Alice:
Lessons Learned from Building a 3D System For Novices

Matthew Conway[1], Steve Audia[2], Tommy Burnette[2], Dennis Cosgrove[3], Kevin Christiansen[3], Rob Deline[3], Jim Durbin[2], Rich Gossweiler[2], Shuichi Koga[2], Chris Long[2], Beth Mallory[2], Steve Miale[2], Kristen Monkaitis[2], James Patten[2], Jeff Pierce[3], Joe Shochet[2], David Staack[2], Brian Stearns[3], Richard Stoakley[2], Chris Sturgill[3], John Viega[2], Jeff White[2], George Williams[2], Randy Pausch[3]

(1) Work done at the University of Virginia
mconway@microsoft.com

(2) University of Virginia

(3) Carnegie Mellon University
http://www.alice.org

ABSTRACT

We present lessons learned from developing Alice, a 3D graphics programming environment designed for undergraduates with no 3D graphics or programming experience. Alice is a Windows 95/NT tool for describing the time-based and interactive *behavior* of 3D objects, not a CAD tool for creating object geometry. Our observations and conclusions come from formal and informal observations of hundreds of users. Primary results include the use of LOGO-style egocentric coordinate systems, the use of arbitrary objects as lightweight coordinate systems, the launching of implicit threads of execution, extensive function overloading for a small set of commands, the careful choice of command names, and the ubiquitous use of animation and undo.

Keywords

Interactive 3D graphics, animation authoring tools

INTRODUCTION

Realtime 3D graphics is becoming mainstream: most PCs shipped in 1999 will ship with some sort of 3D graphics accelerator. We see this as an opportunity to approach 3D graphics research not as a question of rendering speed, but as one of *authoring and pedagogy*. Our goal is to engineer authoring systems for interactive 3D graphics that will allow a broader audience of end-users to create 3D interactive content without specialized 3D graphics training. Implicit in this line of research are a few assumptions:

The New Audience Assumption: we believe that a larger and more diverse audience will be interested in creating interactive 3D content. It is critical to realize that this new audience will not necessarily have the mathematical or programming background that current graphics programmers have; this shapes the nature of the tools that we must provide to this audience.

To Appear in
CHI '00 Proceedings

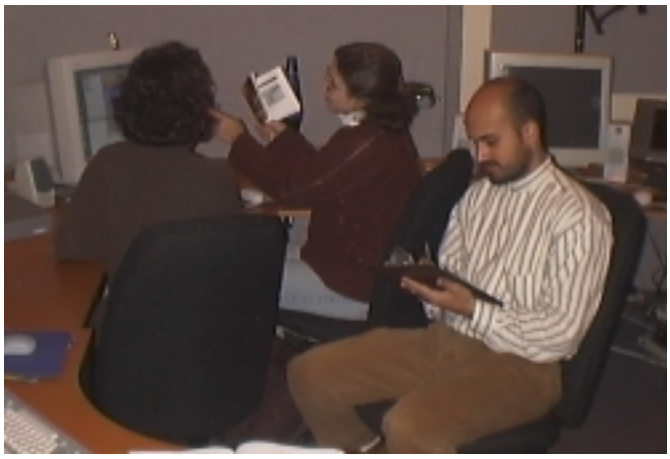


Figure 1 Observing Real Subjects. Two subjects learn the Alice 3D authoring environment while an observer silently sits behind them and takes notes. The major findings of this paper were derived from observations of 100 subjects taken over several months.

The Programming Assumption: Interesting interactive 3D graphics authoring will still involve some level of logic specification/programming, at least in the near term. This is true in part because of conditional behavior, which implies the need for some sort of "if-then" construct. We have focused on scripting in this work; future systems will probably use a combination of techniques including keyframing, programming-by-demonstration, and visual programming, as well as scripting.

We began the Alice research project with the goal of creating new authoring tools that would make 3D graphics accessible to this wider audience, something that current 3D tools would not do. Our basic design principles are:

- Choose a target audience and keep their needs in mind, in our case, non-science/engineering undergraduates.
- Avoid math and cryptic notation in the API (e.g. vectors, matrices) wherever possible and introduce new terminology only when needed.
- Iteratively test our designs with real users, improving both learnability and usability of the system in the process.

