being able to ask “what if” safely.

7.10 We are All Novices

As systems become more open and scripting is made more accessible in systems (e.g. HTML, Visual Basic for Applications, Javascript) more thought will need to go into the design of APIs. Just as there are occasional users and novice users of GUI-based programs, with attendant user interface design challenges, there are also occasional programmers whose needs pose unique challenges to API designers. Lest we think that carefully designed APIs are only for novice programmers, we should remember that even seasoned and experienced programmers are relative novices when faced with an unfamiliar programming library or some new language to learn.

Objects are unobservable. Only relationships among objects are observable
Marshall McLuhan

Chapter 8
Objects, Parts and Children

8.1 Introduction

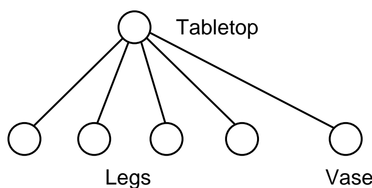
This section examines the way in which Alice's treatment of the traditional API functions and abstractions for 3D graphics differs from that of existing toolkits, and the justifications for breaking with this tradition.

Like many 3D graphics systems, Alice uses a tree-like structure of nodes and children to organize the objects in a 3D scene. Objects are connected to other objects via a "parent-child" relationship where the children nodes move and rotate in a coordinate system defined by their parents. When a node moves, it carries its children with it because the positions of the children are defined in terms of offsets from the parent node. These nested coordinate systems make it easier to think about object motion and thus make programming complex behaviors more convenient.

One shortcoming of this approach is that it is not terribly expressive: there is only one sort of relationship between nodes: the parent/child relation. This isn't even a "is-connected-to" relation: moving a parent moves the children but not vice-versa.

8.2 Children versus Parts

Take, for example, the case of a vase sitting on a table. The vase is an object and so is the table, where the table has four legs which are modeled as children of the table top. If we wanted the vase to move when the table moved, we might reasonably make the vase a child of the table, but there are hidden dangers: Conceptually, the vase really has a different relationship to the table than do the table's legs. We might characterize the vase's relation as an "is-resting-on" relation and the legs as having an "is-a-part-of" relation, but the classic PHIGS parent/child relation does not distinguish between these two important cases. From the point of view of the system, the vase looks just like one of the legs of the table.



Alice clears this confusion by marking some objects as "first class objects" (the table and the vase) and other objects as parts (the legs of the table). This allows Alice to perform operations on an object-wide basis in a way that models our common intuition for what constitutes an object. For example, we can now perform this operation:

```
Table.setColor(Red)
```

and have it change just the color of the table and its legs, not the vase. The rule is simple: operations that are meant to be object wide (SetTexture, SetColor, Destroy,etc) are applied to the named node and to all its *parts*, stopping the traversal at other first class objects. For greater control, operations that are allowed to effect entire object sub-trees are allowed to take an optional keyword parameter, HowMuch, which controls the scope of these operations and is allowed to take on one of the four following values: ObjectOnly, ObjectAndParts, ObjectAndChildren, and JustFirstClass.

Most other systems would either:

- apply these changes to all children, whether or not those children were logical parts of the object or were merely attached OR
- would force the programmer to manage by hand which objects needed to be changed and which didn't.

As it turns out, the concept of first-class object is a reformulation of an older idea from linguistic semantics: inalienable and alienable possession [Lyons]. Inalienable possession describes a relationship that a person has with something permanent or intrinsic like the relationship he has with his mother or the relationship he has with his own body parts. Alienable possession describes the relationship a person has with anything that he owns that can be lost or given away. One's arms are inalienable, one's hat is alienable. In the field of linguistics, alienable possession is an idea that is used to break the ambiguity that sometimes crops up in understanding the varied and subtle concepts of ownership. Alice uses the concept to tell where one object begins and another one ends.

8.3 Events and Parts

One can ask then what are the real distinctions between objects and parts? One real difference is in the way the two kinds of nodes react to user input. When a user clicks on an Alice object or part, the click event is dispatched to the first-class object that was clicked, even if the mouse event happened over one of the first-class object's parts. This event record contains the name of the part that was clicked so that the handler can do interesting things with the part should it so choose. For example, the user can click on a robot arm, the robot gets the event, and the robot code then causes the arm (specific part clicked which is found in the event record) to move.

The normal handler for Alice objects simply moves the first-class object (along with all its parts and any first-class children), but programmers can write their own handlers to do more interesting things if they wish. The fact that events are handled by objects and not by parts is one of the differences between objects and their parts.

This is not to say that Alice does not support the direct manipulation of parts, only that clicking into the rendering window will never send an event directly to a part, only to an object. To address an object's parts directly, users are allowed to click into the tree-view display. By specifying a specific node in the tree this way, we allow the user to bypass the ambiguity that can result from clicking into the 3D graphical scene to select parts.

For example, if a user clicks on a mannequin's toe, we cannot be sure exactly the scope of the click: did the user mean the toe, the foot, the leg or the mannequin as a whole?

Though the click gesture is ambiguous, we can break the ambiguity by looking for the ‘first-class’ flag in the hierarchy of objects as we walk up the object tree. Clicking on the toe always selects the mannequin. To get the toe, the user must specifically click the correct node in the tree-view.

Once the user has selected a part in the tree-view, Alice ***temporarily*** makes this object clickable with the mouse. As long as the highlight remains on the object tree, the part can be dragged with the mouse. Clicking anywhere else will de-select the part, making it invisible to mouse clicks again, until the user clicks in the tree-view again. While this might seem inconsistent with our model (parts to not respond to events, objects do), we added this feature as a way of answering user requests for being able to take objects apart with the mouse[1]. Giving users the ability to drag parts off of objects raises a very interesting question though: is it ever appropriate to automatically promote a part to first-class status as a result of a user dragging a part away?

8.4 Direct Manipulation and Automatic Re-parenting of Parts

I have run a preliminary pilot study to help answer the question of when it might be appropriate to automatically re-parent a part and promote it to first-class status. I gathered 10 users from our user community (not part of the original 104 users of the earlier studies), and presented them with a version of Alice that did automatic promotion of objects. The

---

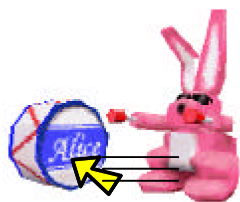
1. This behavior has since been removed from the latest version of Alice in the name of simplicity.

screen image presented to them was that of a pink bunny and a drum.

I asked each subject to grab the drum and to pull it from the bunny, leaving a clear gap between the two.

At this point, I asked each user to move the bunny with the mouse. For five of the subjects, I used an unaltered version of Alice, so that the drum remained part of the bunny. For the other five, the system was set up to automatically promote the drum to first-class status and to re-parent the drum to be a child of the scene.

Subjects were asked to subjectively evaluate the experience. Of the five who worked with an unaltered system, all thought that it was strange and unnatural that moving the bunny should also move the visibly disconnected drum. For the five that used the version



Pull the drum off

Then grab the bunny. Does the drum move too?

with automatic promotion, moving the bunny left the drum behind, and none of these subjects thought the experience was particularly surprising or remarkable. A more controlled experiment with larger numbers would be needed to firmly establish the existence of this effect, but given this early result, I would suggest that in the future, Alice should automatically promote and re-parent parts when they become disconnected from their

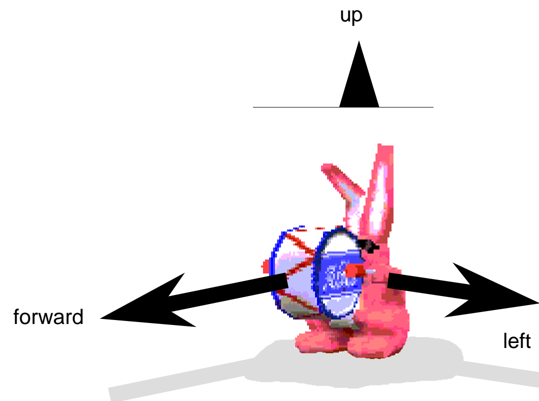
parent objects. For purposes of this direct manipulation operation, I would define "disconnected" as the state where the bounding volumes of the objects are no long visibly in contact with each other from the point of view of any of the cameras that are present in the virtual world. I believe that this behavior ought to manifest itself only through direct manipulation, never as a side-effect to a move command where the sub-parts of objects are made to move under program control.

8.5 Summary

- Alice distinguishes between first-class objects and their parts. Other systems have done this for pick correlation, but Alice uses this information for more than just picking, making the concept a powerful means of scope control for many operations including setting colors, setting textures, reacting to user input and the resizing or deleting of objects.
- Object-wide operations like SetColor act on subtrees of objects, but do not traverse past first-class objects in the tree by default. This allows Alice to tell where one logical object ends and another begins.
- The scope of an object-wide operation can be controlled to affect only one node or to flood whole subtress by using the HowMuch keyword with one of the tokens: ObjectOnly, ObjectAndParts, ObjectAndChildren, or JustFirstClass.
- First class objects react to mouse events. Parts do not.
- The notion of a first class object has precedent in the field of linguistics, where it is called *alienable and inalienable possession.*
- Future versions of Alice may wish to automatically promote and re-parent object parts when users manually disassemble objects with the mouse.

Quick…which way is X?

Chapter 9
The Death of XYZ



9.1 Logo Redux: Forward, Left, Up

Perhaps Alice's most distinguishing API feature is that it allows programmers to create behavior for three-dimensional objects without using the traditional mathematical names for the coordinate axes: X, Y and Z. Instead, Alice uses a more Logo-like [Papert] formulation of Forward, Left, and Up. We made this design decision after we observed many users, even advanced users, engaging in a tiny bit of explicit conversion every time they wished to move an object. This conversion is best summed up in a prototypical conversation:

> *"I want to move the truck forward one unit, and that's positive X to Alice, so I will type* `Move(X_AXIS, 1)`*."*

It became clear that our users already had a vocabulary for moving objects in space

and that the Alice system wasn't using it[1]. This seemingly tiny, cosmetic change to the API turns out to be one of Alice's biggest improvements to the API. By using direction names in lieu of XYZ, we relieve the programmer from a tiny, but extremely common cognitive mapping step, a savings that can be multiplied perhaps thousands of times in the course of developing a 3D program.

I also note that removing XYZ reduced the need to talk about negative numbers to an audience that naturally shies away from mathematics.

The six direction names in Alice are object-centric by default: moving an object Forward always moves the object in a direction that is forward from the object's point of view, not from the viewer's. As we will see in section 9.8, Alice has the ability to perform these operations from any other object's vantage point (i.e. Alice programs can do coordinate transformations, we simply do not call them that).

While it is true that some objects do not have an intrinsic forward direction, there are many objects that do. By contrast, there are *no objects that have an intrinsic X direction.* In the case of XYZ, the mapping from direction names to an XYZ label is *always* ambiguous and arbitrary. With direction names, the mapping is only ambiguous for some objects, and then only in some directions. In the worst case scenario, an object will have no intrinsic

---

1 For many years, Alice used the traditional X, Y and Z directions just like other 3D APIs. We switched to using Logo-like directions in late 1995.

directions, in which case the situation is as bad as XYZ. The psychology and linguistic communities have long been interested in the ambiguities that can result from the language of space. I outline some of that research in section 9.10.

9.2 Length, Depth, Width

The axis names Forward/Back, Left/Right, Up/Down also suggest that there are other names that we can use to describe attributes of an object. One such quantity is the size of an object, specifically, the dimensions of its bounding box. Our first attempt at capturing this quantity in the API was suggested by the psychology literature [Miller], and appeared as the first parameter to the Resize command, a call which is capable of changing the size of an object:

```
obj.Resize (Dimension_to_resize, scaling_factor)
```

As in

```
obj.Resize (Height, 0.5)        # make half as tall
```

Where dimension_to_resize is one of :

```
All                       # This is the default, results in uniform scaling
Depth
Width
Height
```

It was hoped that the terms Depth, Width and Height would be sufficiently intuitive that users would not have difficulty in telling these terms apart in some of the most common

cases. We also realized that while there were times that these terms were ambiguous, we reasoned that like Forward, Left, and Up, the ambiguities could be overcome via runtime tools to query an object's dimensions and in any case, was no more arbitrary than XYZ in the worst case. Clearly, like mapping X to Forward, Y to Right, and Z to Up, we have the luxury of mapping these three terms to whichever dimension we liked. The linear extent of an object from its front face to its back face was taken to be Depth. Extent left-to-right was Width, and extent top-to-bottom was Height. This mapping, like the Front, Left, Up mapping, seemed simple to explain, if a bit draconian. Unfortunately, this formulation fell apart in practice, as Depth, Width and Height were seen too often as coming from the camera's point of view: depth was the distance into the scene, height was vertical screen distance and width was horizontal screen distance. Though this is an informal observation, we decided that the problem was severe enough that it warranted attention[1]. We removed the Depth, Width and Height monikers and replaced them with the more artificial terms: FrontToBack, LeftToRight, TopToBottom. These terms have the advantage that they are derived from the direction names (or simple variants of them – Front/Forward) that the user already knows from the move command. The terms are also self-defining and do not suggest the ambiguous "camera point of view" like Depth, Width and Height do.

1 This is another example of an informal observation giving rise to a design change.

9.3 Choice of Names: Translate Considered Harmful

The first rule of good interface design is to make sure that you are speaking the user's language. This often means making sure that you stay away from jargon, and stay close to the natural terms used by your target audience. Cognition literature describes a strategy of label following where users often form a search strategy of looking for terms in the user interface (labels) that exactly match the words they have in their heads. When the interface labels don't match mental terms, performance in discovery of unfamiliar GUIs drops dramatically [Franzke]. This same principle can be extended to the design of programmer's APIs as well, especially when the API in question might suggest "labels" to users of that API.

In that spirit, I suggest that the word *translate* is a distinctly terrible term to use to describe linear motion in space of 3D objects. Translation is colloquially understood[1] to be the process by which (for example) French is converted into German, even though to the mathematically inclined, this term is also used to describe a particular kind of non-rotational affine transformation. We therefore deliberately designed the Alice API to use the word `Move` instead of `Translate`, hoping that this would be one part of the API that would not cause usability problems. Our users, as usual, were more resourceful than we thought.

---

1 The American Heritage Dictionary lists the mathematical definition of the word translate (non-rotational displacemet) *seventh* out of nine possible meanings, coming in just after (implying that is more obscure than?) the rarely-used Ecclesiastical meaning : "to transfer a bishop to another see." We clearly can do better than this.

Through user observation, we discovered a small, but non-trivial percentage of our user community who found the word move to be confusing (about 6%). The nature of the confusion is illustrated in the following reconstructed dialog with an actual student who was part of our Alice observation sessions:

Setting: the 3D scene contains a camera which points at a bunny. The user has seen the move command and has used it to push both the camera and objects around in the scene. The user has also seen the turn command and used it to cause objects (including the camera) to spin and pitch in space. As per the instructions in the tutorial, the user now types this:

```
camera.move (up, 10)
```

The camera rises 10 meters off the ground and the bunny disappears from sight. The camera is high in the air, pointing into blue sky.

Observer:	*So, knowing what you've seen from Alice so far, how would you get the camera to point at the bunny again?*
Subject:	*I would* **move** *the camera down. [*my emphasis*]*
Observer	*Can you gesture with your hands what sort of motion you would want the camera to make?*
Subject	*yes. [*Subject makes a turning motion with the hands, one appropriate to rotating the camera down toward the ground*]*
Observer	*so what command would you use?*
Subject	*move? 90 degrees?*

This, of course, is wrong. The user has taken the word move to mean something very

generic – turning is a kind of movement, as is (presumably) shaking, lumbering, and bouncing just to name a few. Moreover, the failure mode here is dramatic. Typing the command

```
Camera.Move (Down, 90)
```

Results in translating the object 90 meters down, which is almost certainly under the ground plane. Recovery from this sort of misunderstanding is very hard, and usually requires pressing the Undo button a few times.

By trying to steer clear of precise jargon, we seem to have erred in the other direction by choosing a word that is too vague. We can take some solace perhaps that this is not a pervasive problem, having been observed in about 6% of the users in our studies. That the number is this high is still surprising to me.

One possible way out of this dilemma is to design the API to be robust in the face of user input by allowing the word `move` to double as both rotation and translation, given that this is the way people seem to normally use the word. Taking this approach would require that we have some way of distinguishing between the desire to rotate and the desire to translate, which could possibly be done with a required keyword, as in:

```
Camera.Move  (Up, Distance=3)
Camera.Move  (Down, Angle=90)
```

Here, the keyword is used to dispatch to the correct implementation (either

translation or rotation).

This design has some appeal but is sufficiently different in spirit from the rest of the Alice API (no other command has *required* keywords, though nearly all have *optional* ones), that we have not taken this approach.

Another possible solution is to choose a word which is more precise, one that carries no other connotations than that of linear, non-rotational motion. We have been entertaining the idea of introducing the word **slide** into the API as a synonym[1], if not a replacement for the word move. Early tests of the word slide in the beginning tutorial showed that slide elicited none of the confusion associated with the word move. More testing may be needed before we make this design change.

9.4 Nudge

Nudge is a very powerful variant of move. Instead of moving objects by a distance measured in meters, objects are moved by a percentage of their own size. For example:

```
Bunny.Nudge(forward, 0.3)
```

Would move the Bunny in a forward direction (Bunny's forward) through a distance that is 0.3 of the bunny's FrontToBack dimension. Writing animations with Nudge instead of move makes them more robust by making them portable between objects of vastly different

1 Thanks goes to Ari Rapkin who first suggested this term.

sizes. A script that moves an aircraft carrier 100 meters will have to be rewritten by hand if it were to be applied to a Volkswagen, but might not require alteration if the move were re-cast as a percentage-based nudge.

