Introduction to Alice Programming

Baldwin reviews a major 3D interactive graphics Java program named Alice that is used to teach beginners how to program.

Published: April 24, 2007
By Richard G. Baldwin

Java Programming Notes # 1516

- Preface
  - General
  - Viewing tip
    - Figures
    - Listings
    - Tables
  - Supplementary material
- Preview
- Discussion
  - Getting started
  - Creating and animating an Alice world
  - Objects in 3D space
  - A simple Alice program
- Resources
- Complete program listings
- Copyright
- About the author

Preface

Target audience

The target audience for this article includes experienced programmers who would like an interesting diversion from work while eating lunch at their desk, and experienced programmers who have children, grandchildren, siblings or friends who need to learn how to program.

An interesting diversion

If you spend your lunch hour at your desk eating a sandwich, playing Cub Rummy, watching outrageous videos at YouTube, or watching bad-news videos on your favorite news channel, all you have to show for it at the end of the lunch period is indigestion. However, if you spend that time using Alice to create an animation video of the ice skater shown in Figure 11 jumping over

three or four of the characters in Figure 1, you could show it off to your peers and brag a little at the end of the lunch period.

Teaching teenagers to write computer programs

On a slightly more serious note, have you ever had the thought that you would like to teach your teenager to program? Your reason may simply be to try to make them aware that there is more to a computer than a tool for hanging out at myspace.com, visiting YouTube, or playing computer games written by others.

No way!

When you mentioned that possibility to your teenager as casually as you could, did she roll his eyes and give you a look that means "No way!"

Whatever your reason for thinking that it might be beneficial for your teenager to learn a little about computer programming, there may actually be a way to interest that teenager in learning to program. That way is a very impressive Java program named Alice, which can make learning to write computer programs fun even for teenagers. *(It will give them the appearance of being gamers, which is probably more socially acceptable than the appearance of being nerds.)*

Disclaimer: Professor Baldwin has no vested interest in Carnegie Mellon University or the Stage3 Research Group. He is an independent author whose only interest in Alice is that of professional development.

A Java program from Carnegie Mellon University

Alice is a computer program written in Java by the Stage3 Research Group at Carnegie Mellon University that makes it easy to write object-based, event driven, 3D animation and game programs with a cast that can be selected from *hundreds* of available objects. A small sampling of the available objects from the Alice class gallery is shown in Figure 1.

Figure 1. A sampling of the objects available in Alice.

In this article, I will give you some background on the Alice program, and describe some of its features.

General

Alice is not a toy

Alice is not a toy designed solely to create pretty pictures. Rather, Alice is a full-featured programming language designed for use in teaching programming to beginners on the basis of programming principles that are well recognized within the computer science community. For example, Alice supports almost all of the fundamental programming concepts that we have taught for many years in the programming fundamentals course at the community college where I teach.

A serious 3D graphics programming environment

Here is a quotation from the projects page of the Stage3 Research Group:

Alice is a 3d graphics programming environment intended to be a gentle first introduction to students ranging from 6th grade to college, typically students who would not take (or pass!) a programming course.

Elimination of frustration

The Alice project was motivated by the fact that for most first-time students, the experience of learning to program has been filled with frustration. Hours of trying to understand syntax errors in pursuit of a working Fibonacci sequence generation program have lead many students to conclude that Computer Science is uninteresting before they have completed a single course.

The goal of Alice

The goal of the Alice project is to change the first experience students have with computer programming. We believe that Alice will change the experience of learning to program in two main ways: removing unnecessary frustration and providing an environment in which beginning students, of both genders, can create programs they find compelling.

Drag-and-drop instead of type

Continuing with quotations from the Stage3 Research Group:

When students create programs in Alice, they do not type. Instead, they drag and drop words representing commands that objects in the 3D scene understand.

For example, a student may instruct a bunny in the 3D scene to "move forward" or "look at the camera."

Alice is full featured

In addition to straight-forward commands, students can also drag traditional programming constructs, such as "if," "loop N times," "do while something is true," etc. Students can construct "If" statements by dragging questions like "is the carrot near the rabbit" or "how tall is the tree" into them.