We state our findings as empirical research results, not as opinions; they are supported by formal observations of 100 users and hundreds of informal observation sessions over a four year period.

THE ALICE WORKFLOW

Authoring in Alice consists of two phases: *creating an opening scene* and *scripting*. This same two-phase workflow is seen in some commercial tools, including Raydream Studio and WorldUp.

Creating an Opening Scene

Users select objects from an object gallery displayed by clicking the *add object* button (figure 2, A). Alice's library contains hundreds of low-polygon models whose high fidelity comes from carefully hand-painted texture maps [15]. Note that while Alice users can import objects in several popular file formats, Alice itself is not a CAD tool for creating object geometry: Alice is a tool for describing the time-based and interactive *behavior* of 3D objects.

Objects are placed in a PHIGS-like tree of nested objects [2][23] (figure 2, B), displayed along the left edge of the authoring window. Navigation tools (figure 2, C) provide a simple walking metaphor for moving the camera. Alice can support multiple, simultaneous windows/cameras.

Context Menus

Holding down the right hand mouse button over an object displays a *context menu* of common operations, 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)

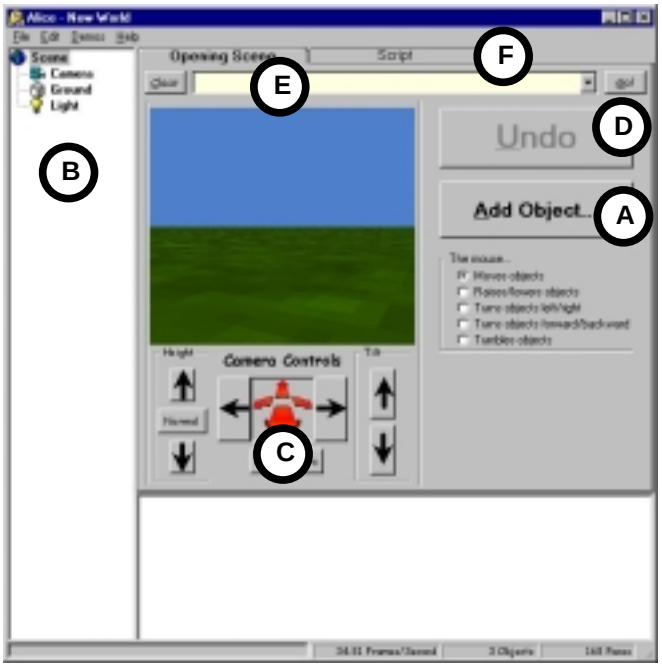


Figure 2: The Alice Authoring Environment (opening scene tab). **(A)** The Add Object button presents a gallery of 3D objects. **(B)** The Object Tree, a PHIGS-like tree of hierarchical objects **(C)** Camera controls allow the user to drive around the scene. **(D)** The Undo button provides infinite animated undo. **(E)** The Alice Command Box for evaluating single lines of Alice script. **(F)** The Script tab reveals a simple text editor where the user writes scripts that control the objects in the scene.

The Alice Command Box

Alice programmers are encouraged to explore the command set via the Alice Command Box (figure 2, E) which is used for trying individual lines of script code. If we wanted to move an object, a bunny for example, a precise distance (as opposed to using the mouse), we could type:

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

and press the GO button (or hit the Enter key) to make the bunny move up by one meter over a period of one second.

Commands are Always Animated

All commands in Alice animate over a period of one second with an Ease-In/Ease-Out interpolation [8] whenever it is semantically reasonable to do so. Programmers can still specify an explicit duration (including zero duration) if so desired. This is not just a flashy trick but is a *critically* important design decision. Not only does animation support the percept of object constancy [17], but it can also aid in the debugging process by providing information about how a bug unfolded.