9.5 Place

Place is a command used to move one object to a specific location with respect to another object, by referring to the intrinsic sides of the objects. This command is handy for scene construction and was directly inspired by the readings in the spatial understanding literature. Place takes the following form:

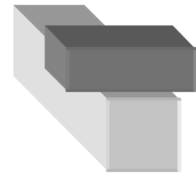
```
Obj.place(contact_condition, other_object)
```

Where contact condition is one of the following:

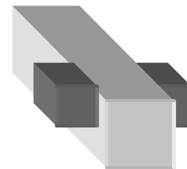
```
OnTopOf
OnBottomOf (or Beneath)
ToLeftOf
ToRightOf
InFrontOf
InBackOf
OnFloorOf
OnCeilingOf
```

For example:

```
Vase.place (OnTopOf, table)
PaddleFan.place (OnCeilingOf, Room)
```



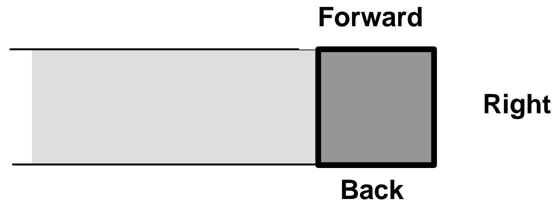
A OnTopOf B



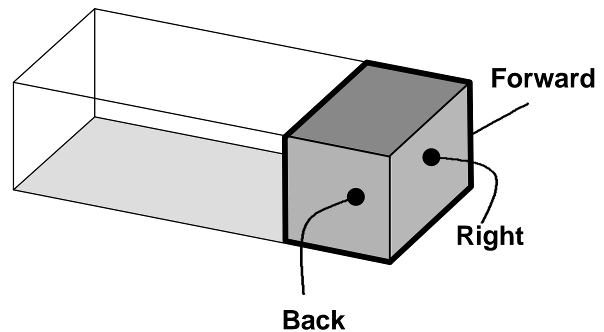
A OnCeilingOf B

The exact semantics of `ToLeftOf` is as follows (the other directions are analogous).

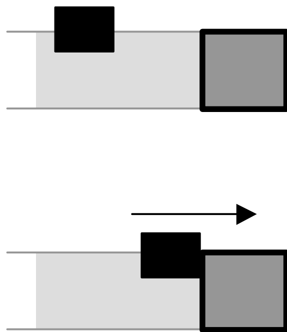
According to research by [Logan] anything in the 2D light grey area is considered to be "Left Of" the big box.



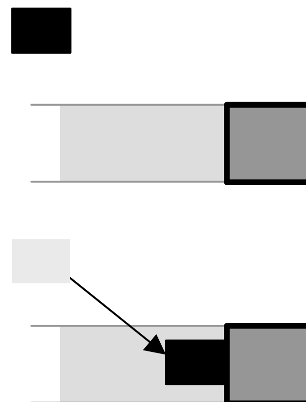
In Alice, we extend this idea into 3D by defining "Left Of" to be anything in the 3D volume shown.



Case One: Object is already to Left of the Target. Object is dropped in perpendicularly to the face of the object in question.



Case Two: Object is NOT to Left of the Target. Object is moved to the center of the face of the object in question.



9.6 Other Prepositions

English has more prepositions and prepositional phrases than those implemented in the Place command. While the list of prepositions is long, it is not infinite, and in fact is quite tractable, especially if one limits the set to those prepositions that have some sort of spatial connotation. A list of all (or at least, nearly all) English prepositions appears in appendix E. These terms might point the way to an interesting avenue of future Alice development. The semantics of the Place command were largely replicated in a system called *Put*, a concurrent and independent work done at Silicon Graphics and the University of California, Santa Cruz by researchers Clay and Wilhelms, as described in [Clay].

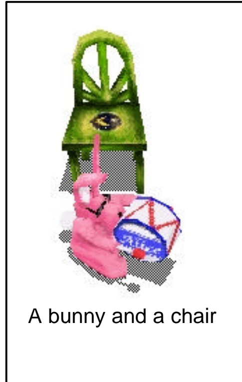
9.7 Move Toward/Move Away

Alice also supports commands that move objects in a more polar-like motion:

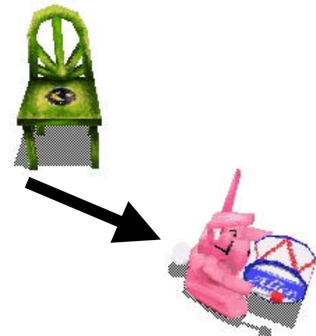
```
Obj.MoveToward(another_object, distance)
Obj.MoveAwayFrom(another_object, distance)
```

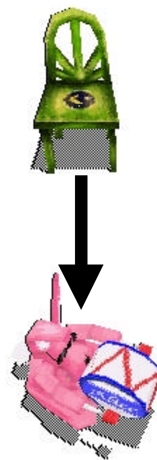
This allows objects to move in a radial fashion, using other objects in the scene as landmarks.

9.8 Any Object is a Coordinate System


A bunny and a chair

Coordinate transformations are common in 3D applications, and are usually done to make operations easier than they would be from a global coordinate system [Foley]. The Alice API already eases the burden of coordinate transformation to some extent by making the move and turn commands operate on the local coordinate system of objects, rather than from a global frame of reference. For more general coordinate system transfers, we designed a system [Gossweiler] that allowed programmers to perform any operation from any other object's coordinate system. This capability was not supplied in the form of an external library of matrix transformations, but through a much simpler interface: an extra parameter to every call that could meaningfully use it—a keyword denoting the object that should act as


Bunny.move (forward, 1)
Here the bunny moves forward AsSeenBy the bunny. This is the default action for move.


Bunny.move (forward, 1, AsSeenBy=chair)
Here the bunny moves forward in the chair's coordinate system. The term **forward** now refers to the chair's sense of forward, not the

the frame of reference for the operation.

In modern Alice, this functionality is exposed through the optional keyword parameter: AsSeenBy, as in:

```
Bunny.Move (Forward, 1, AsSeenBy=Chair)
```

Without this functionality, the burden of changing coordinate systems is placed on the back of the programmer, who must know trigonometry, matrix algebra, or both in order to effectively use it.

One significant benefit to novices is that it leaves the concept of “every object is a coordinate system” as the primary pedagogical challenge, as opposed to advanced mathematics. The mathematics is effectively hidden from the user behind an abstraction that users are already likely to employ: using other objects as a way of expressing the relations of objects in 3D space [Miller].

By making this commonly-used service available as an option to all geometric transformations, programmers are *encouraged* to use it, not penalized as they are in other systems. This shift in the cost of this function changes the way one thinks about programming 3D graphics: it encourages programmers to think of other objects as landmarks, a strategy that the psychology literature tells us is a common way of thinking about 3D relationships. As a side effect, the ubiquity of the AsSeenBy parameter severely diminishes the importance of the global coordinate system, as Scene becomes just another

object/reference frame that objects can move in.

I would argue that this feature is one that should be incorporated into future 3D APIs, whether they are targeted for novices or experts. We have been informed that this feature will appear in Direct 3D version 5.0, and has already been incorporated into the 3D API used by Disney Imagineers for the development of VR-based attractions.

9.9 Issues with Right, Up, and Forward

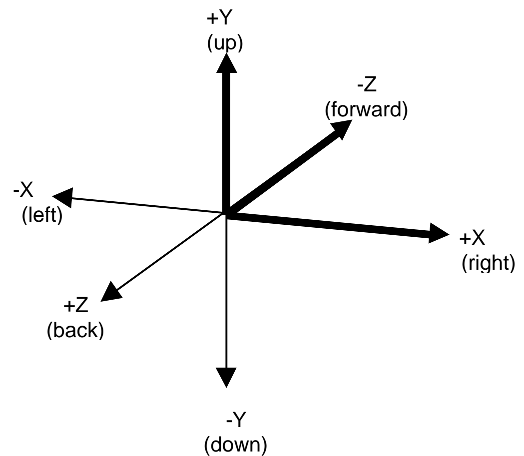
The notion that observers always see objects as having (at most) six sides is an argument put forth in [Miller] and would seem to be widely accepted in the spatial understanding research community. Of course, some objects have fewer than six sides, as summarized here:

	top / bottom	left / right	front / back
plane person car	Y	Y	Y
cup tree bottle	Y	N	N
mirror arrow carbon paper	N	N	Y
rocks balls boxes	N	N	N
very rare	N	Y	N
very rare	Y	Y	N
very rare	Y	N	Y
very rare	N	Y	Y

Note that the combinations listed as 'very rare' may in fact be impossible, but this analysis is conservative.

This lack of completeness would seem to pose a problem to a 3D API that would hope to exploit these as directional terms as the axis labels of an object-centric 3D coordinate system. How do you move a sphere forward when spheres don't have fronts?

For "non-orientable" objects like spheres, or semi-orientable objects like trees, the person building the 3D model arbitrarily assigns an axis to the missing orientations. For objects that are orientable along some direction, the model builder is required to know the Alice mapping that converts XYZ into direction names (see figure). Objects created without Alice in mind will, in general, not move or rotate correctly because they may not be oriented correctly. Alice, of course, has no deep semantic understanding of the objects it is manipulating: `obj.move(right, 1)` is always implemented as a translation in the positive X direction, which might not be the correct way to move, depending on how the model was created. To overcome this problem, the Alice system comes with a GUI-based tool that allows users to reorient their models in a canonical Alice orientation.



The Alice Direction Name Mapping
Note that positive Z is backward, not forward which is done in order to keep a right handed coordinate sytsem.

It has been surprising to see how many 3D models in the public domain are not oriented at all, along any axis, a fact that makes such models extremely hard to work with under any coordinate labeling scheme, even XYZ.

9.10 How Do Objects Get Their Sides? The Spatial Understanding Literature

There is a large corpus of literature on the subject of spatial understanding, spatial perception and the linguistic issues surrounding the language of space. My research has been influenced and informed by this research, and I believe there is fertile ground for further work looking for places where 3D graphics systems can be improved by paying closer attention to the issues raised by psychologists and linguists.

Unfortunately this literature is inconsistent in its terminology, which can make it difficult for an outsider to fully appreciate and follow developments in the field. For a good introduction and a survey map to the myriad terms used across (and within!) sub-disciplines, I suggest starting with Steven Levinson's excellent paper [Levinson].

A general theme emerges from this body of work: that there are many different coordinate systems that people employ in reasoning about space, in assigning direction names to objects and in thinking about the spatial relations of themselves to their 3D environment. The published literature presents many competing theories and formulations, and it is difficult to judge their relative predictive powers, and so, taking a conservative approach, I have decided to outline a fairly well-accepted analysis of how observers and speakers assign direction names to objects. In this analysis, humans use three common coordinate systems to organize their thinking about space: intrinsic coordinate systems,

viewer-centered coordinate systems, and absolute coordinate systems.[1]

Intrinsic Coordinate Systems

This coordinate system is derived from the inherent features or facets of an object. Of course, the term *inherent features* of an object masks a fairly sophisticated and language/culture-dependent combination of memorization, convention, and algorithm by which the tops, bottom, fronts, etc. are assigned to an object. In English, as we shall see, this assignment is usually functional in nature, but for some cultures it is far more influenced by object shape while in others the assignment is based on comparisons of the object to the parts of the human body [Levinson]. No matter the language, there seems to be a common (though not universal) mechanism to assign direction names to objects for the purposes of using objects as landmarks by which one can locate other objects or other speakers/viewers.

In English, the assignment of direction names is largely based on the functions of the parts of the object. Miller and Johnson-Laird proposed that objects are assigned faces according to the following schema: [Miller]

---

1 As Levinson points out, these words sometimes appear under different terms, depending on the academic discipline doing the research. There are other, nearly identical contrastive pairs that are sometimes seen: relative/absolute, egocentric/allocentric, orientation-bound/orientation-free, and deictic/intrinsic, from fields ranging from vision research, developmental psychology, philosophy, brain sciences and psycholinguistics. I prefer viewer-centered/object-centered because it seems to be less jargony than the alternatives.

(A) Is there a side of the object which is usually uppermost? If so, that side is assigned Top and the opposite side is assigned Bottom. If not, go to (B).

  (1) Is there a side of the object that contains perceptual apparatus (a face, eyes, mouth, etc.). If so, assign this face as the Front, and assign all other sides by analogy to the human body. If not, go to (A2)

  doll, person, dog

  (2) Do people take a characteristic orientation with respect to the object? If so, go to (a), otherwise go to (b)

    (a) Is this orientation inside the object? If so, assign sides by analogy to the human body.

    Car, chair

    (b) Is this orienation outside the object, facing one of its sides? If so, assign sides according to a viewer-centric coordinate system.

    Desk, mirror

  (3) Note that this object has at most two sides

  Table, vase, tree

(B) Is there a side lying in a characteristic direction of motion? If so, that side is assigned Front, and the opposite side is the back. If not, go to (C).

Arrow, bullet

(C) Note that this object is not orientable, but has at most six sides.

Block, cube, ball

Of course, this algorithm is no guarantee against ambiguity, but this is to be expected because even among people there is uncertainty about the fronts of objects. Which face is the front of a church? It depends. The point isn't that there is a fool-proof recipe for assigning faces to objects, but that we have a mechanism that at least structures the assignment process. Indeed, the Alice modeling team has made hundreds of 3D models for inclusion in the Alice distribution, each of which needed to be oriented front-to-back, left-

to-right, top-to-bottom. The modelers were never at a loss to assign these directions, nor did they consult the above algorithm directly. The linguistic research suggests that we each follow the above algorithm to some extent when trying to assign faces and beyond that, object sides are memorized on a case-by-case basis.

Viewer-Centered Coordinate Systems

The above algorithm suggests that some objects are assigned faces in a "viewer centered[1]" way, meaning that the sides of the object are not fixed with respect to the object but are imposed on the object from the point of view of the speaker or viewer. To tell someone to place a chair "to the left of the tree" almost certainly means either to the speaker's left, or the listener's left, but not from any sense of "left" that is attributed to the tree[2]. This sort of coordinate system is called "viewer centered" and can make some form of communication more ambiguous.

To make matters worse, objects that lack an intrinsic dimension can often *acquire* those dimensions when placed in the context of some other object. A table by itself may not have a front, but when placed against a wall, it passes the test in (2b), and thus acquires a

---

1 The linguistic literature sometimes refers to this as a deictic (pronounced dyek-tick) coordinate system.

2 The sort of language that supports a viewer-centric coordinate system is not nearly as universal as it was once thought. As strange as it may seem, linguists have discovered some languages that have no way of expressing the (horizontal) location of an object with respect to a non-orientable object like a tree or a rock (as in, "the rock is to the left of the tree"). Languages that lack viewer-centered coordinate systems in their lexical structure often employ both absolute and intrinsic, though Levison notes that viewer-centered coordinate systems seems to require the presence of the intrinsic system [Levison].

front. Levinson refers to these sides as “derived intrinsic” or pseudointrinic.

Absolute Coordinate Systems

Finally, there is the Absolute system. This coordinate system requires the presence of an external, immobile coordinate system such as compass points or a global XYZ coordinate frame. “Put the ball to the north of the chair” typifies this style of object location.

Coordinate Systems and the Alice API

It is not entirely obvious what we can learn as API designers from this line of spatial/linguistic research. Perhaps we should classify objects as being orientable or non-orientable so that when we call

```
Obj.Move (Left, 1)
```

the object either moves to its own left if it has one or to the viewer’s left if it doesn’t. This would seem to make the API more natural while also making it somewhat less predictable. What if there are multiple cameras in the scene? Should cameras (abstractly) be the objects delivering the Alice script to the interpreter, so that the notion of “speaker” is made concrete and explicit? This would seem to make some sense, but this is a line that we never crossed with Alice[1]. Instead, we chose repeatability over intuition and enforced the

1 Though I suggest it might make an interesting exploratory prototype, and one that is not all that hard to implement, given the Alice architecture.