Although the terminology is intentionally simplistic, Alice is actually a complete programming environment, supporting arrays, lists, functions with parameters, recursion, and an object-based data model. In addition, methods can be stored as part of an object and then loaded into different 3d "worlds" created with Alice.

And might I add that Alice provides a rudimentary system for writing event-driven programs, making it suitable for writing games and instructional programs for younger children.

Why does Alice succeed?

Alice succeeds for several fundamental reasons

1. *By removing typing and the ability to make a syntax error, Alice removes much of the initial frustration for new programmers,*
2. *The ideas of data and objects are very concrete when students can *see* what they are, and*
3. *Almost all changes to the program state are visible and animated, so debugging is a much less obscure task it is much easier to realize that "the rabbit moved backwards when I meant to for it to move forward" than to realize that "I subtracted one from the integer 'x' when I intended to add one" (particularly when 'x' isn't directly visible on the screen).*

Alice 2.0 is free and practical

The Alice 2.0 programming environment can be downloaded free of charge from Carnegie Mellon University. Furthermore, it doesn't require a Windows installation. All that is required to run Alice is to:

1. Download the zip file containing the Alice environment.
2. Extract the files and directories from the zip file into a local directory.
3. Double-click on one of the exe files that are extracted from the zip file.

Once the zip file is downloaded, further access to the Internet is not required. Many classes for creating 3D objects are stored locally in an area that is called the gallery. However, in order to conserve local disk space, the classes for many other objects are not routinely included in the local class library. Rather, they can be accessed from a web version of the class gallery.

Fits on a 256 mbyte USB flash drive

The entire local version of the Alice development environment will fit on a 256 mbyte USB flash drive, and can be executed directly from the flash drive. This makes it possible for students to carry the development kit with them from one computer to another.

Improved accessibility

The use of the drag-and-drop programming paradigm causes Alice to be much more accessible to beginning programming students than languages such as Java, C++, and C# that require extensive keyboard activity for use. A student who can type a few strings with one finger and operate a finger-driven mouse pad can write Alice programs just as rapidly and effectively as a student who can type 60 wpm.

What about Alice documentation?

Having been teaching and programming in Java for the past ten years, I have become spoiled by the availability of extensive detailed Java documentation. In my opinion, the main thing that is lacking from Alice 2.0 is good documentation.

Because I am planning to teach a programming fundamentals course using Alice in the near future, I consider a good set of documentation to be vital to the success of that course. Apparently however, most of the efforts of the Stage3 Research Group at Carnegie Mellon are now being dedicated to the development of Alice 3.0. Therefore, I don't expect much in the way of additional documentation on Alice 2.0 to be forthcoming from that group.

Alice documentation by Dick Baldwin

Therefore, I am developing a form of documentation on my own. I am making it freely available online at http://www.dickbaldwin.com/tocalice.htm.

Sending email to Dick Baldwin

This documentation will include a combination of tutorial lessons explaining how to program using Alice and appendices containing slightly more formal descriptions of various features of the language such as methods, functions, and events. Please feel free to use the documentation, and also feel free to contact me via email to make suggestions regarding possible improvement in the documentation.

Send your email message to baldwin@dickbaldwin.com and include the word Alice surrounded by spaces in the subject line to cause the message to bypass my spam blocker. *(Note that if the spammers catch onto this, I will modify these instructions later.)*

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures, listings, and tables while you are reading about them.

Figures

- Figure 1. A sampling of the objects available in Alice.
- Figure 2. The Alice splash screen.
- Figure 3. The scene edit mode.
- Figure 4. An airplane with its 3D axes exposed.
- Figure 5. A Coach object.
- Figure 6. Left arm of the Coach object.
- Figure 7. Coach's arm *turned* 90 degrees to the right.
- Figure 8. Coach's arm *turned* backwards by 180 degrees.
- Figure 9. Coach's arm *rolled* right by 45 degrees.
- Figure 10. Coach's arm pointing to the left and up with palm up.
- Figure 11. Skater's starting and ending pose in program Alice0125f.
- Figure 12. Animation shot from program Alice0125f.
- Figure 13. A reduced screen shot of the Alice program edit mode.
- Figure 14. Object tree with iceSkater object expanded.
- Figure 15. Reduced screen shot of Events area in program edit mode.
- Figure 16. Reduced screen shot of edit pane in program edit mode.
- Figure 17. Screen shot of methods tab in details area for iceSkater object.
- Figure 18. Screen shot of properties tab in details area for iceSkater object.
- Figure 19. Screen shot of functions tab in details area for iceSkater object.