In a system without animation, a user can easily make the mistake of using the *Move* command with a distance that takes the object off the screen. An instantaneous move effectively "teleports" the object out of sight, a mistake that is visibly indistinguishable from a delete command. By animating the move command, we give the user a chance to see the command unfold over time. In this case, the user would see the failure in progress as the object slides off the edge of the screen.

Likewise, Alice provides an animated infinite undo mechanism (figure 2, D). This mechanism always takes one second to undo an operation, regardless of the duration of the original command being undone.

Controlled Exposure of Power

Alice commands are highly overloaded, supporting several different calling patterns through a single command name. For example, *Move* can be called in all these ways:

```
obj.move(forward, 1)
obj.move(forward, 1, duration=3)
obj.move(forward, 1, speed=4)
obj.move(forward, speed=2)
```

```
# change of coordinate system
obj.move(forward, 1, AsSeenBy=camera)
# different interpolation function
obj.move(forward, 1, style=abruptly)
```

Using overloaded methods with optional keyword parameters allows us to provide a *controlled exposure of power* to the Alice user. This characteristic of the API allows novice users to become expert users by incrementally adding to what they know, rather than forcing them to learn entirely new commands or API constructs. As one Alice user said, "you can get as complicated as you want, but not more than you need to."

Scripting

Once the objects are placed and the camera is in position, this initial state is saved into a *world file*. This file also contains the name of each object so that it can be referenced in the script that will control the object's movements.

When the opening scene is ready, the user then presses the scripting tab (figure 2, F), which reveals a text editor and a *Run Script* button. The user iteratively edits the script and runs it, with the script always starting its execution from the saved opening scene.

THE IMPLEMENTATION

For several years, we followed a "time machine" approach to Alice, doing early implementation on high-end SGIs in anticipation of low cost commodity graphics. Alice now exists solely on the PC platform, running on MS Windows 95, 98, and NT with the overall structure shown in figure 4. The layers are described below.

Rendering

The rendering software is Microsoft's Direct 3D Retained Mode (D3DRM). This layer manages the 3D database of objects, their attributes, and their texture maps; illuminates the scene; and maintains the hierarchical tree of coordinate systems.

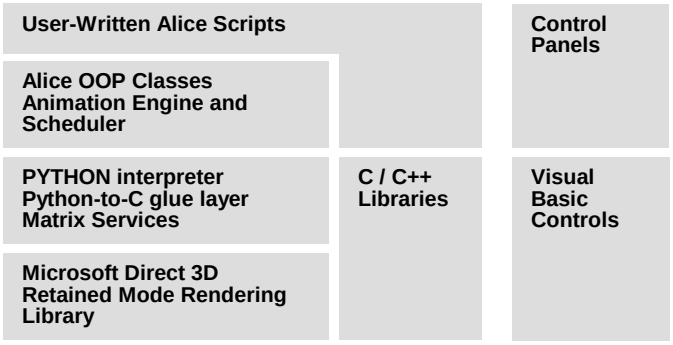


Figure 4: The Alice Software Architecture. User scripts are written in Python as is much of the Animation Engine Layer beneath it. The layers below this are written in C. Alice includes a separate Control Panels facility for creating Visual Basic GUI components directly through the Python scripting language.

Python

Alice uses a general purpose, interpreted language called *Python* [24]. We chose this language for its technical characteristics. Python is:

- a modern language with a rich set of built in data types (maps, strings, lists) and operators for those types.
- freely distributable and available without royalty.
- extensible in C/C++. For example, Alice uses the *Coriolis* collision detection library.

Although we resisted changing the Python implementation, our user testing data forced us to make two changes. First, we modified Python's integer division, so users could type 1/2 and have it evaluate to 0.5, rather than zero. Second, we made Python case insensitive. Over 85% of our users made case errors and many *continued* to do so even after learning that case was significant. Most novice-oriented systems (e.g. Hypercard, Pascal, LOGO) are designed to be case insensitive, a lesson we saw being ignored in the proposed standard for VRMLScript [10].