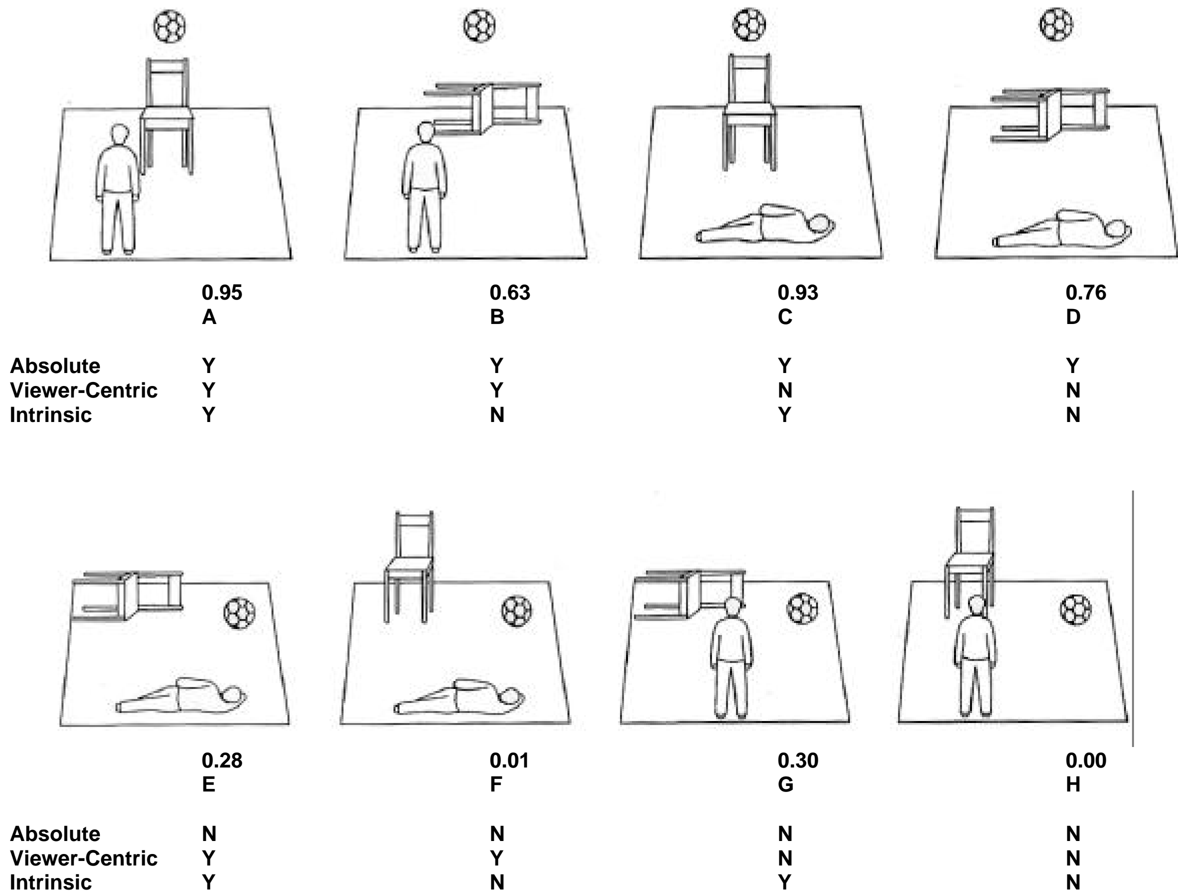
draconian but very simple rule: all objects have six sides, sometimes those sides are obvious and sometimes they are not. This makes the behavior of objects repeatable and tractable, which, for programmatic scripting, seemed to be more a important criterion than "naturalness."[1]

When Coordinate Systems Conflict

One variable that affects an observer's ability to correctly identify an object's intrinsic faces (an important ability when creating Alice scripts) is the orientation of the object and of the viewer, both with respect to each other and with respect to the ground. When objects, viewers and ground are oriented in different directions, the coordinate systems can conflict with one another, and this can cause one coordinate system to gain dominance over the others, an effect referred to as *alignment* and *preemption* [Levelt96]. This effect is most clearly seen when talking about the vertical (Up/Down) direction and was demonstrated dramatically in experiments conducted by Carlson-Radvansky [Carlson].

---

1 For example, I might not make the same design decision if the central activity in the Alice system were that of a kiosk-like, one-command-at-a-time voice driven interaction, instead of reusable script-building.



The illustration, taken from [Levelt], shows the results of experiments that were done by Carlson-Radnavsky and Irwin in 1993. In these trials, subjects were asked to stand or lie in the positions shown in the illustration and asked to name the relation between the ball and the chair (the actual trials used a variety of objects and a variety of backgrounds so as to eliminate confounds). Below each case is a yes/no indicator that tells whether the ball is in fact "above" the chair as seen from the given coordinate system. It is possible for the ball to be above the chair in one, two, or all three of the coordinate systems, leading to 8 total possible cases.

The numbers under each case indicate the percentage of people who thought that it was reasonable to say that the ball was "above the chair." Matching these responses against the yes/no criteria, we can see which coordinate systems interfere with each other under what conditions.

The results confirm that the three coordinate systems each use a different mechanism to determine the direction of "Up" for that system: the intrinsic system uses the Up/Down dimension of the object, the view-centered system uses the viewer's retinal meridian (orientation of the eye) and the absolute system uses gravity.

In the best case scenario, the ball is placed in such a way that it is "over the chair" in all three coordinate systems (case A), and subjects correctly report that this is so 95% of the time. In the worst case scenario, the coordinate systems do not agree (cases B, C, D) but so long as the ball stays over the chair in the absolute frame, subjects still report that the ball is over the chair no less than 63% of the time. It would seem that the absolute coordinate system is the one that most forcefully controls our perception of Up with the speculative explanation that this is due to our constant awareness of gravity. [1]

The fact that a subject's sense of Up is powerfully influenced by the Absolute coordinate system (gravity) and not the object intrinsic system seems to hint that the Alice

1. It is interesting to note that response times follow this pattern too: people answer these questions more quickly when the percentages are higher, probably indicating more confidence in the correctness of their answer.

system, which uses an object-intrinsic system by default, might need a slight re-design to make the word Up refer to a "skyward" direction, regardless of the object being manipulated. The future work section (page 214) describes in detail a modest redesign of the Alice directional system that turns the direction names into explicit properties of objects, which should help make explicit the difference between object.up and scene.up.

But what of the other directions? The Carlson-Radnavsky and Irwin experiments only explored human perception of the Up direction, not the Left/Right or Forward/Back directions, neither of which benefits from a gravity-like force that suggests a preferred direction. [Levinson] describes something called the Principle of Canonical Orientation that suggests under which conditions an object might use its intrinsic directions and when it will derive direction names from its environment.

More work needs to be done to determine how, if at all, Levinson's findings should be folded into future interactive 3D graphics scripting systems.

9.11 Summary

- Objects in Alice move in one of six directions: Forward, Back, Left, Right, Up, and Down. This removes a common cognitive step in the specification of 3D motion.
- Alice can make any object move in any other coordinate system through the use of the AsSeenBy parameter. This design is one of Alice's most important features as it relieves the programmer from needing to know a lot of linear algebra.
- This scheme is less ambiguous than XYZ, if only because some objects have natural fronts whereas no object has a natural X direction.
- The orthogonal dimensions of an object are named FrontToBack, LeftToRight, TopToBottom, names chosen to avoid the ambiguity of Width, Depth and Height.
- For novices, the word translate is a weaker term than move, which in turn may be weaker than slide.
- A small percentage of novice users freely used the word move to mean both translation and rotation.
- Nudge is an Alice command that moves objects through distances that are expressed not in linear distances but in terms of percentages of an object's size. Expressing movement in this way makes it easier to re-use an animation on an object of a different size.
- Place is used for scene construction and allows programmers to make one object abut another. There are other prepositions in English that might make interesting extensions to Place.
- Commercial 3D libraries are using the AsSeenBy idea in their 3D libraries, inspired directly by the Alice system.
- The Forward, Left, Up formulation has some shortcomings of its own, due mostly to the viewer centric coordinate system that humans commonly use in the presence of objects with fewer than 6 canonical sides.
- These shortcomings have been explored to some degree in the spatial understanding literature. More human perception and cognition research is needed in this area in order to know the implications for future 3D API design.

It was an awful, dirty, ugly thing to do,but it did get the speed up.
I do know that it confused almost everyone who tried to use the rotate command for the first time…

Anonymous Silicon Graphics Engineer,
commenting on the decision to have the
gl_rotate command take tenths
of a degree instead of degrees

Chapter 10
Rotation

10.1 Introduction

As described in Chapter 8, The Alice API depends heavily on direction names to structure the way users think about 3D objects. Where traditional APIs specify rotation with two parameters: **amount** (in degrees) and the **axis** around which to turn, we realized that talking about axis names explicitly broke with the Logo-ish feel that we had tried to achieve. Unfortunately, our decision to use direction names (forward, back, left, right, up, down) forced us into ambiguity issues that we had to resolve. This is the story of how we broke the ambiguity.

Rotation may be the hardest common function to encapsulate well in an easy-to-

learn and easy-to-use API. After several years of development, I would suggest that the rotation commands in Alice remain some of the most error-prone primitives in the system. Below I outline several of the reasons why this function, more than the others, seems to defy simplification. There are several reasons why this might be so:

No Consensus of Terminology

Rotation seems to have a large number of synonyms in English with no single term serving the purpose of describing most forms of rotation. When users were asked to supply a term to describe the motion of an object being turned around as if on a turntable, responses included: revolve, spin, twirl, corkscrew, whirl, and orbit.

Gestures

Others have observed that when users describe rotation, they often simultaneously gesture with their hands [Wexelblat], even when such visual displays aren't helpful to the listener (e.g. over the phone). Moreover, users rely on a common and finite set of hand gestures to communicate angles and directions of rotation. This has implications for the design of a text-only API. When users rely on gestures accompanying speech, it allows both the speech and the gesture to be less precise. In a text-only API, users are left only with the textual component to describe rotational behavior which leads to a degraded signal; gesture so often carries much of the signal.

Object-Specific constraints

Another reason that we might find it difficult to capture highly generalizable rotational abstractions in an API is that objects in the real world rarely undergo completely unconstrained rotation. On those rare occasions when they do, we often refer to this sort of motion simply as “tumbling” without regard to rates or amounts of rotation over any given axis. In English, we often use words like “open” and “close” to describe rotations, when it is understood that the object undergoing rotation is constrained by a hinge or a pivot of some kind.

- **open** the door
- **lift** your arms
- **spin** the roulette wheel
- **turn** the steering wheel

If nothing else, the effect of gravity and horizontal supporting surfaces like floors and tables tend to constrain rotations to a single plane, which reduces an imprecise term like “turn” into a two-degree of freedom task (left or right), down from six degrees of freedom, which is the mathematically general case that 3D APIs most often try to capture.

10.2 The Alice Rotation Commands

In Alice, rotation happens through two commands: Turn and Roll. Turn handles rotation around the Up/Down and Left/Right axes (turning Left/Right and Forward/Back, respectively) while Roll handles rotation around the Forward/Back axis (allowing rolling Left/Right). In the sections that follow, I will lay out the rationale for supplying two methods

for rotation. First, though, it is helpful to look into these commands more closely, starting with the units we have chosen to specify rotation amounts.

10.3 Units of Rotation

For most of Alice's development, we used degrees for the unit that described the amount of rotation in a Turn command. Like the mapping problem that forces users to convert from an object-centric Forward, Left, Up formulation to an XYZ system, we observed users converting from "number of times to turn around" into degrees. In mid 1996, we converted the rotation API to take "revolutions" as the unit of turning.

Part of this decision was motivated by the observation sessions with the tutorial. More than once, we observed users engaging in the following sequence of steps in their very early explorations of what the system could do.

```
Bunny.Move (Forward, 1)
Bunny.Move (Left, 1)

# user reads about the turn command and decides to try it…
# user edits the existing text

# note that the user has only changed the verb.
Bunny.Turn (Left, 1)
```

Note that the user only changes the verb. If the units for rotation are understood to be degrees, we are left with a one-degree turn, which is so small as to appear to be a silent failure of the system. No amount of warning in the tutorial seemed to be effective in getting most users to change the 1 to 90, so that the rotation made sense. I could attribute this

problem to laziness on the part of our users (not wanting to type, not wanting to read the instructions[1]), but the better part of system design lies in meeting your users more than half way, and so we changed the units to revolutions, which made this problem disappear.

10.4 The Importance of One (1)

I note anecdotally that users seemed to prefer using the number 1 in all contexts where they were unclear what the units were or what the effect would be. Whether the amounts being filled in were distances, scales, rotation amounts, durations, or color values, our novice programmers were highly inclined to just type a 1 to see what happened. I never observed a user seeking the answer to such issues in the printed or on-line documentation. This lead us to observe "no matter what the question was, the correct answer is 1, all we need to do is find the right units[2]." Exploratory programming systems and APIs might well keep this anecdotal point in mind during their design.

10.5 Turning Rate

At the same time we were making the change from degrees to turns, we began to wonder if this observation extended to the units for specifying turning speed. I prepared a question for the exit interviews in order to find out which units people would prefer to use when specifying the rate at which an object turns around a single axis. Users had already seen

---

1 That users do not read documentation is something that I treat as an axiom.

2 It would appear that Douglas Adams [Adams] was incorrect: the answer to the ultimate question is 1, not 42.

the first Alice tutorial and so were familiar with the Alice `Turn` command. Users had also seen the `EachFrame` construct, which when used for certain commands, establishes a one-way constraint, as in :

```
Bunny.Head.PointAt (target, EachFrame)
```

Users were told that they were being given a survey and not a test, that there were no wrong answers to the following question.

In the tutorial you just finished, you made a pink bunny turn around in place. That command looked like this:

```
Bunny.turn(left, 90)
```

You gave Alice a direction to turn (left) and an amount (90 degrees). But what if you did not know the amount, but you wanted to make the bunny turn around and around without stopping? What would be a good way of telling Alice how fast you wanted that to happen?

A breakdown of the answers appears below:

First Impressions

The first thing to notice is that turns-per-second is a clear favorite and that degrees-per-second (the units we had chosen) came in fifth. One user actually suggested radians-per-second, but this subject was a first year engineering student and chose these units because she thought it would make it easier to do trigonometry with Alice.

Units	Count
Turns/Second	22
RPM	9
Unitless 1-10	7
Fast/Medium/Slow	6
Degrees/Second	3
Seconds/Turn	2
Radians/Second	1
TOTAL	**50**

Unitless Turning Rate

Note that the combination of a unitless ***1-10*** turning speed together with the coarser-grained variant ***fast-medium-slow*** were both very popular answers. Users who gave either of these answers clearly have no strong preference for a physically-based turning rate, preferring a more qualitative way of characterizing turning speed, even if it means using a scale that is bounded at both ends and has very coarse granularity. As an engineer, my first reaction to seeing this result was that our test users were reacting naturally and intuitively, but that they were clearly not thinking about the long-term flexibility of their answers. While not rejecting their answers, I wondered how their answers might change in the face of needing to specify a speed that a bounded scale might not be able to capture.

A conversation with one subject (female, 19, first-year economics major) is telling:

Observer: How slow is 1?
Subject: Very slowly.

Observer: So 1 isn't stopped?
Subject: No, just slowly.

Observer: And how fast is 10?
Subject: Explode.

Observer: Explode?
Subject: Yes, it spins very fast for a moment, then all the pieces fly off. (User makes an 'exploding' gesture with her hands and sounds out a booming noise).

This conversation was typical of several that I had with subjects – it would seem that some users have an intuition that objects possess a lowest and highest speed at which they can turn. Units for turning rate, such as degrees per second, offer an open-ended scale where objects can turn arbitrarily fast. While an open-ended scale seems more flexible, it would also seem to violate physics and common sense at the same time. This class of users, perhaps unwittingly, has captured this bit of common sense in their answer to this question.[1]

This question requires further study. For example, given Python's polymorphic API, it should be possible to supply both a unitless turning speed and a more traditional numerical rate-based API for turning:

1 Some have noted that we may have seeded the users with the notion that animated destruction is somehow a central concept in 3D animation.

```
# turns per second by default…
Obj.turn(left, 1, speed=1)

# …or allowing for something simple…
Obj.turn(left, 1, speed=Slow)

# …and with the power to choose something else
Obj.turn(left, 1, speed=10, units=degreesPerSecond)
```

10.6 Two Kinds of Turning : The Problem With Roll

Part One of the Alice tutorial teaches users the turn command with a pink bunny rabbit as the object under rotation:

```
Bunny.Turn (Left, 1)
Bunny.Turn (Right, 1)
```

This specification never caused any problems. Because the bunny was seated on the floor, turning Left/Right always happened in a plane parallel to the ground. Users were also shown how to lean the bunny backward (what pilots would refer to as pitch):

```
Bunny.Turn (Back, 1)
Bunny.Turn (Forward, 1)
```

Of course, this is underspecified, leaving open the amount of "roll", or in Alice parlance, the amount of rotation around the object's Forward direction. The tutorial itself did not mention this degree of freedom, but we raised the issue in the exit interviews. While holding a stuffed pink bunny toy very much like the graphical one they had been manipulating during the tutorial, I demonstrated to users the four turning directions that they had already practiced with Alice:

Observer: "Here is turning left/right [*gesture with stuffed toy*], and here is turning forward/back [*gesture again with toy*].
Now, are there any other directions that the bunny can turn? If you can think of any, don't worry about finding a word for it, just show me what it looks like. *[Hand the stuffed toy to the subject[1]].*

The first result is surprising. 87% (26 out of 30) of the respondents questioned handed the bunny back to me and reported that there were ***no other ways to turn the bunny***. To the vast majority of Alice subjects, roll was invisible for this object. I repeated this question with a model plane, but subjects still reported that there were no other reasonable ways to turn the object.

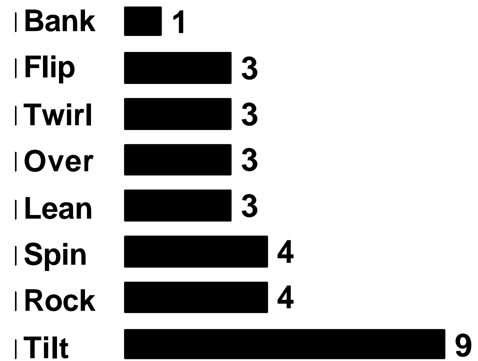
The next step was to demonstrate, via gesture, the missing degree of freedom. The subjects had a variety of reactions, ranging from surprise, "Ah, yes…I didn't see that", to reactions of skepticism: "isn't that just sort of both turn-left and turn-down at the same time?" Other users were genuinely taken aback: "oh, that's not really turning though" was heard more than once.

Once convinced that the missing degree of freedom was actually there, I repeated my question, this time, asking users to ***name the direction*** that the bunny was turning. The test subjects **never** gave me the name of a direction to turn, but instead offered up a variety of new verbs or method names to describe the motion.

---

1 This is important. By using a plush toy, I avoided suggesting words to the subjects, while still being able to communicate differences in rotation directions.

Tilt seems to be a strong favorite, but users were not at a loss to supply a wide variety of other terms too, *but in all cases, the motion was described as being in a left/right direction.*



I also find it significant that users insisted on finding a new method name to describe this sort of rotation, rather than finding a new direction to use with the word turn, which they already knew. It would seem that users think of rolling an object as a significantly different form of rotation – not at all the same sort of thing as turning.