Listings

- Listing 1. Events for the program named Alice0125f.
- Listing 2. The main method for the program named Alice0125f.
- Listing 3. Source code for the method named setTheStage.
- Listing 4. Source code for method named playTheShow.
- Listing 5. Source code for the program named Alice0125f.

Tables

- Table 1. Standard primitive methods in Alice.
- Table 2. Standard functions belonging to objects.
- Table 3. Categories of functions in the world.
- Table 4. Custom methods for a penguin object.
- Table 5. Event types in Alice.
- Table 6. Allowable types in Alice.
- Table 7. Control structures in Alice.

Supplementary material

I recommend that you also study the other lessons in my extensive collection of online tutorials. You will find a consolidated index at www.DickBaldwin.com.

Preview

The remaining sections of this article will contain brief discussions on the following topics as they pertain to Alice:

- Getting started
- Setting the stage
- Objects in 3D space
- A simple Alice program

Discussion

Getting started

Alice is not an object-oriented programming *(OOP)* language

While Alice is an outstanding product for teaching *object-based* programming, in my opinion Alice is not an object-oriented programming language.

A true OOP language must make the following features available to the programmer:

- Encapsulation
- Inheritance
- Polymorphism

Alice supports encapsulation quite well. In addition, it makes a reasonable gesture towards making inheritance available to the Alice programmer. However, there is no semblance of support for polymorphism.

In addition, there are numerous detailed features *(such at type casting)* that are expected from a true OOP language which are not available in Alice.

Makes writing 3D graphics a breeze

However, the Alice program makes it possible for a student to learn in a few months how to write 3D animation programs that would probably require the student to study for years if he were programming in hard-core Java 3D instead of using a Java program to write his programs in Alice.

While Alice is not a fully object-oriented programming environment, it is object-based and therefore is very suitable for getting a student started down the road towards object-oriented programming.

Downloading, installing, and running Alice

The installation of Alice on a Windows XP computer is very straightforward. No formal *"Windows Installation"* is required. All the student needs to do is to download a zip file from the Alice website *(see Resources)* and to extract its contents into a directory of his choice on his hard disk. *(Alice is also available for the Macintosh, but I'm not qualified to provide any information in that area.)*

Once the student has extracted the files from the zip file, he double-clicks on one of two executable files to start Alice running. The two files are:

- Alice.exe
- SlowAndSteadyAlice.exe

Why two files?

As I understand it, the file named **Alice.exe** is for use with computers having high-quality hardware graphics capability, whereas the file named **SlowAndSteadyAlice.exe** is for use with computers that don't have that capability.

The student should first try double-clicking on the file named **Alice.exe**. If that works OK, use it. If not, double-click on the file named **SlowAndSteadyAlice.exe**.

When things go right

If things go right when the student double-clicks on the exe file, the flash screen shown in Figure 2 should appear on the screen.

Figure 2. The Alice splash screen.

For more information on the Alice development screen and the testing of an installation, follow the link to the tutorial titled "Getting Started" in Resources.

Creating and animating an Alice world

Creating and animating an Alice *world* consists of two very distinct steps. The first step, which I refer to as *setting the stage*, is similar to setting the stage for a stage play or a movie production. It involves the process of painting scenery, selecting costume colors, putting the players, the scenery, and other objects in their correct positions on the stage, and getting ready for the curtain to rise, or the cameras to roll.

A small portion of the effort required to set the stage must be accomplished manually outside of the program code. The remainder of the effort to set the stage can be accomplished either outside of the program code or using program code, depending of the preferences of the programmer. As I will illustrate in the simple Alice program that I will present and explain later, my preference is to accomplish that effort using program code.

The second step in creating and animating an Alice world is to write the program code to animate the world causing the players to behave according to the script.