The Animation Engine

The Alice animation engine interpolates data values from a starting-state to a target-state over time, with a default duration of one second. When two or more animations run in parallel, the Alice scheduler interleaves the interpolations in round-robin fashion. This allows a user to evaluate a command while another command continues to animate, without any explicit thread management. The use of these *implicit threads* is a major contribution of the Alice system.

Alice itself is a single-threaded application. We built an experimental Alice prototype using native Windows 95 system threads, one per animation, but it exhibited poor load balancing between threads, giving rise to poor-quality, lurching animations.

Animated Alice commands return an animation object:

```
scoot = bob.move(forward, 1)
```

These objects respond to several methods (stop, start, loop, stoplooping) and can be composed with other animation objects:

DoInOrder(anim1, anim2,..animN), which causes the animations to run in sequence.

DoTogether(anim1, anim2,...animN), which causes them to run in parallel.

Of course, DoInOrder and DoTogether animations can also be composed, giving rise to more interesting animations. For example, given a world with an object called **Bunny**, we could make the bunny beat his drum by writing:

```
ArmsOut = DoTogether(
        Bunny.Body.LeftArm.Turn(Left, 1/8),
        Bunny.Body.RightArm.Turn(Right, 1/8) )
ArmsIn = DoTogether(
        Bunny.Body.LeftArm.Turn(Right, 1/8),
        Bunny.Body.RightArm.Turn(Left, 1/8) )
BangTheDrumSlowly = DoInOrder(
                ArmsOut,
                ArmsIn,
                Bunny.PlaySound('bang') )
BangTheDrumSlowly.Loop()
```

TESTING AND REAL USAGE

The table below summarizes the subjects in our tutorial observation sessions.

Number of Subjects	100	
Ages	High: Low: Mean: Std Deviation:	41 18 22 7.4
Sex	Female Male	58% 42%
Computer Experience	90% self-described as using email, some www, some word processor, no programming.	

Subjects were tested using a two-person talk-aloud protocol [12]. During a 30 minute session, pairs of users worked through the Alice tutorial. The tutorial walked the users through the steps necessary for creating a simple Alice world. Users would load in an object, practice moving it with the mouse and with Alice commands. They then learned about how to manipulate the parts of objects and other object attributes. Finally they learned how to name animations and build simple composite animations.

We used pairs because pairs of people naturally talk to each other about what is happening, what is confusing, and what they expect at each step. As they talked, we took notes.

Only under the most dire circumstances (e.g. system crash) did we assist users, a rule that sometimes requires a great deal of discipline, especially if the observer is also one of the system developers. While there was often strong temptation to show the subjects the "right way" out of a problem, we were strict about letting users find their own way, only interceding if the error was so great that it jeopardized the subjects' ability to finish the session.

Often these sessions were sobering. Encouraging everyone on the development staff to observe real users is an excellent way of sensitizing an entire team to the needs of one's target audience.

Other Sources of Observation Data

In addition to the formal usability sessions, we gathered observations from several other sources:

- suggestions from about 20 graduate students in a graduate-level graphics class.
- longer term observations of three in-house users using a critical-incident debriefing technique [4].
- deploying Alice at a magnet school in the Lynchburg, Virginia public school system and gathering on-site data from their students.
- exchanging detailed email with many members of our Alice user community, including some very valuable exchanges with home schooled grade-school children.

In short, these findings are neither opinions or the results of a marketing-style (*did you like it?*) questionnaire. The data comes from the observed behavior of real members of our target audience during use of the system, focusing on discovering which parts of the API were understood and which were not. We actively used this data to drive our design: we would build part of the system, and then user test to see where users consistently had difficulty. We would then redesign those parts, build new ones, and user test again.

FINDINGS

The Death of XYZ

Perhaps Alice's most distinguishing API feature is that it allows people to create behavior for three-dimensional objects without using the traditional mathematical names for the coordinate axes: X, Y and Z. Instead, Alice uses LOGO-style [16], object-centric direction names: Forward/Back, Left/Right, and Up/Down. We made this design decision after using XYZ for two years where we routinely observed users, even expert ones, saying things like:

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

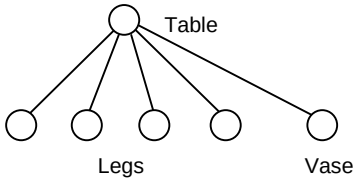
Our users already had a vocabulary for moving objects in space, but our early system was not using it. While it is true that *some* objects do not have an intrinsic forward direction, this is at least an improvement, because there are *no objects that have an intrinsic X direction*.