Many months before this study, we had seen this sort of confusion coming. For pragmatic reasons we implemented this form of rotation under a different verb – called roll – simply because we had run out of good direction names (left/right was already taken for turning). Note that the word roll does not appear anywhere in the list of words that our target audience used to describe this motion, in spite of the fact that 'roll' is word that is well-known in aviation. For that reason alone, we chose roll as the verb name-of-choice before querying our users on this issue. Even the best-intentioned and well-principled guesses can be wrong.

A Possible Explaination

There are two possibilities that might explain what is happening with roll:

1.) People really do consider roll as different from pitch and yaw.

2.) The Alice tutorial is leading the witness.

It is not inconceivable that roll really is different for some objects. In the case of human perception, the human head yaws left and right and pitches backward and forward, but yaw is biomechanically much harder and far less useful. Both pitch and yaw provide an observer with more information about his/her environment (*What is behind me?* or, *What is above me?*) whereas roll provides no new or useful information, and in fact makes it harder to understand the environment by presenting it upside-down. We may have a very low-level perceptual bias against roll.

On the other hand, perhaps there is no effect here whatsoever. I am very open to the possibility that roll is no different than the other forms of rotation, and that the tutorial itself is responsible for the effect that I report. I think it quite likely that having seen turning left/right and turning forward/back that subjects found that all the "good words" had been taken when it came time to describe roll. The fact that few people even saw roll as a possibility might be due to the particular nature of the objects I asked them to manipulate, but knowing this for certain will require more study. In particular, one interesting study would be one where we split the subjects into three distinct groups, each group being shown

only two of the three degrees of rotational freedom (roll/pitch, roll/yaw, yaw/pitch). The subjects would then be asked if they could offer up any other degrees of freedom. Done over many subjects and with several kinds of objects, perhaps in several different contexts (with and without ground plane), the results of such a study would give us a much better idea whether this effect is real or imagined.

Even if roll isn't treated differently from the other directions of rotation, I would note that I was able to elicit a very strong separation response from my subject pool and that this can be useful from a system-design point of view. While psychologists can run well-controlled studies to find out for certain whether there is an effect here (a study I would be most interested in reading), I can be content as a system builder to know that I can get users to *believe* there is a difference simply by leading them through a pedagogical experience in the form of a tutorial that leaves them with no other choice. Roll is different from pitch and yaw because we teach roll last and by the time we get around to it, all the good names are exhausted, so it must be something different.

10.7 Clockwise and Counterclockwise

The terms *clockwise* and *counterclockwise* (also *anticlockwise*) are used in English to describe rotation direction. Early Alice implementations included these words in the API to describe roll-rotation, but we eventually came to remove these terms, replacing them with the roll method. We were forced to abandon vernacular terms due to ambiguity; even the implementation team could not come to agreement over what the terms meant. The

confusion comes from the fact that the terms clockwise and counterclockwise assume the presence of an outside observer whereas the Logo-like Alice direction names explicitly assume that the direction names are to be taken from the moving object's point of view. There seemed to be no happy compromise. To make *clockwise* and *counterclockwise* viewer-centric would single them out as exceptions to the Alice object-centric directional system, and to make the terms object-centric was even worse: to make clock hands turn clockwise, one would have to turn them counterclockwise from their own point of view.

10.8 Absolute Rotation

The first systems we built did not have absolute orientation commands, but now we have several ways of getting an object to point in a particular direction:

obj.PointAt (another_object)

PointAt points the Forward direction of an object directly at the origin of another object. Of course, there is an unspecified degree of freedom here: the amount of rotation around the Forward direction, sometimes referred to as "twist" or "roll". Our implementation of PointAt attempts to keep the Up vector of the pointing object aligned with the Scene's Up direction as closely as possible, which neatly constrains the amount of roll.

obj.StandUp()

StandUp is very much like PointAt, except that it orients the object so that its Up

direction is aligned with the Scene's Up direction as closely as possible (makes them parallel) without inducing any pitch or yaw in the object (no turning left/right or forward/back). Like the PointAt command, StandUp is implemented as a linear interpolation of a quaternion [Shoemake].

obj.AlignWith(another_object)

AlignWith turns an object to exactly match the orientation of some other object. This is really just an alias for TurnTo(0, 0, 0, AsSeenBy=another_object).

obj.TurnTo(pitch, yaw, roll, AsSeenBy=object)

TurnTo is a general purpose, absolute turning command and is the most difficult of the absolute rotation commands to master and control. The three numerical parameters represent the amount of rotation (given in revolutions 0.0…1.0) around each of the three principle axes of the object (note that the strange order of the parameters is vestigial and represents rotation around the X, Y, and Z axes of the object, respectively). There is an option to specify another object's coordinate system, which is handy for operations such as orienting an object so that it is turned 90 degrees with respect to some other object. This call is the least Alice-like of any in the Alice API, given that it manipulates something not unlike Euler angles, hardly an idea that novices are likely to appreciate. Note that it is harder than the others, in part, because it is the only one that does not use some other object as an orientation reference. The future work section contains a design for an absolute orientation

call that I suspect may be easier for novices to use than this.

"…if it makes me grow larger, I can reach the key; and if it makes me grow smaller, I can creep under the door; so either way I'll get into the garden, and I don't care which happens!"

– Alice considers the consequences of changing size,
Alice's Adventures in Wonderland

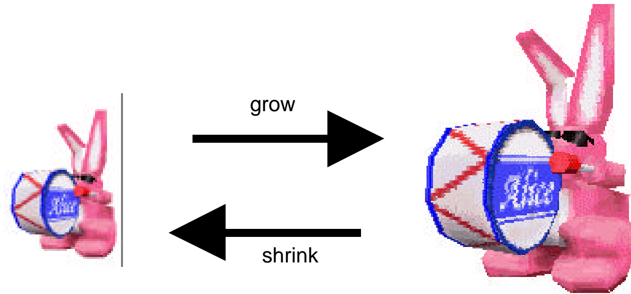
Chapter 11
Sizing and Scaling

11.1 Naming

The word Scale has several meanings, though be warned: the one that graphics programmers think of is not the one that novices think of. Many novices with whom I talked thought scale was primarily a noun (a device for determining the weight of an object) and so guessed that that the word scale had something to do with measuring an object's weight. This situation is very much like the word *translate*, where we, as technical professionals, have appropriated a word from colloquial use, and have forgotten that the word has a life outside the world of our technical jargon. In reaction to this discovery, we have decided to call Alice's scaling operations Resize and SetSize.

Note that users are not designers. As Nathanial Borenstein suggests, we listened to

our users but ignored what they said [Borenstein]. When freely asked what word they would use to describe the operation that makes the picture on the left look like the picture on the right, we had 82% of 95 respondants tell us “grow” (with the inverse operation being “shrink”.) Nearly all the rest answered “increase” and “decrease.”



This is interesting because people seem to be very strongly motivated to use two different words to describe the operation of changing size, and yet, as programmers, we see the value in folding this functionality into a single verb that can be used to describe scaling in either direction. Consider the case of an object whose size is controlled by a slider. The programmer who creates this tiny program will probably end up writing a code snippet something like this:

```
value = read_slider()
if value > orginial_size:
		Object.grow(value)
else:
		Object.shrink(value)
```

We know from other studies of novice programmers that the if statement is a significant stumbling block – and one that we know how to avoid in this case:

```
value = read_slider()
Object.resize(value)
```

As it turns out, it does not take much convincing to show novices that grow and shrink are troublesome verbs. Here is a typical interaction with an Alice test subject in the post-observation interview:

Observer:	*So how would you make the bunny on the left [the small version] turn into the bunny on the right [the larger version]?*
Subject:	*that would be grow(2)*
Observer:	*And to go from right to left?*
Subject:	*shrink(1/ 2)*
Observer:	*So both commands, shrink and grow, they each take a single number?*
Subject:	*Yes….*
Observer:	*So what would you want grow(1 / 2) mean?*
Subject:	*[pause]*
Observer:	*And what would shrink (2) mean?*
Subject:	*They would both be mistakes.* [1]

Users had a very hard time seeing the correct semantics for growing by an amount less than one or for shrinking by an amount greater than one. Nearly all thought that these conditions should result in an error of some sort.

Once they were convinced that grow and shrink were not perfect, they were

1 During these informal conversations, I also came across a fairly common (misplaced) intuition that some people had regarding scales. About 5% of the subjects claimed that in order to make something smaller, a person should not use fractions, but should use negative numbers. To this segment of the population, grow(2) and shrink(-2) should do the same thing. This is another example of listening to one's users but ignoring what they say.

motivated for a solution: resize. When asked if resize(2) and resize(1 /2) captured the meaning of grow(2) and shrink(1 / 2), most said yes, though many admitted that this is not the first word they would think of.

Our solution: provide a resize call so that users would be happy in the long term and provide grow() and shrink() terms in the help index so that they would be lead to the correct solution in the short term. There is a mild argument in favor of providing grow and shrink as aliases, given that the reaction from the user community was so strong. I would also argue that aliases in this case can't hurt, and that it could even be used as a runtime check: the first time a programmer uses grow or shrink, we have an online pedagogic opportunity to teach the novice programming community the advantages of using resize. Providing many aliases to the same functionality is often seen as a system flaw, but there is some evidence [Furnas] that error rates in user interfaces drop when system commands have several names by which they can be accessed.

During the post-test interviews I also discovered that users were evenly split on the meaning of Resize(1) – some thought it would do nothing, others though that doing nothing was a waste and so wanted Resize(1) to set the size of the object back to its original size.

11.2 Relative Geometry Scaling without Space Scaling: Resize

The semantics of the Resize command represents one of the first breakthroughs of the Alice API. Our first user observations were done with graduate and undergraduate

students enrolled in a graduate-level graphics class, who were given the SGI version of Alice. We performed frequent post-mortems with these students and logged dozens of hours of observation time with them, watching them struggle with Alice, trying to determine what they found hardest. At the end of the semester, we found that there were many complaints about the Alice Scale[1] command (which allowed Alice programmers to multiply the current dimensions of an object by a constant factor.)

The first thing that people complained about was the strange side-effect that scale had on objects. Calling

```
Bunny.Scale(3)
```

On an object resulted in a bunny that was 3 times as large, but it also had the effect that

```
Bunny.Move(forward, 1)
```

Moved the bunny forward 3 meters, not 1. Our users correctly surmised that making the bunny larger also changed its sense of space. To get the correct behavior out of Alice, the programmer needed to track each object's scale factor by hand and then divide by the appropriate scale factor just before doing the move command. Things got a fair bit more complicated if the programmer wanted to perform a move command on a scaled object in

---

1 We had not yet discovered the troubles that less experienced users would have with this word.

some other object's coordinate system using the `AsSeenBy` parameter. In this case, the programmer needed to know the relative scale *between* the two objects.

This terrible state of affairs came about because we failed to insulate our users from the implementation: the 4x4 matrix. The most traditional representation for position, scale and rotational information for a 3D object is the four dimensional homogeneous coordinate transformation matrix. This mathematical object is an efficient way of encoding an object's orientation in space, and forms the backbone of a great many of the world's 3D graphics software and hardware. Understanding the mathematics behind this object is usually the subject of a specialized linear algebra course, often taught as a college-level freshman or sophomore mathematics course. Matrices can be used in a wide variety of applications, one of which is encoding a compact representation for a spatial frame of reference. Two dimensional objects can be encoded with a 3x3 matrix, three dimensional objects require a sixteen-element 4x4 matrix. Matrices used for this purpose encode translation, rotation, scale and sometimes shear.

Unfortunately, the way that scale information is encoded in a 4x4 matrix necessarily causes it to interact with position information when the matrix is used to compute translations. Quite literally, the scale elements scale an object's space and the object gets larger or smaller as a side-effect. Unfortunately, so too does its sense of distance.

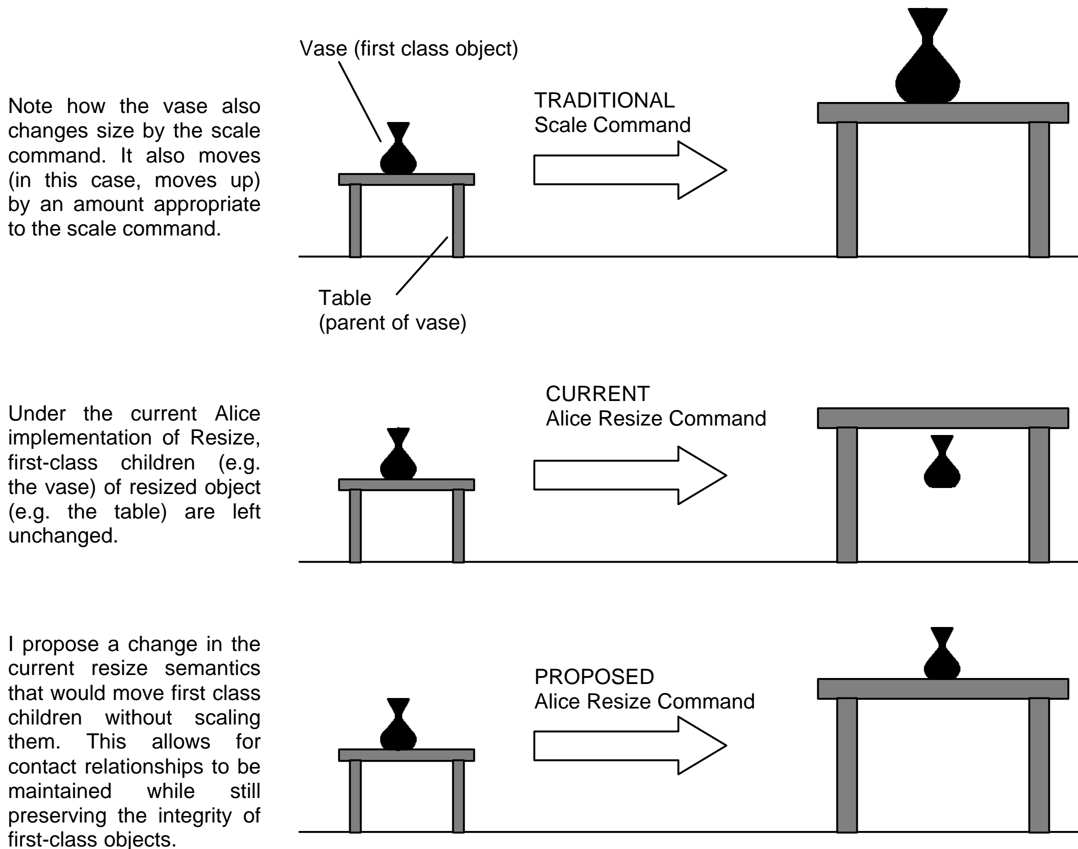
This is not to say that the scaling of space is a bad thing: in order to make virtual maps, models and other miniaturized graphics elements, "one inch equals a foot" is often

exactly the semantics that a programmer needs. The point is that this operation is very different from the operation of "make this object twice as large." Until now, the graphics community has been guilty of making the scale function perform double duty.

In light of these troubles, we implemented the Alice Resize command. This command changes the size of an object, and then caches the accumulated scale factor(s) with the object. When the user then executes a move command, Alice divides the distance to move by the cached scale factor in order to overcome the effects of the scaled space. A meter in Alice is always a meter..

11.3 The Scope of Resize

Implementing the scale operation so that it does not change space eliminates the most common complaint against traditional scale implementations but it does introduce some interesting questions regarding scope. A traditional scale operation operates on entire sub-trees, but the Alice resize command works on ObjectsOnly, stopping at the first-class object barriers inside the tree. In Alice, if a vase is a child of a table, resizing the table to be twice as large will not make the vase twice as large as well – only the table is affected. These semantics will, in general, leave the vase behind, as the table grows larger. While this strange behavior has never been formally tested with users, it has been seen in practice in various guises and I believe it to be a real and easily avoidable problem. In the future work section, I suggest that if a resized object has first-class sub-objects somewhere in the subtree, those objects are moved through a distance commensurate with the scale factor.



11.4 Non-Uniform Scale

As mentioned in section 9.2, users can resize an object along just one dimension using one of the words : FrontToBack, LeftToRight, TopToBottom. The default is to Resize All.

11.5 Absolute Scale : SetSize

The operation of setting an object's size is a curious omission from most 3D APIs. In fact, even the earliest versions of Alice did not have a way of setting the dimensions of an object directly and it was not until we were asked for this functionality by our user community did we implement a ScaleTo call in early 1993. This first call was a baby step along the way, though. Its semantics were simple: it set the scale factors for an object relative

to the size of the object when it first appeared in the world. With this call, a programmer still could not make an object be 2 feet long without first knowing what the original size was.

What are the possible alternate forms of an absolute SetSize call? Our users were very helpful in suggesting the following possible ways of setting an object's size.

- Set size to a particular distance   `obj.SetSize(FrontToBack, 1)`
- make X times as big as it is now   `obj.Resize(3)`
- make X times as big as another object   `obj.Resize(3, asSeenBy=fred)`

And some forms of scaling that we currently do not support:

- make X times as big as its original size
- make X meters longer (FrontToBack, LeftToRight, TopToBottom)
- make X meters longer than object Y

11.6 Volume Preserving Scale: LikeRubber

Another form of scaling that we implemented was one that for many years we called stretch. Stretch looks very much like Resize, except that it preserves the volume of the object being resized. Make an object twice as tall and its width and breadth will get smaller by about 0.707.

$$\begin{aligned} Volume &= H \times W \times D \\ &= 2H \times \sqrt{2}W \times \sqrt{2}D \end{aligned}$$

This technique gives objects a rubbery, plastic look to them which is sometimes a desirable effect and has been long used in traditional character animation [Lasseter][Thomas]. Recently, we folded this functionality into the Resize command with an optional modifier: LikeRubber, as in:

```
Obj.Resize(TopToBottom, 0.5, LikeRubber)
```

11.7 Summary

- Changing the size of an object and changing the scale of the space in which it operates are two different, useful but orthogonal ideas. The common implementation of 3D graphics systems in the form of 4x4 modeling matrices makes this separation difficult, but it is well worth the effort.
- For novices, Scale is a terrible name for the operation that changes an object's size.
- The HowMuch keyword can be used to control the scope of the resize action. There is an open question surrounding the issue of moving first-class children of resized objects.
- Alice supports several different kinds of size changing, though not all of the kinds of resizing that users might want to do.
- Alice supports a volume-preserving scale through the optional token LikeRubber.