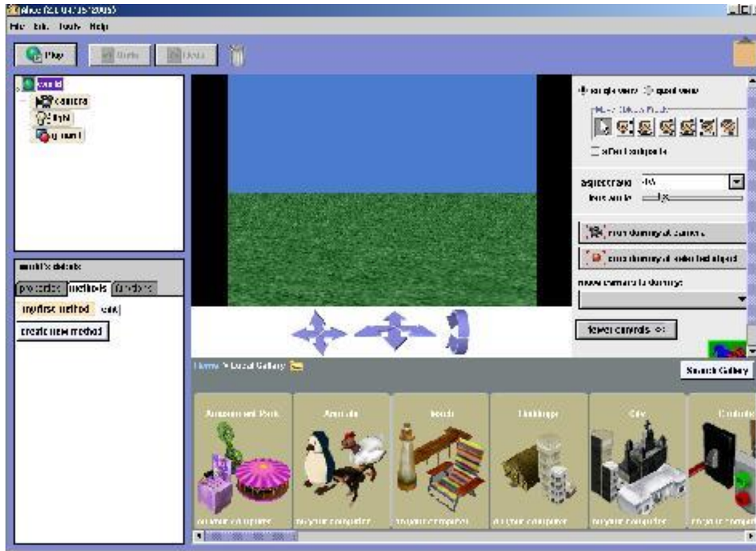
Two major edit modes

The Alice development screen can be switched between two major edit modes:

- Scene edit mode
- Program edit mode

Figure 3 shows a greatly reduced screen shot of the scene edit mode.

Figure 3. The scene edit mode.



A picture of the world

The picture of the green grassy area at the top center shows an image of the current state of the world including the objects that have been added to the world. *(The only object that has been added to the world in Figure 3 is the ground.)*

The gallery

The bottom of the screen in Figure 3 shows the graphical user interface into the class library called the gallery. This is where you go to find classes from which to instantiate objects and add them to the world. This area also contains a link to a web-based gallery where you will find even more classes from which to instantiate objects.

The remainder of the scene edit screen

You can learn about all of the other features of the development screen in scene edit mode by following the links to the various tutorials listed in Resources.

Most of the time, you work in the scene edit mode while setting the stage.

Instantiating new objects

One of the limitations of Alice as compared to other programming environments such as Java, C++, and C# is that objects cannot be created and destroyed dynamically at runtime. Rather, all objects must be manually created and added to the world in scene edit mode outside of program code. Those objects that are not needed in a particular scene can either be moved off camera or made invisible or possibly both.

Methods, functions, and

The members of an Alice object

Everything in an Alice world, except possibly the world itself is an object. All objects in Alice *(except for the world itself)* encapsulate twenty standard primitive methods, about twenty-nine standard functions, and a fair number of properties. This includes the objects that represent the camera, the light, the ground, and objects instantiated from classes in the gallery.

properties
In Alice, functions always return a value and methods never return a value. A property is a variable belonging to the object, which would probably be called an *instance variable* in Java.

A list of the twenty standard primitive methods is presented in Table 1.

Table 1. Standard primitive methods in Alice.

Primitive methods in Alice 2.0

1. move(direction,amount)
2. turn(direction,amount)
3. roll(direction,amount)
4. resize(amount)
5. say(what)
6. think(what)
7. playSound(sound)
8. moveTo(asSeenBy)
9. moveToward(target,amount)
10. moveAwayFrom(target,amount)
11. orientTo(asSeenBy)
12. turnToFace(target)
13. pointAt(target)
14. setPointOfView(asSeenBy)
15. setPose(pose)
16. standUp()
17. moveAtSpeed(direction,speed)
18. turnAtSpeed(direction,speed)
19. rollAtSpeed(direction,speed)
20. constrainToPointAt(target)

To learn more about the primitive methods in Alice, follow the link to the tutorial titled "Appendix A, Behavior of Primitive Methods" in Resources.

Controlling motion and viewpoint

As you can see, most of the primitive methods shown in Table 1 have to do with motion, position, and orientation *(viewpoint)*. The Alice programmer has full motion control over all of the objects in the world including the camera, the light, and the ground. *(The world has no primitive methods and cannot be moved or re-oriented.)* Furthermore, the programmer can cause

the motions of different objects to occur either concurrently, in order, or a combination of the two.