This seemingly tiny, cosmetic change is probably Alice's biggest contribution to making a usable API for 3D graphics. By using direction names in lieu of XYZ, we relieved the user of a cognitive mapping step that may occur thousands of times while developing a 3D program. Removing XYZ also reduced the need to talk about negative numbers to an audience that naturally shies away from mathematics.

First Class Objects and Parts

Like many 3D graphics systems, Alice uses a tree-like structure of nodes and children to organize the objects in a 3D scene.

Take, for example, the case of a vase sitting on a table. If we wanted the vase to move when the table moved, we might reasonably model the vase as a child of the table.



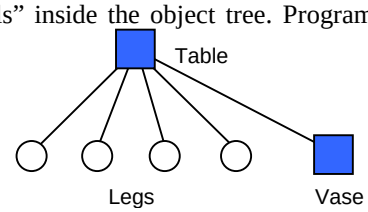
But which are parts of the table, and which are independent objects merely resting on the table? Said another way: how would we ensure that

Table.setColor(Red)

turns the table red without also turning the vase red? Intuitively, the vase is different than the table legs, but in an undifferentiated PHIGS tree, all objects look alike.

Alice marks objects loaded from disk as "first class objects" (the table and the vase) and marks the objects *defined inside those files* as parts (the legs of the table).

This allows some operations (e.g. SetTexture, SetColor, Destroy) to tell the difference between an object's parts and nodes that are attached for some other reason; the operation stops at other first class objects. First-class objects thus act like "firewalls" inside the object tree. Programmers can use



the optional parameters ObjectOnly, ObjectAndParts, ObjectAndChildren to override this behavior when they need more control.

The first-class attribute of an object is also used to control picking into the scene and other event-dispatching within Alice. This is very much like the pick-bit that some PHIGS systems have used in the past.

Objects Are Coordinate Systems (AsSeenBy)

Although 3D applications can perform all geometric operations in a global coordinate system, most 3D applications perform coordinate transformations to make geometric operations easier to compute or reason [5]. The Alice API already eases the burden of coordinate transformation to some extent by making the move and turn commands operate from the object's local coordinate system, rather than from a global frame of reference. For more general coordinate system transforms, we designed a system [7] that allowed programmers to perform any geometric operation within any other object's coordinate system. This capability is invoked by adding the coordinate system object with the optional AsSeenBy keyword parameter, as in:

bunny.move(forward, 1, AsSeenBy=chair)

Users can also use this mechanism to work in a global coordinate system by using the Scene as the reference object. Alice provides a default Scene object for every world.

Inspired by Alice, Disney Imagineering's *Player* system [15] and Microsoft's Direct 3D graphics libraries have both added this mechanism to their systems. Both are excellent examples of improving a system for experts based on lessons learned from a system for novices.

Beyond Translate, Rotate, and Scale

The Alice system provides commands that go beyond the classic translation, rotation, and scale operations:

Place – this command allows one object to be placed OnTopOf, InBackOf, ToRightOf etc, some other object: cup.place(OnTopOf, Table). This command was developed independently of and simultaneously with the similar but somewhat more expressive *Put* system, developed at SGI [3].

PointAt and **StandUp** are underconstrained rotation operations that use a global "up" direction to resolve the ambiguous nature of the command. A.PointAt(B) will make A's up vector parallel to the global up direction, while it rotates A around its other two axes so that A's forward direction passes through B. Fred.StandUp() will rotate Fred so that Fred's up direction runs parallel to the global Up direction, minimizing the rotation around the other two axes.

Nudge is similar to the *Move* command, but translates objects by a percentage of their size, not by an absolute distance. obj.nudge(forward, 0.5) will move an object forward a distance equal to half its front-to-back length. This allows people to create animations that are more portable across a wider range of object sizes.