Chapter 12
Color, Texture and Visibility

12.1 Color Names

Our first attempt at color names came from the standard list of colors that ships with the Unix X windows system. This is a relatively rich set of names, but is quite large and contains the names of colors that people tend not to know (*Peru*?) We eventually replaced the color names with the names of Crayola Crayons™ on the assumption that this list would be more familiar to our users, and, while smaller, it would provide very rich coverage of color space for most purposes.

At first, colors in Alice were represented as strings:

```
obj.SetColor("Red")
```

but this led to a problem with our novice users who were confused by the need to use quotation marks. To our users, it seemed inconsistent that things like `Forward` should not require quotes where tokens such as "Red" and "Blue" did. To remove this confusion, and

to simultaneously side-step the issue of having to teach novices the subtle differences between constants and strings, we decided to change the SetColor API so that it no longer took strings, but took color tokens that the system now supplied: Red, Blue, Green and all the other color names became reserved words in Alice. This certainly does pollute the Alice name space to some degree – it adds 172 tokens to a namespace that already contained 602 reserved tokens in the form of class names, constants, and the like. Of those 172 color name tokens, 100 of them are names for grey levels (gray3, gray4, gray5, etc.). The cost of not adding these color names to the namespace is the cost of confusing the user and making them memorize yet another rule.

While color names are easy enough, they have the disadvantage of being imprecise (Bluish green? Greenish blue?) which makes them hard to use in comparing and matching colors across systems and color references. If a user sees a nice shade of blue in his or her favorite paint program, it might be difficult to find the Crayola color name for that particular color. As API designers we bow to this pressure and allow colors to be specified in RGB (Red, Green, Blue) triples as well, where the range for R, G and B is the usual Alice range of 0.0 to 1.0. We can afford this API flexibility due to the polymorphic nature of the Python programming language – at call time, the implementation can always tell the difference between SetColor(Red) and SetColor(1, 0, 0).

Of course, sometimes you can't win as an API designer if someone else gets to your target audience before you do. We got this email from an artist who had some exposure to

creating HTML web pages:

```
From: XXXXX@murasaki.cs.Virginia.EDU
To: uigroup
Subject: setColor
Date: Mon, 10 Jun 1996 22:06:53 -0400

I thought this might be useful:
I was surprised to learn that the RGB values for
setColor run between 0 and 1. This is because
I have encountered RGB values before in HTML,
where color is set with RGB hex triplets. i.e.
0 through F.

Beth
```

Alice does not supply color specs in the form of Hex values.

12.2 SetColor and Scope

SetColor is one of the set of commands that operate on object subtrees. Setting an object's color will set the color of every polygon of every sub-part of the object with that color. The tree traversal stops when it encounters first-class objects, keeping the color change from affecting "attached" objects, only parts. This default behavior can be overridden with the HowMuch keyword parameter (see page 127).

12.3 GetColor

Querying an object's color would seem to be a trivial matter, but in practice is full of subtleties and ambiguities that we wrestled with for a very long time, searching for a way of simplifying the model so as to capture the most expressive power with the least intellectual

overhead.

At the heart of the problem is this question: What does this return?

```
Color = Bunny.GetColor()
```

What does it mean to query the color of a multi-colored object? To answer this question, we need to know where color is ultimately stored in a 3D object – what is the lowest-level part of a 3D object that we can apply a color to? Current systems have a variety of strategies for handling this question. While some 3D object formats allow color to be set on individual vertices, others bind colors to each polygon, while still others will gather like-colored polygons together into "face sets" – allowing individual objects to contain an arbitrary number of face sets.

Each of these schemes imposes problems to the API designer: with face sets, a GetColor command could conceivably return multiple values. How does the system print such a value out? What would the GUI look like for displaying the color of a mulitcolored object? What is the API that controls this? Is the multivalue record that comes back from GetColor a color at all? How could one pass such a record to a SetColor command and expect it to work? To explore these issues, we came up with several alternate paper designs:

- one color from a specified (part, face-set) pair
- flat list of colors from all this object's face-sets
- dictionary of object: (face-set, color) pairs across this subtree
- dictionary of object: (face-set, color) pairs across this subpart-tree

All these solutions ended up being far too complex. We concluded that:

- The best solution is a simple one, and therefore we should eliminate the notion of face sets. Objects and parts in Alice each only have one face-set, and therefore can only have one color at a time. To make multi-colored objects in Alice, an object must have parts, each with its own color.
- The operations to get and set colors were a lot less powerful and less interesting once the Alice object library was filled with texture mapped objects.

As a result, the GetColor method in Alice always returns just one color: the color of the single face-set inside the node being queried. No attempt is made to traverse the subtree to gather the colors of sub-parts. This guarantees that the value that is returned from a GetColor call is always a color, never a multi-valued record, and therefore can always be passed to SetColor.

This part of Alice represents months of design effort and uncountable paper designs that were generated in an attempt to make this part of the Alice API consistent and easy to use. It is sobering to note that our effort here was probably wasted: it seems that users rarely query the colors of objects through code.

12.4 Colors and Textures

As if this was not bad enough, not only do users rarely query the color of objects, but the whole notion of object color begins to fade in importance in the mind of a novice if

the objects in the scene are supplied with high-quality, hand-painted textures.

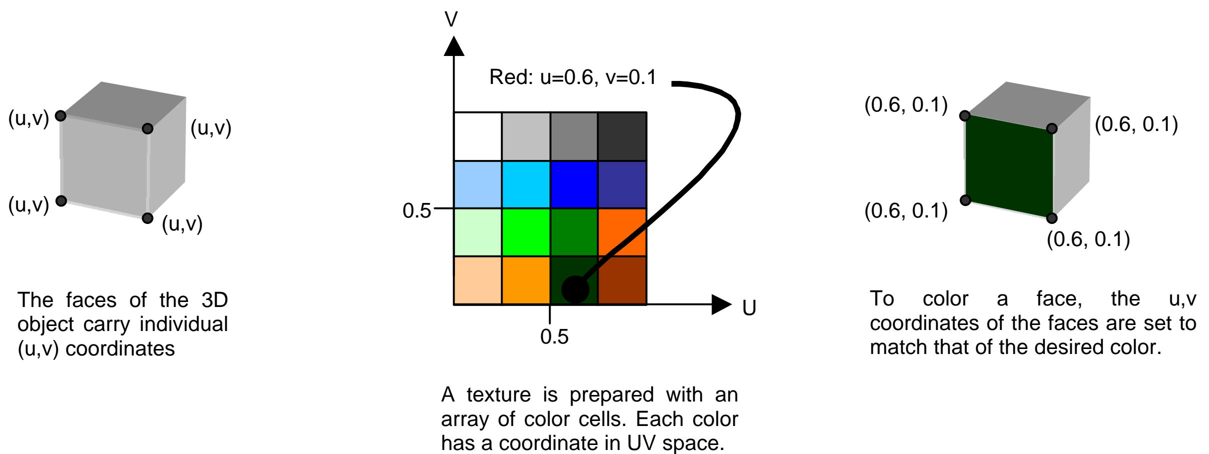
Of course, this statement should not be taken out of context: I do not argue that textures obviate the need for polygon color. Rather, I argue that when 3D objects come with interesting and skillfully-rendered textures painted on them, there is less incentive for novice users to want to change those colors and so the complexity of the color API becomes less of a stumbling block to new users. The fact that object color getting and setting is complex becomes much less of an impediment than it might be in a world where all the objects come in gray, thus inviting, even begging a novice to look for color-setting mechanisms.

Even with texture, there is a place for object color:

- there are objects that do not have their own textures (objects from the web and generic objects like cubes and cylinders)
- texture colors cannot be changed very easily at runtime
- the color of a texture will change depending on the color of the underlying polygon color

12.5 Setting Face Color with Texture Maps

Finally, I would like to describe a technique that allows programmers to set the face colors of a 3D polygonal object, even when the graphics database does not explicitly store color properties for the individual faces. To work, this technique requires that the system perform texture mapping, storing U,V coordinates at each vertex. Given this capability, it is a simple matter to create a texture with a grid of colors (no more than one pixel per color is necessary). This gives each color in the texture map a unique UV coordinate. Given a polygonal face and a color to apply, the system can find the vertices of the face in question, and finally to apply those UV coordinates to the vertices of the face. The face is rendered with a solid color, but that color is supplied via a texture, not through color data stored with the face.



The faces of the 3D object carry individual (u,v) coordinates

A texture is prepared with an array of color cells. Each color has a coordinate in UV space.

To color a face, the u,v coordinates of the faces are set to match that of the desired color.

12.6 Opacity, Invisibility and Hide/Show

The earliest versions of the Alice API had three different calls:

```
obj.SetTransparency(X)          # X is a number 0.0 to 1.0
```

```
obj.SetVisibility(bool)         # bool is True or False
```

```
obj.Hide()
obj.Show()
```

We changed SetTransparency to the somewhat more obscure SetOpacity, because we observed that users were getting the value of the parameter backwards, passing in 1.0 to get a solid, easy-to-see object, when in fact, this made the object invisible (100% see-through.) The effect isn't terribly strong, but is present and costs nothing to get it right.

SetVisibility was used to make something go away from the screen completely and was a Boolean toggle.

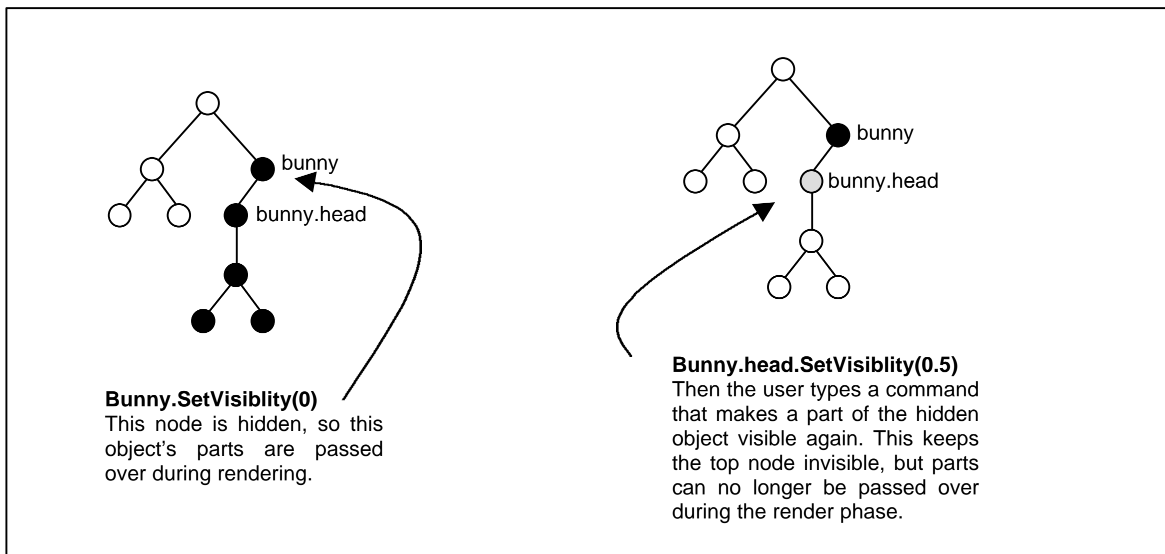
Hide and Show were optimization routines used to manually cull the object database. An object that was hidden would stop the renderer's tree traversal just as if the object were a leaf node. Thus, anything under a hidden node will not be rendered until the node is sent the Show() method. This can speed up program performance in those situations when the programmer knows for a fact that an object cannot be seen (behind the camera, occluded by other objects, very distant objects). As retained mode 3D databases like D3D become more sophisticated, they start to employ these sorts of optimizations automatically, relieving the

programmer of the burden of managing such things. Still, this call can be useful.

Over time, it became clear that we could fold these calls together into a single method where most of the complexity was hidden in the implementation:

```
Obj.SetVisibility (X)            # X is 0.0 to 1.0
```

When X = 0, the object and all its parts is hidden. If any of Obj's children should become visible, the Obj is no longer culled but is now traversed in order to reach the visible children, though the object itself is not rendered. Should the implementation detect that all of the objects in a given subtree are invisible, the implementation is allowed to cull that subtree as if it were hidden. When X > 0, the object is rendered in varying levels of transparency, with lower numbers denoting a more see-through object.



Visibility and Simulation: How Aggressive Should We Be?

One issue that we never resolved either through design sessions or through user observation was the interaction between visibility and the simulation. Does making an object invisible pause that object as well? Does hiding it? If an object is invisible does it participate in any part of the simulation? What about collision detection? What about other forms of "rendering", such as sound? Casting a beam of light? Without guidance and a principled way of addressing these issues, we erred on the conservative side and left these issues in the hands of the application programmer. While I believe that the ultimate answer to each of these questions is "it depends on the application," I also believe that there may be a useful set of defaults that remains to be discovered. I would strongly endorse an API that made all combinations of visibility, renderability, and simulation possible, but would hope that there are smarter default interactions than we currently have.

12.7 Summary

The following chart collects the primary points of the previous chapters:

New	Old	Benefit
Forward, Left, Up	X, Y, Z	Forward, Left, Up avoids a common mapping step. Ambiguous cases are no worse than XYZ.
LeftToRight, TopToBottom FrontToBack,	X1-X2 Y1-Y2 Z1-Z2	FrontToBack expresses a commonly used quantity that relieves script writer from performing explicit math and concerns over sign error.
Any object is a coordinate system/ AsSeenBy Keyword parameter.	World Coordinate system and Local Coordinate system.	AsSeenBy builds the notion of change of coordinate system into the implementation, and relieves the programmer of this common and error prone task. Leaves “coordinate system” as the primary pedagogical element.
Place	Translate	Common operation for scene construction, but not directly supported in other APIs.
Resize	Scale	No scaling of space. Better Name. Allows other resizing operations (LikeRubber).
Move	Translate	Better name (though some data suggests that Slide may be even better).
Narrow/Polymorphic API	Wide/One Command API	Polymorphism allows controlled exposure to power
SetVisibility	SetVisibility Hide SetAlpha	One call that folds together several functions, masking the complexity as an implementation detail

The 3D stuff seemed pretty easy, but I kept mixing up the periods and commas
An Alice Test Subject

Chapter 13
Programming Language Issues

13.1 Controlled Exposure To Power

How does one design a system in such a way that it allows novices to become experts without requiring the novice to abandon his or her hard won skills? Bill Buxton first asked this question in the context of GUIs [Buxton] where he argued that classic linear menus are (in a limited way) appropriate for novices, but represent a barrier in allowing novices to become experts. Traditionally, novices become experts by forsaking menus for the higher performance of keyboard shortcuts or “hotkeys.” Buxton argues that this solution asks users to forget their hard-won muscle-oriented menuing skills so that they can learn a whole new set of gestures in the form of keyboard shortcuts. Buxton’s solution, *Marking Menus,* show how GUIs can be designed so that novice skills are still useful as the user gains skill, providing a seamless transition from novice to expert.

Alice demonstrates that APIs can be designed in the same way. If the language

supporting a programming library provides default parameters and polymorphism, library designers can create programming libraries where the defaults are set in reasonable ways to help smooth the way for newcomers to the library. As those newcomers gain expertise, the library calls they already know can still be used, overriding the defaults, and gaining more control and more sophisticated behavior. In this way, the complexity of the API is exposed to the user gradually on a "need to know" basis. The library designer, of course, has to be sure that the defaults are set correctly and that the abstractions are appropriate, but at least with these two language features (polymorphism and default parameters) the API designer has the tools necessary to provide controlled exposure to power.

13.2 Keywords

To provide controlled exposure to power in the Alice API, nearly all Alice commands are allowed to take optional keywords which allows the simplest command forms to act as the backbone for more sophisticated commands. For example, all of the following are accomplished through the same Move command:

```
# simplest form of move
obj.Move(Forward, 1)

#move, but make it take more time
obj.Move(Forward, 1, Duration=3)

# move, but go faster
obj.Move(Forward, 1, Speed=4)

# in some other object's coordinate system
obj.Move(Forward, 1, AsSeenBy=Camera)

# use a different interpolation function
obj.Move(Forward, 1, Style=Abruptly)

# start a steady-state animation (note the lack of an amount)
obj.Move(Forward, speed=2)

# start a steady-state animation and stop it after 3 seconds
obj.Move(Forward, speed=1, lifetime=3)

# delay execution of this command – this is only a definition
scoot = obj.Move(Forward, 1, Start=No)

# you can combine multiple keywords
obj.Move(Forward, 1, Duration=3, Style=EndGently, AsSeenBy=Scene)
```

```
# other commands take different keywords.
# simple form of SetColor
obj.SetColor(Red)

# control what parts of the object are turned Red
obj.SetColor(Red, HowMuch=AllChildren)
obj.SetColor(Red, HowMuch=ObjectOnly)
```

One Alice test subject made the comment that with the Alice API: "you can get as complicated as you want" which is exactly what we were shooting for.

Instead of keywords, we could have implemented this scheme using Python's default parameter feature. Under that design, we would have function calls that looked like this:

```
# ALTERNATE ALICE DESIGN –
obj.Move(Up, 1, Duration(2))
obj.Move(Up, 3, Rate(2))

# of course, this gives us a natural place to include modifers
obj.move(Up, 3, Duration(2, hours))
obj.move(Up, 3, Rate(2, miles_per_hour))
```

This also has the advantage that it can be done in a language that does not support keyword parameters, but that does support polymorphism and default parameter values.