Method parameters

With one exception, all of the primitive methods have one or two required parameters, which are fairly intuitive in nature. *(A few are not so intuitive, which is one of the reasons that I wrote and published Appendix A, which is referenced in Resources.)*

In addition, most and probably all of the primitive methods have optional parameters with default values. *(In my writings, I refer to these parameters as default parameters.)* When writing an Alice program and calling one of these methods, you can either accept the default values, or provide different values.

Three very interesting default parameters

Three of the default parameters that most of the primitive methods have are:

- duration
- style
- asSeenBy

These are not the only default parameters. They are simply the ones that I elected to highlight in this article.

The duration parameter

The **duration** parameter specifies the amount of time that will be required to accomplish the action dictated by the method. The default value is one second, but you can set it to any value you choose, including zero if you need the action to be completed very quickly. This is a parameter whose value you will frequently change.

The style parameter

The **style** parameter specifies the behavior of the action at the beginning and the end. The options are:

- BEGIN_AND_END_GENTLY
- BEGIN_GENTLY_AND_END_ABRUPTLY
- BEGIN_ABRUPTLY_AND_END_GENTLY
- BEGIN_ABRUPTLY_AND_END_ABRUPTLY

The default is the first item in the list. That choice would be appropriate, for example for the motion of an object in space under the influence of mass and inertia. I find a need to change the value of this parameter much less frequently than the value of the **duration** parameter for example.

The asSeenBy parameter

The **asSeenBy** parameter is particularly interesting. The default value is the object on which the method is called. However, some very interesting effects can be achieved by setting the value to a different object. In that case, the effect will depend on the fundamental behavior of the method.

For example, one of the sample programs that I explain in the lessons referenced in Resources uses this parameter to cause a hungry shark to circle a hapless bunny in the water. A second use of this parameter causes the camera to circle the unfortunate scene below as though the camera is in a news helicopter witnessing the scene.

Another example that I present in those tutorial lessons uses this parameter to cause an airplane to fly loops.

Standard functions

Table 2 shows the standard functions that belong to most, if not all objects other than the world, including the camera, the light, and the ground. These functions are called for a variety of purposes in an Alice animation program.

Table 2. Standard functions belonging to objects.

Standard functions
1. isCloseTo(threshold,object) 2. isFarFrom(threshold,object) 3. distanceTo(object) 4. distanceToTheLeftOf(object) 5. distanceToTheRightOf(object) 6. distanceAbove(object) 7. distanceBelow(object) 8. distanceInFrontOf(object) 9. distanceBehind(object) 10. getWidth 11. getHeight 12. getDepth 13. isSmallerThan(object) 14. isLargerThan(object) 15. isNarrowerThan(object) 16. isWiderThan(object) 17. isShorterThan(object) 18. isTallerThan(object) 19. isToTheLeftOf(object) 20. isToTheRightOf(object) 21. isAbove(object)

22. isBelow(object)
23. isInFrontOf(object)
24. isBehind(object)
25. getPointOfView
26. getPosition
27. getQuaternion
28. getCurrentPose
29. partNamed(key)

Functions in the world

The world provides more than fifty other functions, most of a very utilitarian nature. They are organized into categories. The categories along with some examples are shown in Table 3. You will note that many of the functions in both Table 2 and Table 3 have a distinct Java-like appearance.

Table 3. Categories of functions in the world.

Categories of functions in the world

- boolean logic example: (a && b)
- math example: a != b
- random example: Random.nextDouble()
- string example: what.toString()
- ask user example: NumberDialog(question)
- mouse example: Mouse.getDistanceFromLeftEdge()
- time example: getTimeElapsedSinceWorldStart()
- advanced math example: Math.IEEERemainder(a,b)
- other example: getVector(right,up,forward)

You will frequently need to drag these functions and drop them onto placeholders in expressions in the edit pane to construct the desired expressions in your statements.

Custom methods

Some objects instantiated from classes in the gallery contain custom methods in addition to the primitive methods. For example, Table 4 shows the custom methods that are provided by Alice for a penguin object.

Table 4. Custom methods for a penguin object.

Custom methods for penguin object
- wing_flap(times) - jumping(height) - turn_head_right() - turn_head_left() - glide() - jump(times) - walk(move_times) - walking(x)

Custom methods may or may not have default parameters, depending on whether or not the author of the method created them.