AlignWith makes one object point in the same direction as another. This simple command is very handy and would be fairly difficult for novice users to implement given just a turn (rotate) command.

Pan rotates objects left or right around an axis that is parallel to the global up vector. This allows a camera that is tilted down toward the ground to remain pointed at the ground as it turns in place.

Resize

One of our first user observation sessions gave rise to vocal complaints about the strange side-effect that scale had on distances. Calling

bunny.scale(3)

resulted in a bunny that was 3 times as large, but it also had the effect that a subsequent call to

bunny.move(forward, 1)

Moved the bunny forward 3 meters, not 1. Our users correctly surmised that making the bunny larger also scaled its local space.

This side effect comes from our implementation: Alice uses a four dimensional homogeneous coordinate transformation matrix to represent object position, rotation and scale, an implementation that by design scales the space of objects [5].

To provide more useful semantics to this command, we use a second 4x4 matrix to keep track of scale. Alice's *Resize* command now changes the scale of an object via this second scale matrix without changing the object's position and rotation matrix; this allows us to resize the object's geometry without scaling its space. By propogating the effects of this scale matrix to the object's children we can similarly resize the geometry and offset relative to their parent of the object's children without scaling their space. As a result, a meter in Alice is always a meter regardless of whether or not an object has been resized.

Resize also takes an optional parameter, LikeRubber

```
bob.resize(FrontToBack, 1/3, LikeRubber)
```

which scales an object in a way that preserves volume, an important technique for many animation effects [8].

Vocabulary Issues

Novice users are strongly influenced by surface issues, and seemingly inconsequential name choices can often make the difference between a clear API and a confusing one. Some notable examples:

Resize, not Scale: Scale is usually regarded as a noun, not a verb, and has strong connotations of weight, not size.

Move, not Translate: Translation is understood by our target audience to be the process by which French is converted into German and has little to do with movement.

Speed, not Rate: Alice commands can specify how fast something happens, as in bob.move(left, speed=1). Users were observed to have a few problems with *Rate* in that it seemed to have percentage or interest rate connotations, while *Speed* never caused confusion.

FrontToBack, not Depth: Previous versions of Alice used the words Depth, Width and Height to denote the dimensions of an object. We found that these terms were sufficiently ambiguous to users that we resorted to the clearer, but somewhat more cumbersome FrontToBack, LeftToRight, TopToBottom. While these terms are somewhat contrived, they at least have the advantage of clarity and are formed out of terms that a novice Alice user already knows.

AsSeenBy, not CoordSys: this name change was almost aesthetic in nature, and allowed script-writers to read scripts more naturally.

Color Names, not RGB Triples: Alice uses popular crayon color names like Red, Green, Peach, and Periwinkle to specify colors, not a numeric color model like RGB or HSV.

No individual name choice is pivotal to Alice's success, but the aggregate effect of getting these names right is quite powerful. Almost all Alice scripts use only the following commands:

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, ShowFrustum

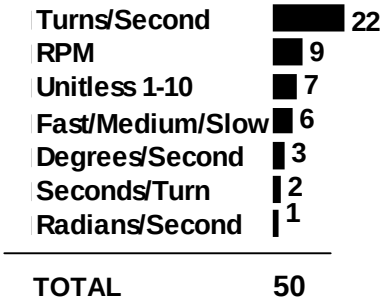
ROTATION RATE: ROTATIONS, NOT DEGREES

Alice's *Turn* command originally allowed programmers to specify angular amounts in degrees and the animation time in seconds, so it seemed natural that rotational *speed* be specified in degrees-per-second. Informal observation suggested that this unit was confusing.

After our test subjects had seen the first Alice tutorial and were familiar with the Alice *Turn* command, we posed the following question:

To turn objects in Alice, you specify a direction to turn (left, for example) and an amount (90 degrees, for example). Suppose you did not know an exact amount, but you wanted to make the bunny turn around and around without stopping? How would you want to describe the speed that the bunny turns?