We decided against this design in Alice because our target audience was already having a hard time dealing with parentheses for function calls. Using the keyword=value form meant that we lost the ability to cleanly fold in modifiers (units for duration), but we gained code legibility in the form of less nesting of parentheses.

13.3 Beware of Irregularities

This kind of flexibility sometimes gives rise to overconstrained conditions:

```
Camera.Turn(Left, 1, Speed=2, Duration=5)
```

In which case, Alice produces a dialog box to warn the user that something is amiss.

13.4 Beware of Ambiguities

Sometimes even the most careful choice of names can confuse users. The following command smoothly animates a color change: it turns an object yellow over an interval of one second:

```
obj.SetColor(Yellow, Duration = 1)
```

This once prompted a user to ask: "why is it still yellow?" This user mistakenly thought that the object should be set to Yellow for duration of one second, and then should return to its previous color. We almost never saw this error repeated once the tutorial was changed to explain the Duration keyword more clearly.

13.5 Controlled Exposure To Power in Other Languages

Python's language features facilitated much of Alice's API design. Other languages cater to different programming needs, and therefore impose different constraints than Python's very dynamic environment. Languages that lack polymorphism through keyword parameters, default parameter values, and variable argument lists can make it difficult for API designers to duplicate the Alice-like feel to the AnObject API. Keyword parameters allow the Alice API to hide powerful functionality inside simple commands. Default values for those arguments allow those features to be exposed "on demand

needs it. Variable argument lists allow those keywords to be used freely in combination with one another. All these qualities together provide the controlled exposure to power that

characterizes the Alice API.

13.6 Language Features We Wish We Had

Python is a wonderful language for interactive scripting, but this is not to say that Python leaves nothing to be desired. The features listed below do not reflect simply a programmer's wish list that would make the implementation easier, they are all language features that I believe would, if Python supported them, make the end-programmer Alice experience an easier and more enjoyable one.

Constants

Python currently does not have the notion of a "read only" attribute. Any object in the interpreter's namespace can be overridden, with the exception of several system-defined attributes that begin and end with double underscores. This makes Alice programs fragile because users can inadvertently overwrite system tokens like `Forward` and `Up`, leading to ill-behaved and very hard to debug Alice programs. Changing the language to support constants is relatively easy, and like case sensitivity, we may find it worthwhile to actually modify the language to gain this language feature.

Assignment hook

Alice programs would be more stable and more powerful if Python supported an "assignment hook" – a function that was called before an assignment was made to an object

in a user script. The function to be called should take the rvalue (value being assigned) and the lvalue (object being assigned to), with the ability to pass back a Boolean value denoting if the assignment succeeds. Alice end-programmers would (we hope) never need to (or want to) use this feature, but the implementation could use it to help warn Alice programmers should they accidentally assign one Alice object to another (Camera=Bunny). This operation has no well-formed meaning in Alice, but is easy enough to perform. If we had the ability to catch such errors with an assignment hook, we could warn the user that this operation was dangerous, with expert override to turn the warnings off. Also, with an assignment hook, the implementation could tell the difference between

```
Bunny.Move (Forward, 1)
```

which probably denotes an animation that should be defined and then executed, and

```
Slide_The_Bunny = Bunny.Move(Forward, 1)
```

which is better interpreted as a command that is simply being defined for execution at some later time. The first command would run, and the second command would be created in paused state.

Robust Parsing of Mixed Fractions

We noted that novices were highly inclined to type 1 / 2 instead of 0.5 in order to express the notion of "one-half." It seems reasonable that users might wish to express mixed fraction quantities in a similarly natural notation: 1.5 would be 1 1/2 . Of course, this sort of

thing would require a more sophisticated parser than Python's.

Call Parameterless Commands Without Parentheses

Some commands in Alice take no parameters at all:

```
Obj.Destroy()
```

And a common user error with this kind of command is to leave off the trailing parentheses. It is extremely hard to justify and explain to a novice user why

```
Obj.Destroy
```

is malformed. Both Pascal [Wirth] and Hypertalk [Hypertalk], were specifically targeted to novices, and both languages permitted programmers to use both forms of the command. This would be a welcome addition to Python.

Note that in Python, C, and C++ (but not in Pascal), a function without parentheses is an expression with a legal value, which means that evaluating such a string of characters leads to a silent failure which is very hard to explain.

Parameter Passing "*By Closure*"

Pass-by-closure is a language feature that allows a library provider to designate that a parameter to a function is a callable object with an environment. In Alice's context, we would very much like the ability to evaluate the environment at the point of call, but to delay

the execution of the function until later.

The best way to understand our motivation for wanting such a language feature is by example. We wished to implement a function that would allow a programmer to register a function with Alice so that the function would be called once for each execution of the Alice main loop. We called this function "DoEachFrame", and we envisioned a typical call to DoEachFrame to look something like this:

```
AddAction(obj1.PointTo(obj2) )
```

Note that in this context, we would very much like the expression obj1.PointTo(obj2) to not be evaluated at the point of call, it is merely a description of the function that we wish to have called – obj1 and obj2 and PointTo are all parts of the expression that need to be resolved, but in spite of their appearances, we do not want this entire expression to be evaluated before being passed to DoEachFrame.

Even though designers have the ability to control some of the parameter passing semantics (by value, by reference, by name, etc.) this is an example of semantics that the current Python implementation does not support. Of course, we could implement this functionality by breaking the expression into individual terms:

```
DoEachFrame(obj1.PointTo, obj2 )
```

But then we would have to explain to novices why commands sometimes appear one way and sometimes they appear another. The time-of-evaluation for parameters to functions

is a subtle issue that we would rather not burden our target with. All this would take is a new parameter passing keyword:

```
DoEachFrame(closure arg)
```

Where closure denotes that arg is not to be fully evaluated at the point of call, but all its subexpressions are.

This is somewhat like using a quote in LISP to suppress evaluation, but differs in the one critical detail that with LISP quotes it is the caller and not the implementation that is required to annotate the expression to suppress evaluation. Pass-by-closure is defined as a syntactic annotation that is to be applied at the time a function is *defined*, not at the time the function is *called* by a client.

It is apparently possible for a skilled programmer to construct pass-by-closure semantics from within the Common Lisp Object System (CLOS) [Steele] by using macros and the language's meta-programming facilities. By careful design, a programmer can re-defines the semantics of function calling from within the language, which is not unexpected in a language with reflection capabilities and an open implementation. Still, pass-by-closure is not a language primitive and is not highlighted in CLOS.

Differences between Interactive Mode and Scripts

Finally, there are some interesting differences between commands evaluated in the Alice command box (a one-line interpreter) and commands executed in batch through an Alice script. The table below gathers together a summary of the issues:

Issue	Interactive Mode	Script Mode	Note
when to renderobjects	after a carriage return in the command box or an "apply" operation out of a GUI control panel.	Once per frame, at the end of the loop.	1
textual feedback for operations that return a value	always	never	2
undo	always on, except during the execution of another undo	always off, esp. during the micro state changes inside an interpolating command	3
where do you put objects when you create them?	somewhere in view	a canonical location (origin)	4
Re-parenting / promoting objects	when bounding boxes don't overlap, promote to first-class and re-parent to scene	never re-parent unless asked	5
requests to change state without visual effect	Catch some special cases and warn user with a dialog warning box.	Never warn	6

Notes:

1: Rendering and computation are interleaved on a "frame-by-frame" basis in Alice. One frame's worth of computation is followed by a call to the internal render() function which redraws the entire window.

2 : This refers to calls like obj.getcolor() which will return a values. The point is that simple queries like this should print their results in interactive mode, and should not require the user to explicitly ask for this with a print statement.

3: Single commands in Alice can be undone, but not the evaluation of entire scripts.

4: The current Alice solution for interactive mode is not satisfactory. See page 209 for a discussion of failure modes.

5: This subject is covered in section 8.4.

6: Examples include : Resizing and object by a factor of 1.0, rotating an object 360 degrees without animation.

It's hard to make predictions, especially about the future
Yogi Berra

Mistakes were made…
Ronald Reagan

Chapter 14
Future Work

14.1 Introduction

There is much work left to do in Alice, some of which is research, some of which is development. This chapter will serve as a summary of the flaws and shortcomings mentioned in previous sections of this dissertation (with cross references back into the text), and a look ahead and future features that Alice might employ that could make behavior specification easier.

14.2 Alice's Biggest Flaws

No Modeler

Creation of new media absolutely requires the creation of new content. While not

having a modeler is fine as a research constraint, this restriction will permanently restrict the use of Alice to people who have modeling software and the skill to use it. The "Big Tent" goals of Alice (3D everywhere for free) will go unfulfilled as long as creating new 3D shapes is impossible.

Straightline code executes in Parallel

The fact that code does not execute in a linear fashion in Alice is a huge issue and prevents a lot of straightforward exposure of the tool to novice audiences. There is no easy fix for this flaw, and may require a significant redesign in order to correct. The main issue that needs to be addressed is the launching of parallel animations. A better implementation should have the following qualities:

- Sequence should be the default condition, with parallelism being possible through the application of some keyword or command.
- The Command box should execute in parallel with the running script as it does now.
- Synchronization and locking mechanisms should be present but like the current implementation, should be as automatic as possible. Users should not have to worry about launching their own threads most of the time.

Typing still too hard for novices

Users should be able to specify behaviors through ways in addition to typing. One strategy might be to employ GUI tools that assist in making the creation of textual script fragments easier. This could help introduce script writers to the polymorphic APIs that Alice has, showing off the complete set of keyword parameters and the default values they carry,

along with the constraints that can sometimes exist between formal parameters (e.g. you can't pass duration, rate and amount all at the same time – such a system is overconstrained. A GUI panel might be able to expose these constraints in a useful and lucid way.) Building GUI panels automatically for any generic API call is an interesting research thrust and would build on work first started in the Mickey system [Olsen].

Decomposing Animation Too Hard

While 3D animations are easier to build with Alice than they have been in the past with other tools, it is still the case that the Alice AnObject API relies on primitives that are far too low-level for the easy specification of many common 3D animations. It still requires a great deal of skill to look at a walking human and to subsequently decompose that behavior into the proper set of joint-angle turn commands. Most interesting animations are probably like this. It does suggest that further research should be directed at the problem, hopefully finding other primitives, abstractions or whole new mechanisms for creating 3D animation.

Undo Needs Improvement

There are several changes that the Undo command needs:

- Making undo capture a sense of path, not just state (page 48)
- Making the next undo operation visible
- Making undo more selective – allowing users to undo animations without undoing interactive camera motion.
- Allow a `DoInOrder` command to be undone in a single undo step.
- Redo: Undo's tricky friend.

14.3 Future Changes To Python

There are a few easy-to-make changes to Python in Alice's future:

- Make a full-fledged namespace object so that we can implement read-only objects (constants) and an assignment hook.
- Make tuples mutable: a = (1,4,3) a[1] = 2
- Implement Lists with a better data structure so that the append operation is not linear in the size of the list.

14.4 Provide Useful Views

The current Alice runtime system creates a single rendering window. There several arguments for facilitating the creation of other points of view other than the default Alice view. Some suggested abstractions for interactive 3D graphics include:

MapView

This is a point of view taken from high above the scene, often rendered in a parallel projection so as to make eliminate foreshortening (in architecture, commonly called a *plan* view). An overhead point of view that includes the location of the user (presumably some other camera in the scene) can help the user navigate and orient in a 3D environment. Getting a map view is critical for gaining survey knowledge of unknown 3D environments, both in virtual worlds and in the real world [Darken].

Elevation Views: Left, Right, Top, Bottom, Front, Back

It would be a helpful if in addition to the `PointAt` command, Alice provided a

command for getting a profile shot of an object along the six canonical directions for an Alice object, a point of view that in architecture is referred to as an *elevation*.

Over-The-Shoulder View

There are first-person 3D games for the PC and for dedicated video game consoles that perform sophisticated camera control algorithms based on the manipulation of a 3D avatar in the virtual world. These interaction schemes display the world in an *over-the-shoulder* or *wingman* point of view, forcing the user to interact with the world vicariously through a directly controlled 3D puppet from a vantage point that is above and slightly off to one side of the avatar. In Alice, it could be helpful to automatically generate these points of view by command.

Over-The-Shoulder View - This is a screen shot from *Mario World,* a 3D adventure game for the Nintendo 64 game platform. The player has no direct control of the camera's position and orientation – this is automatically computed from the motion of the character that the player controls. This over-the-shoulder shot is subtly but importantly different from a true first-person point of view if only because this vantage point allows the player to see at least a little of what is behind Mario.

Object Creation View

This is not so much a camera position as it is automatic camera control that engages upon object creation. Currently, when an object is created in Alice, the object is placed on the ground and at the origin. There are several failure modes to this:

- Object can be behind the camera
- Object too small to see
- Object too far too see
- Object too large to see (camera inside the object)
- Object interpenetrates other objects

Future versions of Alice may want to include smart-object creation (preventing inter-object penetration) and smart camera controls (moving, rotating and zooming the camera so that newly created objects are always visible). Usability testing will need to be done to determine if there are surprises that come about by wresting control of the camera from the user, and in maintaining context as the camera moves from one orientation to the next, both of which I suspect will be problematic.

14.5 Object-specific behavior

Alice design has been driven from the beginning by the desire to find a set of highly useful yet object-independent animation primitives for the support of interactive 3D graphics. Specifically, we have shied away from teaching our novice user community complex programming topics like subclassing and method overriding. While there is still much work to be done in finding the common set of animation primitives, it seems clear from conversations with our users that this is not what our users want. Our users want the energizer bunny to hop, not to move up, pause, and then move down.

To not explore the power of object-specific behavior is to condemn the Alice system to the same sort of "Turing Trap" that has plagued Alice's predecessors where everything is

possible, but nothing is easy. In Alice's case, we have managed to make the primitive things easy, but the common, everyday things (walking, flying, swimming, throwing, dancing, etc.) remain as hard as ever.

14.6 Object Shape Encoding

The shape of an object will often constrain the kinds of spatial relations that an object can have with other object. We see this when we look at the language people to talk about volumes (**inside** the jar), surfaces (**on** the lake), lines (**along** the track) or points (**closer** to the corner). The object-understanding literature describes this breakdown [Miller] noting that this is a powerful way of organizing our perception of objects.

When we break objects down into these categories, we can allow them to respond to some interesting messages and predicates:

	Possible Prepositions	Type Of Structure This Might Be
Volume	inside/outside left/right over/under front of/behind bounding box bounding sphere through	Any solid object voids and spaces (“the hallway”)
Planar	over/under along-and-towards along-and-away-from on towards/away perpendicular	named surface in the tree (e.g. chair seats) the "obvious" surface for those objects that are primarily planar like mirrors, plates, doors, wall
Linear	along away/toward, perpendicular to	the “generating axis” – longest dimension of an object named paths
Point	toward/away	Insertion point Center of bounding box Naml;med point in the tree “named points” on an object (see page 215)

I propose augmenting Alice objects with a shape classification, which would allow it to respond to the messages shown above. This shape classification could probably be made automatically in many cases by examining the rough geometric shape of the object.

14.7 Making Surface Names Intrinsic

The psychology literature notes that people perceive the left side of an object as being an inherent and permanent part of an object [Miller][Olson]. Given this observation, it might be worthwhile testing an API where the surface names (Front, Back, Left, Right, Top, Bottom) and two additional directional names (Forward, Up) are provided as attributes of the object, denoted no differently than the way parts are denoted now. This would yield an API that looks something like so:

```
Bunny.move(bunny.forward, 10)
```

Some advantages of this approach:

- It would remove the AsSeenBy keyword parameter without removing the functionality. To move in some other object's coordinate system, the programmer simply uses a direction name belonging to another object.

```
Bunny.move(Camera.forward, 1)
```

- It could remove the Place command as part of the API, folding its functionality into the moveTo command.

```
Cup.moveTo(table.top)
```

- It could generalize the PointAt and GetAGoodLookAt commands in powerful new ways with no new mechanisms: This could be the implementation of the Map and Elevation views mentioned earlier in this chapter.

```
Car.left.PointAt(Camera)
Camera.pointAt(Bunny.Top)
Camera.GetAGoodLookAt(Bunny.Left)
```

- Alignment commands become more expressive as well

```
Table.left.AlignWith(Chair.right)
```

It is interesting to note that in these examples, we are calling methods on direction names (Table.left) and not on objects. Of course, not all object methods can be applied to surface or direction names (what does it mean to call move on Table.left?) but this opens the question of implementing interesting operations that work with surfaces and directions. The table in section 14.10 hints at just a few such functions. On the whole, I believe there may be more advantages than disadvantages though there is the risk that this formulation favors the intermediate and advanced user at the expense of the novice. Only testing can tell us if this is true.

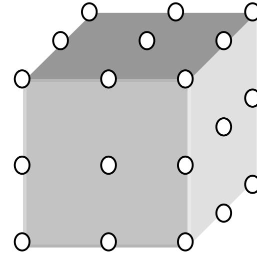
As a final note, note that if the API is designed to allow a shorthand

```
Bunny.move(forward)    ≡    bunny.move(bunny.forward)
```

Then I would argue that the new API is a strict superset of the old API and is therefore the complexity of the new API does not impact novice programmers.

14.8 Named Corners

Extending the idea of named faces (Front, Back, Left, Right, Top, Bottom), I propose that we implement the notion of named corners on objects. The names of these corners would be triples, derived from the names of the faces: (A, B, C) where



- A ∈ {front, center, back}
- B ∈ {left, center, right}
- C ∈ {top, center, bottom}

With the special alias:

- middle ≡ {center, center, center}

Which yields 27 new object-independent locations on objects. I would propose a shorthand where the token center could be elided under certain conditions, so long as there no ambiguity:

{Front, Center, Center} ≡ {Front, Center} ≡ {Front}

This would allow programmers more flexibility in the Place command as well as more choices in determining which point an object scales around under the resize command. Notably, it would allow programmers a relatively easy way to keep objects from "scaling through the floor" when they get larger, or hovering off the floor when they get smaller: simply Resize the object around the {Bottom, Center, Center} point.