Defining and saving a class

Once you create an object and add it to a world, you can define and add new methods, functions, and properties to the object.

After you add methods, functions, and/or properties to an object, you can export and save the class that is represented by the current configuration of the object. You can use that new class to create additional objects in the same world, or you can use it later to create objects for a different world. If you wish to do so, you can add it to the gallery. This represents a rudimentary form of inheritance, and is the mechanism by which you can create a library of custom classes.

Various choices for setup

Except for the manual creation of the objects, which is required, the process of setting the stage can be accomplished either manually or using setup code at the beginning of the program *(as I will do in a sample program later in this article)*, or using a combination of the two.

Objects contain component parts

Many of the Alice objects contain various component parts *(see Figure 14)*, which are also objects. Examples of component objects are the legs, arms, and hands of an object that represents a person, the propeller, wheels, and fuselage of an airplane, the head, wings, and feet of a penguin, etc.

Creating and saving a pose

Often the process of setting the stage involves manipulating the various parts of an object to create a pose. If you are doing a manual setup, you have a choice of doing this by:

- Dragging components with the mouse.
- Interactively calling methods on the objects that constitute the parts.

- A combination of the two.

Once you have created a pose, you can save it as a property of the object and use it in program code at runtime. To learn more about operations such as this, follow the links to the lessons in Resources.

Objects in 3D space

An object's viewpoint

Every Alice object has a *viewpoint*. The viewpoint of an object is determined by:

- The position of the object in 3D space.
- The orientation of the object as determined by a 3D coordinate system that belongs to the object.

For example, causing an object to move to a different position or to face in a different direction, or both would change the viewpoint of that object. Generally speaking, when viewpoint is important, I usually normalize the viewpoint of each object relative to the viewpoint of the world as the baseline standard.

An object's center point

Every Alice object has a *center point* and three axes. The center point is the position in space *(relative to the object)* at which that object's three orthogonal coordinate axes cross. *(This is often called the origin in other programming environments.)*

Sometimes the center point is inside the object and sometimes it is outside the object. For example, the center point of the penguin shown in Figure 1 is a point on the ground midway between the penguin's feet, while the center point of the airplane shown in Figure 4 is inside the fuselage of the airplane.

An object's motion

If you ***move*** an Alice object, you are actually moving its center point. The rest of the object comes along for the ride. *(See sidebar.)*

A disconnected object
Note, however, that it is possible to break the connection between an object and its component parts. For example, you could cause an object that represents a man to move away and leave his arms behind.

If you ***turn*** an Alice object to the *right* or to the *left*, you are rotating the object around one of its three axes. This is sometimes called *yaw*. If you ***turn*** an Alice object *forward* or *backward*, you are rotating the object around a different axis. This is sometimes called *pitch*. If you ***roll*** an Alice object to the *right* or the *left*, you are rotating the object around a third axis. This is sometimes called *roll*.

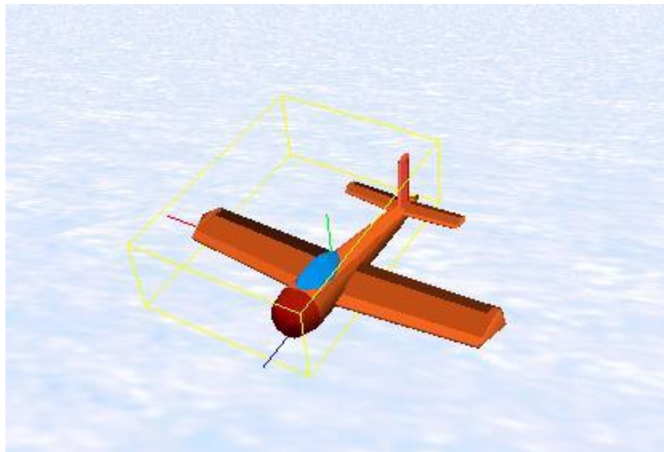
Italicized method names

Why did I represent the words move, turn, and roll in boldface Italics in the above text? I did that to emphasize that I wasn't simply using a generic term for producing motion. Rather I was using a very specific term that has an official connotation. The official connotation is that ***move***, ***turn***, and ***roll*** are the names of primitive methods that are used to produce very specific kinds of motion.