A breakdown of the answers appears below:



Notice that turns-per-second is a clear favorite and that degrees per second, *the units we, the engineers had chosen, came in fifth*. In reaction to this, we now specify rotational

speed in turns-per-second, and angular amounts in turns. In retrospect, it seems very natural to express a "quarter turn" by typing bunny.turn(left, 1/4).

Other Observations About Novices

Typing is Hard – Most of our users were non-typists and appreciated any help we could give them (mouse control, dialogs, etc.) that would keep them from having to use the keyboard. We are currently working on addressing this issue in Alice.

Problems in 3D Perception – A small percentage (~5%) of our subjects were confused about the depth of objects on screen, sometimes mistakenly seeing objects as approaching or receding when in fact they were being resized. Shadows or other depth cues might help reduce these problems.

High Expectations – Our subjects often expected collision detection and gravity and were surprised when objects passed through each other or hovered in mid-air.

The Importance of 0 and 1 – When faced with a new Alice command that required a numeric parameter, we saw many users try using a "1" to see what would happen for a wide variety of data types (distance, color, time). Partly due to this, we adopted a convention that all bounded scalar parameters to Alice calls would range between 0.0 and 1.0. "Magic ranges" like 0..255 and 0..32767 do not hold much appeal for novices.

RELATED WORK

LOGO [16], Bolio [25], and the Alternate Reality Kit (ARK) [19] and the animated Self programming environment [22] were all strong influences in the Alice project.

Smalltalk [6] and HyperCard [13] both demonstrated that programming-in-the-small was feasible by nonprogrammers.

The Simple User Interface Toolkit (SUIT) [14] used a two-user protocol to test an API for novice GUI programmers.

BAGS (Brown Animation Generation System) [20], 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.

Like Alice, Obliq 3D [11] uses an interpreted scripting language for 3D graphics, but unlike Alice, is designed for experts.

Superscape [21] and WorldUp [18] include advanced geometric modeling capabilities and scripting languages. WorldUp shares some ease-of-use goals with Alice, but has a very different model for the distribution of scripts and the timing of animations.

FUTURE WORK

Decomposing complex animations (e.g. walking) into Alice animation primitives is still too hard. A richer set of animation primitives might help.

Writing a serial sequence of code (A then B) requires too much syntax , due to Alice's implicit threads.

We need to find ways of exposing the number and order of parameters to a function to ease the burden of typing.

There are times when programming declaratively (e.g. constraints) is more natural than expressing solutions procedurally, and vice versa. Finding the correct mix of programming styles, while folding in the advantages of some of the other animation paradigms (e.g. keyframing) remains an open problem.

SUMMARY OF LESSONS LEARNED

- Forward/Left/Up is an improvement over XYZ.
- Coordinate transformations can be made easier by allowing other objects to act as the frame of reference in which other operations happen.
- Function overloading and optional keyword parameters in a programming language can be used to support the *controlled exposure of powe*r, masking API complexity until the user is motivated to use it.
- Matrices appear nowhere in the Alice API.
- APIs can and should be tested against real users from one's target audience.
- Marking some objects as *first class objects* is a powerful technique for segmenting one object from another in the object tree.
- All commands should animate by default, including Undo.
- Implicit threads make it possible for novices to control surprisingly complex animations.
- Object resize and the scaling of space are both useful, but should be presented to the user as two distinct operations.
- Surface characteristics of programming languages matter to novices, especially case sensitivity and careful name choices.
- All bounded, scalar parameters should have a valid range of 0.0 to 1.0.

CONCLUSION

Alice represents the culmination of many independent design decisions based on hundreds of observations of novices. These decisions combine to form a 3D graphics API that allows 3D script writing with minimal distraction by "unrelated" issues. As one researcher in the field kindly noted, current tools inflict the "death of a thousand cuts" compared Alice's "joy of a thousand tickles." Although originally designed for undergraduates, we have observed that many middle and high school students are capable of using Alice to build interactive 3D graphics programs. Alice is available for free from http://www.alice.org. We have currently distributed over 50,000 copies of Alice.