14.9 Named Edges

In the same spirit as named corners, we can leverage the direction names yet again to allow programmers to refer to named edges of objects for purposes of specifying rotation axes or other vectors (for purposes of PointAt or PutBetween, or placing things along a path). Named edges would be formed out of pairs of direction names (e.g. {Front, Left}) similar to the way that the names of corners are constructed.

14.10 Summary of Geometric Operations

The preceding three sections dealt with volumes, planes, lines, and points. The table below gathers together the relationships between these objects and the ways in which many of the lower-dimensional objects can be combined to form members of the higher-dimensional set. I also introduce the notion of an Axis, which is a directed and rooted line (a rooted vector), and a Surface, which is a directed plane.

Element	Informal Description	Ways to Specify	
		Alice Structure	Functional Operators
Point	A location in space	An Alice object (origin) 27 named points	Intersection Of (Plane, Line) Intersection Of (Plane, Axis) MidpointOf(PointA, PointB) ProjectPoint(Point, Axis, Distance) ShadowOf(Point, Line)
Line	Undirected line segment	8 named edges Two Objects	PerpendicularTo(Plane) IntersectionOf(planeA, planeB)
Axis	Directed line segment	6 named object directions Obj.Up Obj.Down Obj.Left Obj.Right Obj.Forward Obj.Back	ShadowOf (Axis, Plane) AxisBetween(PointA, PointB) PerpendicularTo(Surface)
Plane	Undirected surface	6 named object surfaces Obj.Top Obj.Bottom Obj.Left Obj.Right Obj.Front Obj.Back	PlaneOf(PointA, PointB, PointC) PlaneOf(Point, Line)
Surface	A plane with an up direction or a topside or an "outside"	Ground Obj.Ceiling Obj.Floor Obj.RightWall Obj.LeftWall Obj.Front Obj.Back	

Note the rich grammar that results from this, where objects are points and the direction names and corner names can stand in for the arguments in the functional operators used to build up the higher-dimensional objects. ShadowOf is a geometric projection function.

14.11 Higher-Level APIs

Turning In Planes

Rotation is still too hard in Alice (page 156). Toward that end, I would propose that the surfaces and edges notation introduced above be used as the basis for a more powerful set of commands that control the rotation of objects.

Given that we have a way to refer to surfaces, we could support a command that allowed an object to turn left or right in a given plane:

```
Obj.TurnInPlane(direction, amt, plane)
```

As in

```
camera.TurnInPlane(left, 1, Ground)
```

If objects carried with them the notion of a supporting surface, as first proposed in [Houde], we could take the plane of rotation as being the “supporting surface”. This has the advantage that it might allow us to use the turn command alone in a great many more contexts, without the need to appeal to either a “turn in plane” command or a second

rotation command like roll quite as often. Promoting a supporting surface to a first-class Alice abstraction would also be in keeping with Alice's tendency to respecting the global up vector, which recognizes the importance that gravity plays in our understanding of space.

Another powerful rotation abstraction missing from the Alice API is the notion of turning toward or away from something by some number of degrees.

```
Obj.TurnToward(other_object, amt)
```

Like the PointAt command, TurnToward is, strictly speaking, underconstrained, but can be given precise semantics by using the ground plane (in general, a resting plane) as the plane in which to rotate and always rotating the object through the smallest angle (at least by default).

Orbiting Other Objects

Making one object rotate around another is a common operation, but one that Alice supports only imperfectly. Currently in Alice, programmers are encouraged to use the AsSeenBy keyword to specify the other object to turn around:

```
Horse.turn(left, 1, AsSeenBy=carousel)
```

Unfortunately, this has some drawbacks:

If the orbiter ends up rotating around a moving object, the orbiter will end up in a strange orientation with respect to the object it was rotating around. This comes about

because the orbiter does not counter-rotate during the orbit so as to keep a constant relative orientation. This wouldn't happen if the orbiter was made a part or a child of the other object, but that can be a heavy-handed solution.

Another usability problem with rotation and AsSeenBy is that the direction of motion is taken from the object being orbited, which can be hard to visualize and hard to specify.

In short, while we have a mechanism that causes one object to rotate around another, it is hard to control, has strange behavior in some common circumstances and does not match the semantics that we often need.

To overcome these difficulties, I would propose the implementation of a set of specialized Orbit commands that, while much more specialized than a general rotation command, together these commands capture an interesting subset of useful behaviors.

All forms of Orbit follow the same general steps:

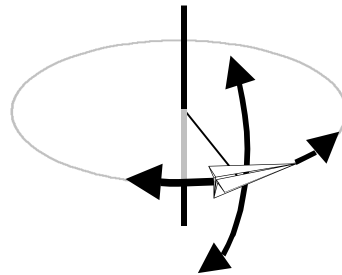
1. Point one face of the orbiter at the orbited object
2. (Optionally) Parent the orbiter to the orbited object
3. Allow the orbiter to move in four of the six possible directions of motion.

Note that this last step has the advantage that it allows the object to move in a local coordinate system, in direction names that relate to the moving object, not to the object being orbited, as is currently the case. The hope is that this will restore some Logo-like simplicity to the act of rotating in another object's coordinate system.

Different orbiting styles are:

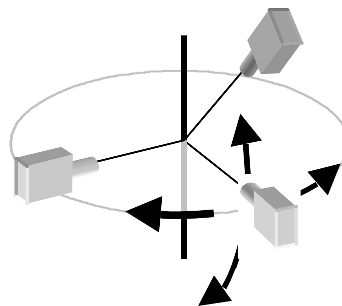
Carousel Orbit

- Orbiter face pointing toward center : Left or Right
- Orbiter moves: Forward, Back, Up, Down



Examine

- Orbiter face pointing toward center : Front
- Orbiter moves: Left, Right, Up, Down



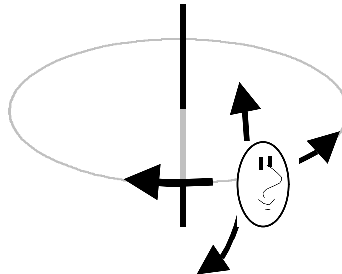
DriveAround

- Orbiter face pointing toward center : Bottom
- Orbiter moves: Forward, Back, Left, Right

Shun

- Orbiter face pointing toward center : Back

- Orbiter moves: Left, Right, Up, Down



Shuttle Orbit[1]

- Orbiter face pointing toward center : Top
- Orbiter moves: Forward, Back, Left, Right

Lights API

The current API for lights still needs a great deal of work. The current set of methods on lights include calls that control the constant, linear and quadratic attenuation factors of the lights – parameters which control the degree to which the light is capable of illuminating distant objects. This is present only because the underlying rendering libraries expose this functionality, though it seems clear that even expert programmers have a hard time controlling these parameters to good effect. The Alice development team has not paid a great deal of attention to this problem in the past aside from setting the lights to reasonable defaults and hoping that this is good enough for most applications (which it appears to be). [2]

---

1 An admittedly strange name, inspired by the fact the US Space shuttle often flies with its top pointed toward the Earth.

2 Borrowing a trick from Disney Imagineering, Alice configures textured objects as being “unlit” by default, and given that nearly all objects in Alice carry texture maps, it turns out that even the one default light in Alice does not illuminate anything.

Path API

A Path is just a named series of points in space or an ordered set of linear and arc-shaped segments that can be used to constrain object motion. Real-world paths are a vital part of understanding space and navigating space; there is a great body of literature that describes how people form and use paths[Miller][Darken][Lynch]. Alice currently has no built-in facility for creating paths, but some early prototypes and the creation of a roller-coaster both used ad-hoc path capabilities to good effect. Paths constrain object motion in interesting ways, allowing users to talk about moving forward and backward along complex routes, without having to manage the details of the path itself, which can be pre-authored or automatically generated by the system based on the objects that exist in the 3D field. Paths might also be an interesting way to implement the Orbit commands, orbits being just a circular path of some radius centered on some other object.

14.12 Drawing the Invisible: Representing Non-Graphical State

One of the most dangerous and seductive things about 3D graphics is that it lulls the designer into thinking that the most important parts of a 3D graphical simulation are graphical. This is often not the case.

Every graphical object has associated with it a host of “invisible” information that the user might be interested in seeing. This information is typically textual or abstract in

nature. Sometimes such information does not have an obvious graphical representation and often it is not well suited to being rendered as a graphical part of the simulation. As a concrete example of this kind of information, I would point to the Alice treeview that displays the parent-child relationships of the object tree. This screen element displays abstract information about the 3D scene (object names, parent/child relations, first-class/part state) and displays it in a textual/graphical form that can be inspected and manipulated directly. It takes the invisible and makes it visible. Alice needs to do more of this. Alice scripts maintain a great deal of this invisible state, but the runtime environment often does not make this state visible on the screen in a way that can be inspected or manipulated by script developers. This makes Alice a harder environment than it should be or needs to be.

Things That Are Invisible That Should Not Be Or Do Not Need To Be:

Object directions – Alice objects currently do not have a way of displaying which of their faces is their forward direction. The capability of showing coordinate axes on an object is one that earlier versions of Alice had, but ironically later versions lost. We should bring this critical feature back to Alice.

Mouse and Keyboard reaction functions that are currently active for a given object. Editing capability is important here. Also helpful would be a list of mouse modes in the form of presets that would make it trivial to set up an Alice simulation for "flight simulator" mode or "gaze-to-fly" mode.

List of Animations currently running on a per-object basis. The challenge here is making sure that the functions displayed accurately reflect the code that the user executed. Often times the code that is being evaluated by the simulation loop is not exactly the same as the code that the user executed from script. For example, a PointAt command is implemented in terms of a TurnTo command. We must maintain the illusion that the simulator is executing a PointAt if we intend to show the user a list of running animations.

The set of possible actions that an object can do. We should have an exhaustive and extensible mechanism for doing this, not the ad-hoc approach we have now.

Bookmarks – the user should be allowed to express the notion of "mark this position and orientation and give it the string name S." Alice users often do this themselves, but it is all done in code and the list is often invisible

What the Undo button does when you press it. Currently, there is no way to tell what the next press of Undo actually does. We should display this to the user.

History of commands recently executed. Alice currently has a drop-down box of commands, accessed through a standard Windows 95 "combo box". Currently, this list contains a non-repeating set of commands (executing a command multiple times puts it in the history only once), making it useless as a trace of past activity. A visible and complete display showing all past commands could be very helpful in letting Alice users see what happened in the past and what the Undo button will do next when pressed. Mouse and

keyboard gestures should also be captured in this window in a way that is easy to see, with the option of letting the user undo items with or without undoing the mouse motion of the objects or of the camera.

14.13 A Summary of Open Questions

- Is roll really different? (page 164)
- Should Up be taken as a Ground Up by default? (page 150)
- Should we re-parent and promote objects to first-class during interactive disassembly ? (page 128)
- Are there better defaults for visibility and simulation? (page 192)
- Are there more powerful relationship types than “is parent of/is part of”? (page 125) What of relations like attached-to, pivots-around, and is-supported-by – what are the relationships that matter and how do they conflict and combine?
- What happens to the camera when a new object is created in the scene? (page 204) Currently, when the user creates a new object, Alice neither moves nor rotates the camera, nor do the lens parameters of the camera change. As a result, adding a new object to a simulation at scene construction time can put the object inside the camera, behind the camera, at a point very distant from the camera, or, perhaps worse of all, on top of the camera, which makes the object is invisible due to backface culling.
- Should the resize operation have the side-effect of *moving* first-class sub-objects? (page 178)

Chapter 15
Previous Work

15.1 Programming Systems For Novices

Perhaps the work that most closely resembles Alice is Papert's LOGO work [Papert]. Papert was interested in getting children to program computers, which placed some very strict learnability requirements on his system. In the course of his research, he discovered the power of allowing grade school children to "act out" graphical programming tasks, essentially letting the children assume the role of the cursor, or "turtle." This strategy of shifting the programming task from a third-person (exocentric) to a first-person (egocentric) perspective allowed the children to take advantage of knowledge they already had about how their own bodies move in space. Using an ego-centric coordinate system helps small children reason about drawing circles so effectively that they can often outperform older children attempting to write the same "draw circle" routine using trigonometry. Papert also discovered there were certain concepts in programming (procedures, variables and the importance of debugging) that gave young programmers more trouble than any of the other

ideas in LOGO.

Alan Kay's and Adele Goldberg's experiences with Smalltalk [Goldberg] were similar to Papert's with LOGO; Kay showed that children were able to engage in far more advanced programming tasks than the experts thought possible, provided that the tools and pedagogy supported exploration without penalty, and the children were sufficiently motivated.

HyperCard is widely used as an authoring tool by non-programmers in a wide variety of settings on the Macintosh platform. On-screen tools were used in HyperCard to obviate a lot of programming, and when users found that they needed to write a script that made a decision of some kind, they became highly motivated to learn the "if" construct provided in the scripting language. Apple engineers designed HyperCard in a way that allowed novice programmers to go relatively far without having to appeal to explicit flow-of-control issues, by burying the lion's share of the logic in the widgets. Alice takes a similar strategy by showing users that a great deal of scripting can be done with simple scripts that don't carry if statements, burying the logic inside the `RespondTo` command.

Microsoft's Visual Basic has also shown that high-level graphical tools can ease some of the programming burden while at the same time exposing the details in a controlled manner through the display of menus and popup lists of available choices. These tools can help keep the programmer's mind on the task at hand and not on "driving" the tool.

SUIT [Pausch], the Simple User Interface Toolkit from the University of Virginia,

also heavily influences the Alice design. SUIT's primary goal was to simplify the creation of 2D widget-based user interfaces by distilling out the most important abstractions in widget programming, and presenting them in a powerful authoring and runtime inspection tool. Part of SUIT's design was that it be learnable in under two hours, with most of that time spent with a carefully crafted 10 page tutorial. Usability testing of the system was instrumental in obtaining these goals; the SUIT team watched hundreds of undergraduates interact with the system, in order to make the system easier to learn and use. Part of SUIT's success was due to the SUIT Property Editor, a control panel that controlled the display and manipulation of all the visual and behavioral properties of all the objects in the system. The SUIT team designed the Property Editor so as to carefully expose SUIT's underlying three-level property inheritance model. In this way, the on-screen tools helped to explain the SUIT system itself. The Alice system follows much of the same methodology as used in the SUIT project: Alice relies on feedback from the user community to help identify the problem areas of the toolkit, and uses the system tools and tutorials to teach the hardest parts of the underlying model.

15.2 Graphical Systems For Exploratory Programming

Also close in spirit to Alice is Bolio [Zeltzer]. Bolio was a time-based animation system for programming interactive graphics simulations in three dimensions. Bolio allowed for "task-level" animations where the user declared what was to happen, and the runtime system would determine how the request was to be met. Bolio also supported one-way,

loop-free constraints and a simple kinematics model for simulating simple physical processes.

Randy Smith's Alternate Reality Kit (ARK) [Smith 86][Smith 87] was a graphical simulator system that allowed both application writers and application users to interact with a graphical depiction of dynamic two-dimensional objects. Using ARK, Smith explored ways in which powerful graphical metaphors, strictly adhered to, can make a system easy to learn, even if lacking in power. Smith's primary interest was in examining the tension in user interface metaphors between what he termed magic (being able to do powerful, if somewhat unexpected things, given the visual metaphor presented by the software) and literalism (staying close to the visual metaphor at the expense of power). ARK showed that literalism in an interface improves the usability of a system.

Dave Ungar, Randy Smith, Bay-Wei Chang and others were responsible for a programming language called Self [Ungar] that featured a graphical user interface programming environment. Even though Self is a traditional textual programming language, the GUI is considered an integral part of the development environment, and features advanced cartoon-inspired animation [Chang] entirely for the purposes of making the environment engaging, entertaining and compelling to the user. The cartoon animation facilities which were inspired by Disney and [Lasseter] were also inspirational to the Alice system development, even though (unlike Self) Alice uses animation mostly in the 3D rendering window, rarely using it in the 2D GUI/programming environment.

Paul Strauss's system BAGS (Brown Animation Generation System) [Strauss88], was one of the first interactive 3D systems to use an interpreted language to describe the static layout and dynamic behavior of a 3D scene. The language, SCEFO, was not quite a full programming language in that it lacked a conditional operator, though, to its credit, it was extensible in C. Alice also uses an interpreted, extensible language, though we chose to use an off-the-shelf, full-fledged programming language instead of inventing a new language.

15.3 3D Graphics APIs

The need for increased usability isn't something restricted to novices; even experts perceive the need for higher level abstractions, leading to the development of 3D graphics toolkits like Inventor [Strauss92]. Inventor implements a hierarchical display list that controls both visual appearance and dynamic behavior under a single tree structure. The developers of Inventor had hoped that the system would be easier to use than the SGI's native GL library for the creation of 3D direct manipulation applications and even higher level 3D toolkits and to that extent, they were successful.

Open GL [OGL] is an open standard API intended to standardize the SGI's popular GL library while at the same time removing some of the hardware-specific elements from the API. In the process of standardizing GL, the OpenGL group also simplified some of the trickier parts of the GL API, but always with the intention that the library be used by seasoned programmers, never simplifying it with the novice programmer in mind.