Color coded axes

When you select a view of an object in scene edit mode that causes the axes belonging to that object to be visible, the axis that protrudes *up* is colored green, the axis that protrudes to the *right* is colored red, and the axis that protrudes to the *front* is colored blue. For example, Figure 4 shows an airplane object with its three axes exposed in scene edit mode. As you will see later, however, up, right, and front from the viewpoint of the object are not always the same as up, right, and front from the viewpoint of the world. For example, if you ***roll*** the airplane in Figure 4 by one-half a revolution, the green axis will then be pointing *down* from the viewpoint of the world but will still be pointing *up* from the viewpoint of the object.

Figure 4. An airplane with its 3D axes exposed.



Many objects have component parts

As mentioned earlier, many objects have component parts. For example, a Penguin object is made up of five smaller component objects:

- head
- right leg
- left leg
- right wing
- left wing

Each of those component objects may contain other component objects. For example, the head contains the following component objects:

- upper beak
- lower beak

Every component object has its own center point

Because the right and left wings of a penguin object are themselves objects, it is possible to animate a penguin causing it to flap its wings independently of one another. We can do that by rotating each wing around one of its axes. However, in order for us to know how to do that, we must know where the center point is for each wing and we must know what constitutes front, right, and up from the viewpoint of the wing. In other words, we must know the directions of the red, green, and blue axes relative to the center point of the wing.

Turn and/or rotate

If we tell a penguin or any other object, *(such as a penguin's wing)* to ***turn*** to the right or to the left, that object will rotate *(yaw)* around the green axis that goes through the object's center point. If we tell an object to ***turn*** forward or backward, the object will rotate *(pitch)* around the red axis that goes through that object's center point. Finally, if we tell an object to ***roll*** to the right or to the left, the object will rotate around the blue axis that goes through that object's center point. Think about that while viewing the image of the airplane with its axes exposed in Figure 4.

As you can see, in order to cause a penguin to flap a wing, we must know the position of the wing object's center point, and we must also know the directions of the red, green, and blue axes that go through the center point for that wing.

Yaw, pitch, and roll

An object in 3D space can *yaw*, *pitch*, or *roll*, and can do any one of the three in either of two directions. Therefore, the object can experience any combination of the following three rotations:

1. Yaw left or yaw right *(it cannot yaw left and yaw right at the same time)*
2. Pitch down or pitch up *(it cannot pitch down and pitch up at the same time)*
3. Roll left or roll right *(it cannot roll left and roll right at the same time)*

Translation

In addition to rotation about the three axes, an object can also move or change its position in 3D space *(translate)*:

1. Forward or backward *(but not both at the same time)*
2. Right or left *(but not both at the same time)*
3. Up or down *(but not both at the same time)*

Six degrees of freedom

The combination of the three possible rotational motions and the three possible translational motions, *(which can occur concurrently)* results in what is often called *six degrees of freedom*. Thus, Alice objects can be animated with six degrees of freedom *(or more if you count the fact that the legs, arms, wings, etc., can experience independent rotation and/or translation while the object to which they belong is also experiencing rotation and/or translation).*

Six degrees of freedom
I told you earlier that many Alice objects are composed of other objects. I also told you that every Alice object has six degrees of freedom. Even the smaller component objects that make up other objects have six degrees of freedom. However, it may not make sense to exercise all six in all cases. For example, an airplane cannot fly backwards, but an object that represents a person could walk backwards if properly animated to do so. An interesting exercise would be to animate the mad scientist shown in Figure 1 to cause him to do Michael Jackson's moon walk.

An object's axes travel with the object

Every object has a center point and has its own set of 3D axes. The center point and the 3D axes belonging to an object travel with and rotate with the object, independently of the other objects in the world. Thus, the 3D axes belonging to the penguin that I mentioned earlier travel and rotate with him. If I caused the penguin to ***turn*** forward one-half revolution in order to dive head-first into the water, that will cause his green axis, which originally pointed up *(from my viewpoint)* to be pointing down *(from my viewpoint)* . As a result, at that point, I must move him *up* to force him to fall *down* into the water head first. *(However, there are more elegant ways to accomplish the same thing in Alice using the default parameter named* ***asSeenBy****.)*

Animating component objects belonging to an object

To give you a taste of what is possible in Alice, I am going to show you how an Alice programmer might go about animating a component object that belongs to a larger object. In particular I will show you how to animate the left arm of a Coach object.