REFERENCES

1. Card, S. K. Robertson, G., and Mackinlay, J. The Information Visualizer, an Information Workspace. *ACM SIGCHI 91 Conference Proceedings*, 1991, pp. 181-188.

2. Clarke, J. H. Hierarchical Geometric Models for Visible Surface Algorithms. *Communications of the ACM*, 19(10), October 1976, pp. 547-554.

3. Clay, S. R., and Wilhelms, J. Put: Language-Based Interactive Manipulation of Objects. *IEEE Computer Graphics and Applications*, March 1996. Vol 16, Number 2, pp. 31-39.

4. Fitts, P. M., and Jones, R. E. Pychological Aspects of Instrument Display: Analysis of 270 "Pilot Error" Experiences in Reading and Interpreting Aircraft Instrument. *Memorandum Report TSEAA-694-12A*, Aero Medical Labaroatory, Air Materiel Command, Wright Patterson Air Force Base, Dayton, Ohio, October 1, 1947, pp. 47.

5. Foley, J. D., van Dam, A., Feiner, S. K., and Hughes, J. F. *Fundamentals of Interactive Computer Graphic*, Addison-Wesley Reading, MA 1990.

6. Goldberg, A., and Robson, D. *Smalltalk80: The Language*, Addison-Wesley, Reading, MA, 1989.

7. Gossweiler, R., Long, C., Koga, S., and Pausch, R. 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.

8. Lasseter, J. Principles of Traditional Animation Applied to 3D Computer Animation. *SIGGRAPH 87 Conference Proceedings*, pp. 35-44.

9. Mackinlay, J. D., Card, S. K., and Robertson, G. G. Rapid Controlled Movement Through a 3D Virtual Workspace. *ACM SIGGRAPH 1990, Conference Proceedings*, pp 171-179.

10. Marrin, C., and Kent, J. Proposal for a VRML Script Node Authoring Interface, *VRMLScript Reference*, Silicon Graphics, Inc.October 6, 1996.

11. Najork, M. Obiq-3D Tutorial and Reference Manual. *DEC SRC Research Report #129*, December 1, 1994.

12. Nielsen, J. *Usability Engineering*, Academic Press, Boston, 1993.

13. Nielsen, J., Frehr, I., and Nymand, H. O. The learnability of HyperCard as an object-oriented programming system. *Behaviour & Information Technology* 10, 2 (March-April), 111-120.

14. Pausch, R., Conway, M., and DeLine, R. Lessons Learned from SUIT, the Simple User Interface Toolkit. *ACM Transactions on Office Information Systems* October 1992, 10:4, pp. 320-344.

15. Pausch, R., Snoddy, J., Taylor, R., Watson, S., and Haseltine, E. Disney's Aladdin: First Steps Toward Storytelling in Virtual Reality. *ACM SIGGRAPH 96 Conference Proceedings*, August 1996.

16. Papert, S. *MindStorms: Children, Computers, and Powerful Ideas*, Basic Books, New York, 1980.

17. Robertson, G. G., Card, S. K., and Mackinlay, J. D. The Cognitive Coprocessor Architecture For Interactive User Interfaces. *ACM Symposium on User Interface Software and Technology*, 1989, pp. 10-18.

18. Sense8 Corporation: http://www.sense8.com.

19. Smith, R. B. 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.

20. Strauss, P. BAGS: The Brown Animation Generation System. *Technical Report No. CS-88-22*, Brown University, May 1988.

21. Superscape: http://www.superscape.com.

22. Ungar, D., and Smith, R. SELF: The Power of Simplicity. *OOPSLA 87, Conference Proceedings*, published as SIGPLAN Notices, Volume 22, Number 12, 1987, pp. 227-241.

23. van Dam, A., et. al. PHIGS+ Functional Description Revision 3.0, *Computer Graphics* 22, 3, (July 1988), 124-218.

24. van Rossum, G., and de Boer, J. 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.

25. Zeltzer, D., Pieper, S., and Sturman, D. J. An Integrated Graphical Simulation Platform, *Graphics Interface 89 Conference Proceedings*, pp. 266-274.