Mark Najork's Obliq 3D [Najork] is a system that is very close to Alice in spirit, and not surprisingly, the two systems share many similarities. Both systems use an interpreted scripting language (Obliq3D uses a variant of Cardelli's Obliq programming language), both have facilities for divorcing the simulation frame rate from the rendering frame rate, and both use wall-clock time as their primary means for specifying behavior. Where Obliq3D differs is in its target audience. Even though Obliq3D is designed to make 3D scripting easier to use (and was largely successful in that regard), Najork's system does not target young, non-technical adults as the primary user community, but chooses to cater to the needs of experienced programmers. The result is a system that is elegant and functional, but still too hard for the uninitiated.

Finally, TBAG [Elliot] is a programming system that uses time and constraints as the primary structuring mechanisms for the specification of behavior. At TBAG's center is a constraint engine that drives animations forward by constraining object property values. The system is extremely clean and elegant for those animations that are time-dependent but can be somewhat awkward when dealing with animations that are not easily expressed entirely in time-dependent ways.

15.4 Graphical World Builders

Concurrent with the Alice research effort, there have been commercial ventures such as Sense8's World Toolkit, Sense8's WorldUp [Sense8], and [Superscape], all of which are meant to be CAD-like 3D world editors, complete with scripting languages and advanced

geometric modeling capabilities. Of these, Sense8's WorldUp seems to be the one closest in spirit to Alice, sporting a scripting language (a variant of Visual Basic) and a powerful runtime environment. Unlike Alice, the scripts in WorldUp are attached to the objects themselves, and not centrally located. This has the advantage that code snippets for a single object tend to remain small and tractable, but that inter-object behavior can be hard to understand and to debug. Fully understanding a pre-authored world in WorldUp can also be difficult, as the logic that drives the runtime behavior is distributed throughout all the objects. In this sense, WorldUp is very similar to traditional 2D GUI builders, in that it depends heavily on an external control model for program control. WorldUp also does not support a built-in animation model, forcing the user to perform time-based animations by hand by examining the system clock and moving objects incrementally.

Chapter 16
Conclusions

16.1 Contributions

This work presents several ideas which are new to the user interface and interactive graphics research communities:

Cognitive load is reduced on extremely common operations through the removal of X,Y,Z from the API, replacing these terms with the more useful and more Logo-like direction names and surface names of Forward/Back, Left/Right, Up/Down. This idea is new for interactive 3D.

APIs can be simplified and improved using user observation, a well-known technique for GUIs but a novel concept for application programmer interfaces. There are certainly subtleties in API testing that do not exist for GUI testing, mostly due to the highly cognitive nature of programming. More work can clearly be done in researching the best ways to test and improve APIs.

Transformation of coordinate systems is made easily available through a keyword parameter (AsSeenBy) without burdening the programmer with the implementation details.

First-Class objects vs. parts allow programmers to bring more semantically meaningful behavior to objects.

Implicit threads in Alice make launching parallel actions easy.

Making objects larger or smaller should not change their sense of space. The resize operation and the space scaling operation are both useful, but are independent and orthogonal, even if using a 4x4 matrix in the implementation makes this separation difficult to build.

Crayola names for colors allow us to cover a wide color space using familiar color names.

Controlled exposure to power is a new idea for APIs, being inspired by a known concept in GUI design. This API feature is easily implemented in languages that support keyword parameters, default values for function and method arguments and polymorphism.

Elevating common animation options such as distance, rate, acceleration, interpolation, and change of coordinate system into the list of optional arguments in many parts of the API so as to make them more easily accessible.

16.2 Known Issues

This research also underscored several issues that were known in the user interface, spatial perception, and systems design communities:

- Perceptual constancy: all operations animate with slow In / Slow Out, a fact emphasized in the Self programming language development environment [Ungar]
- Provide undo for everything
- An interpreted language facilitates a rapid turnaround time between changes, which is essential for exploratory programming and rapid prototyping
- 3D systems should respect the global up direction, noting that it organizes the way humans perceive and understand 3D space.
- Set the defaults for the novice, not for the expert.
- A case insensitive programming language is essential for novices
- Tasteful and careful word choice is essential in supporting label following and in leveraging real-world knowledge that users bring to the task. Some Alice examples:

Good Term	Bad Term
Speed	Rate
Visibility	Opacity/Transparancy
Move (Slide)	Translate
Revolutions per second	Degrees per second
Tilt	Roll

16.3 What Remains Hard

- rotation is still poorly abstracted
- creation of new animations requires a decomposition skill that is hard to acquire
- external control model is hard to explain, just as it was in 2D GUI toolkits

Chapter 17
References

[Bainbridge] L. Bainbridge, *Verbal reports as Evidence of the Operator's Knowledge.* International Journal of Man-Machine Studies, 11, 411.

[Ball] J. Eugene Ball, Daniel T. Ling, David Pugh, Tim Skelly, Andrew Stankosky, David Thiel *Reactor Technical Report* MSR-TR-93-18 ftp://ftp.research.microsoft.com/pub/tr/tr-93-18.rtf

[Bajcsy] R. Bajcsy and L. Lieberman. *Texture gradient as a depth cue*. Computer Graphics and Image Processing, 5:52-67, 1976.

[Bloom] Paul A. Bloom, Mary A. Peterson, Lynn Nadel, Merrill F. Garrett, editors, *Language and Space*, MIT Press, 1996.

[Borenstein] Nathaniel S. Borenstein. *Programming As If People Mattered. Friendly Programs, Software Engineering, and Other Noble Delusions*. Princeton University Press, 1992.

[Bowman] Doug A. Bowman, David Koller, Larry F. Hodges, *Travel in Immersive Virtual Environments: An Evaluation of Viewpoint Motion Control Techniques*, Proceedings of the Virtual Reality Annual International Symposium (VRAIS), 1997, pp. 45-52.

[Bricken] William Bricken and Geoffrey Coco. *The VEOS project*. Presence Teleoperators and Virtual Environments, 3(2):111-129. The MIT Press, 1994.

[Brooks75] Frederick P. Brooks, *Mythical Man-Month*, Addison-Wesley, 1975.

[Brooks88] Frederick P. Brooks, *Grasping Reality Through Illusion: Interactive Graphics Serving Science*.ACM SIGCHI 88 Conference Proceedings, pp. 1-11.

[Card] S. K. Card, George Robertson, Jock Mackinlay. *The Information Visualizer, an Information Workspace*. ACM SIGCHI 91 Conference Proceedings, 1991, pp. 181-188.

[Chang] Bay-Wei Chang, David Ungar, *Animation: From Cartoons to the User Interface*, UIST 93: User Interface Software and Technology Conference Proceedings, pp. 45-55.

[Clarke] James H. Clarke, *Hierarchical Geometric Models for Visible Surface Algorithms*, Communications of the ACM, 19(10), October 1976, pp. 547-554.

[Clay] Sharon Rose Clay, Jane Wilhelms, *Put: Language-Based Interactive Manipulation of Objects.* IEEE Computer Graphics and Applications, March 1996. Vol 16, Number 2, pp. 31-39.

[Conway] Matthew Conway, *Using Tk From Other Programming Languages,* ACM Interactions, April 1995.

[Corballis] M. C. Corballis, *The Nature of the Left-Right Coding*. Paper presented at the annual meeting of the American Psychological Association, Los Angeles, 1981.

[Darken] Rudy Darken, John L. Siebert, *Wayfinding Strategies and Behaviors in Large Virtual Worlds*, ACM SIGGRAPH 96, Conference Proceedings.

[di Sessa] A.A. di Sessa, H. Abelson, “Boxer: A Reconstructible Computational Medium”. *Studying the Novice Programmer,* E. Soloway and J.C. Spoher, Hillsdale NJ. 1989.

[Dunn] B.E. Dunn, G.C. Gray, D Thompson. Relative height on the picture plane and Depth Perception. Perceptual and Motor Skills, Vol 21, pp. 227-236.

[Eisenhart] Douglas M. Eisenhart *Publishing in the Information Age.* 1994, Quorom Books, Westport CT.

[Elliot] Conal Elliot, Greg Schecter, Ricky Yeung, Salim Abi-Ezzi. TBAG: A High-Level Framework for Interactive Animated 3D Graphics Applications. ACM SIGGRAPH 94. Conference Proceedings, pp. 421-434.

[Fitts] Paul M. Fitts, R.E. Jones, *Pychological Aspects of Instrument Display. I: Analysis of 270 “Pilot Error” Experiences in Reading and Interpreting Aircraft Instruments,* Memorandum Report TSEAA-694-12A, Aero Medical Labaroatory, Air Materiel Command, Wright Patterson Air Force Base, Dayton, Ohio, October 1, 1947, pp. 47.

[Foley] J. D. Foley, A, van Dam, S.K. Fiener, J.F. Hughes, *Fundamentals of Interactive Computer Graphics,* Addison-Wesley, Reading, MA, 1990.

[Franzke] M. Franzke *Turning research into practice: characteristics of display-based interaction,* ACM CHI'95 Human Factors in Computer Systems, Conference Proceedings 1995, pp. 421-428.

[Furnas] G.W. Furnas, T.K. Landauer, L.M. Gomez, S.T. Dumais, T*he Vocabulary Problem in Human-System Communication*, Communications of the ACM, November 1987, Vol. 30, Number, 11, pp. 964-971.

[Goldberg] Adele Goldberg, and David Robson, *Smalltalk80: The Language*, Addison-Wesley, Reading, MA, 1989.

[Gossweiler] Rich Gossweiler, Chris Long, Shuichi Koga, Randy Pausch, *DIVER: A Distributed Virtual Environment Research Platform,* IEEE Symposium on Research Frontiers in Virtual Reality, October 25-26, 1993, San Jose, CA, pp. 10-15.

[Green] T.R.G. Green, “The Nature of Programming”, *Psychology of Programming*, J.M. Hoc, T.R.G.Green, R. Samurcay, J.D. Gilmore, London Academiv Press, 1990.

[Hauptman] Alexander G. Hauptmann , *Speech And Gestures For Graphic Image Manipulation.* Proceedings of the ACM SIGCHI Human Factors in Computer Systems Conference, 1989. pp. 241-246.

[Houde] Stephanie Houde, *Iterative Design of an Interface for Easy 3D Direct Manipulation,* ACM Computer Human Interaction Conference 92, Conference Proceedings, 1992, p. 135-141.

[Karney] *Chronicle of the Cinema,* Editor Robyn Karney, Chronik Verlag. 1995.

[Knep] Brian Knep, Craig Hayes, Rick Sayre, Tom Williams. *Dinosaur Input Device* Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems. 1995. Pp. 304-309. http://www.acm.org/sigchi/chi95/proceedings/papers/bk_bdy.htm

[Kosslyn] Stephen Kosslyn, *Image and Mind*, Harvard University Press, Cambridge, Mass, 1980.

[Lasseter] John Lasseter, *Principles of Traditional Animation Applied to 3D Computer Animation*, SIGGRAPH 87 Conference Proceedings, pp. 35-44.

[Levelt] Willem J. M. Levelt, “Perspective Taking and Ellipsis in Spatial Descriptions” from *Language and Space*, edited by Paul Bloom et.al, pp. 77-109. Massachusetts Institute of Technology, 1996.

[Levinson] Stephen C. Levinson, “Frames of Reference and Molyneux’s Question: Crosslinguistic Evidence”, from *Language and Space*, edited by Paul Bloom et.al, pp. 109-170. Massachusetts Institute of Technology, 1996.

[Logan] Gordon D. Logan, Daniel Sadler, *A Computational Analysis of the Apprehension of Spatial Relations*, in Language and Space, edited by Paul Bloom, et. al. MIT Press, 1996.

[Lutz] Mark Lutz, *Programming Python.* O'Reilly & Associates ISBN: 1-56592-197-6 October, 1996.

[Lyons] J. Lyons, *Introduction to Theoretical Linguistics,* Cambridge Press, 1968.

[Lynch] K. Lynch, *Image of the City,* Caimbridge MIT Press, 1960.

[Mackinlay] Jock D. Mackinlay, Stuart K. Card, George G. Robertson, *Rapid Controlled Movement Through a 3D Virtual Workspace*, ACM SIGGRAPH 1990, Conference Proceedings, pp 171-179.

[Miller] George A Miller, Phillip N. Johnson-Laird. *Language and Perception.* Harvard University Press, 1976.

[Mine] Mark Mine *ISAAC: A Virtual Environment Tool for the Interactive Construction of Virtual Worlds.*, 1995 . UNC Chapel Hill Computer Science Technical Report TR95-020.

[Najork] Mark Najork, *Obiq-3D Tutorial and Reference Manual,* Digital Equipment Corporation, Systems Research Center, (DEC SRC) Research Report #129, December 1, 1994.

[Neilsen] Jacob Neilsen, *Usability Engineering*, Academic Press, Boston, 1993.

[OGL] Open GL Architecture Review Board, *The OpenGL Reference Manual*, Addison-Wesley, 1992.

[Olsen] Dan Olsen, Jr. *A Programming Language Basis for User Interface Management,* ACM SIGCHI 89 Conference Proceedings, Conference on Human Factors in Computing Systems, 1989 pp. 171-176.

[Olson] David R. Olson and Ellen Bailystok, *Spatial Cognition: The Structure and Development of Mental Representations of Spatial Relations*, Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1983.

[O'Malley] C. E. O'Malley, S. W. Draper, M. S. Riley. *Constructive Interaction: A method for Studying Human-Computer-Human Interaction*. Proceedings of IFIP INTERACT `84 First International conference on Human-Computer Interaction. London U.K. 4-7 September, 269-274.

[Pausch92] Randy Pausch, Matthew Conway, Robert DeLine, *Lessons Learned from SUIT, the Simple User Interface Toolkit,* ACM Transactions on Office Information Systems October 1992, 10:4, pp. 320-344.

[Pausch95] Randy Pausch, Tommy Burnette, Dan Brockway, Michael Weiblen, *Navigation and Locomotion in Virtual Worlds via Hand-Held Miniature*s, ACM SIGGRAPH 95, Conference Proceedings, pp 399-400.

[Pausch96] Randy Pausch, Jon Snoddy, Robert Taylor, Scott Watson, Eric Haseltine, *Disney's Aladdin: First Steps Toward Storytelling in Virtual Reality,* ACM SIGGRAPH 96 Conference Proceedings, August 1996.

[Papert] Seymour Papert, *MindStorms: Children, Computers, and Powerful Ideas*, Basic Books, New York, 1980.

[Pierce] Jeffrey Pierce, Andrew Forsberg, Matthew Conway, Seung Hong, and Robert Zeleznik *Image Plane Interaction Techniques in 3D Immersive Environments,*, 1997 Symposium on Interactive 3D Graphics.

[Potegal] Michael Potegal, *Spatial Abilities, Development and Physiological Foundations*, Academic Press, New York, 1982.

[Robertson] G.G. Robertson, S.K. Card, J.D. Mackinlay. *The cognitive coprocessor architecture for interactive user interfaces*. In ACM Symposium on User Interface Software and Technology (Nov. 13-15, Williamsburg, VA), ACM/SIGGRAPH/SIGCHI, 1989, pp. 10-18.

[Sense8] Sense8 Corporation: www.sense8.com

[Shoemake] Ken Shoemake, *Animating Rotation with Quaternion Curves*, ACM SIGGRAPH 85, Conference Proceedings, 1985.

[Smith 86] Randall B. Smith, *The Alternate Reality Kit: An Animated Environment for the Creation of Interactive Simulations*. Proceedings of the 1986 IEEE Computer Society Workshop on Visual Languages, 1986, 99-106.

[Smith 87] Randall B. Smith, *Experiences with the Alternate Reality Kit: An Example of the Tension Between Literalism and Magic*, ACM SIGCHI 1987, Conference Proceedings, pp. 311-317.

[Steele] Guy L Steele Jr. *Common Lisp, The Language*. Second Edition. 1990 Digital Equipment Corporation.

[Strauss88] Paul Strauss, *BAGS: The Brown Animation Generation System,* Technical Report No. CS-88-22, Brown University, May 1988.

[Strauss92] Paul Strauss, Rick Carey, *An Object-Oriented 3D Graphics Toolkit*, ACM SIGGRAPH 92. Conference Proceedings, pp 341-349.

[Stoakley] Richard Stoakley, Matthew Conway, Randy Pausch, *Virtual Reality on a WIM: Interactive Worlds in Miniature,* ACM CHI'95 Conference on Human Factors in Computing Systems, 1995.

[Sutherland] Ivan Sutherland, *The Ultimate Display*, Information Processing 1965, IFIP Congress 65, 506-508.

[Thomas] Frank Thomas, Ollie Johnston, *Disney Animation – The Illusion of Life*, Abbeville Press, New York, 1981.

[Ungar] David Ungar, Randy Smith, *SELF: The Power of Simplicity,* OOPSLA 87, Conference Proceedings, published as SIGPLAN Notices, Volume 22, Number 12, 1987, pp. 227-241.

[vanDam] Andries vanDam, et. al. *PHIGS+ Functional Description Revision 3.0*, Computer Graphics 22, 3, (July 1988), 124-218.

[van Rossum] Guido van Rossum and Jelke de Boer, "Interactively Testing Remote Servers Using the Python Programming Language", *CWI Quarterly*, Volume 4, Issue 4 (December 1991), Amsterdam, pp 283-303. For more information on Python, see http://www.python.org

[Wexelblat] Alan Wexelblat, *Natural Gesture at the Under Interface*, ACM CHI 94, Conference Proceedings.

[Wixon] Dennis Wixon, *Qualitative Research Methods in Design and Development*, Interactions, Volume 2, Number 4, October 1995, pp 19-26.

[Zeleznik] Robert C. Zeleznik, Kenneth P. Herndon and John F. Hughes, *SKETCH: An Interface for Sketching 3D Scenes*, SIGGRAPH 96 Conference Proceedings , pp. 163-170.

[Zeltzer] David Zeltzer, Steve Pieper, David J. Sturman, *An Integrated Graphical Simulation Platform*, Graphics Interface 89 Conference Proceedings, pp. 266-274.