Consider the Coach object shown in Figure 5.

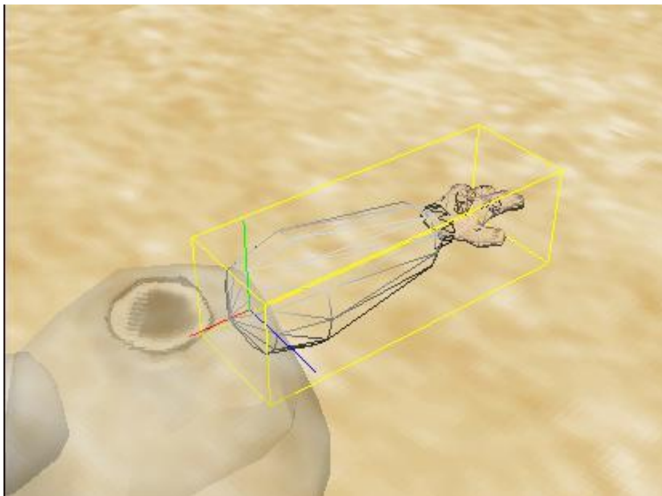
Figure 5. A Coach object.

The Coach object is actually made up of a large number of component objects, each of which has six degrees of freedom.

Consider the left arm at the shoulder

Let's consider just his left arm at the shoulder joint as shown in Figure 6.

Figure 6. Left arm of the Coach object.



The coach looks like a headless ghost

You may be wondering how I produced the image shown in Figure 6. To begin with, I repositioned the camera so that it would provide a better view of the center point on the left arm, which is what I wanted to see in detail.

Then I made his head invisible to get it out of the way. Then I set the **opacity** property of the upper body to 30-percent so that we can still see it for reference, but we can also see through it in order to see the shoulder joint.

Finally, I caused the left arm to be rendered as a *wireframe* drawing instead of a *solid* drawing. This made it possible for us to see the center point of the left arm along with the red, green, and blue 3D axes associated with that center point.

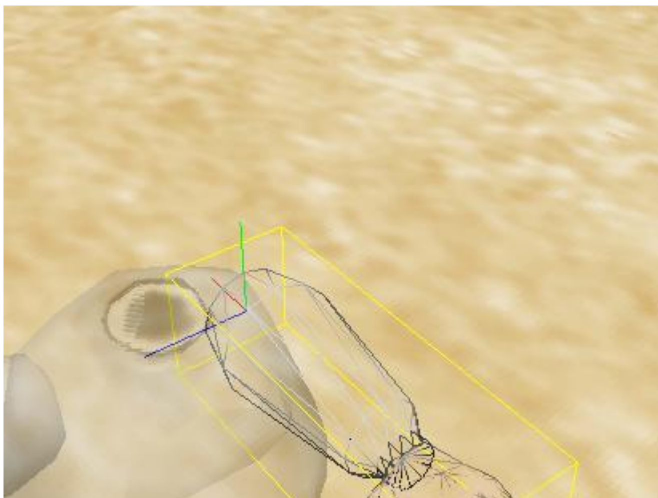
Note the directions of the axes

To begin our analysis, we recognize that from the coach's viewpoint *(and from our viewpoint as well)*, the green axis is pointed *up* at this point. Similarly, the blue axis is pointing toward the *front*, and the red axis is pointing into the coach's upper body toward his *right* side.

Rotate around the green axis

We can rotate the arm around the green axis. If we ***turn*** the arm one-fourth of a revolution *(90 degrees)* to the *right*, the coach will be pointing to the front. That would be OK, as shown in Figure 7. He would still be comfortable. *(Note that from the coach's viewpoint, the blue axis no longer points to the front and the red axis no longer points to the right but the red axis still points up.)*

Figure 7. Coach's arm *turned* 90 degrees to the right.



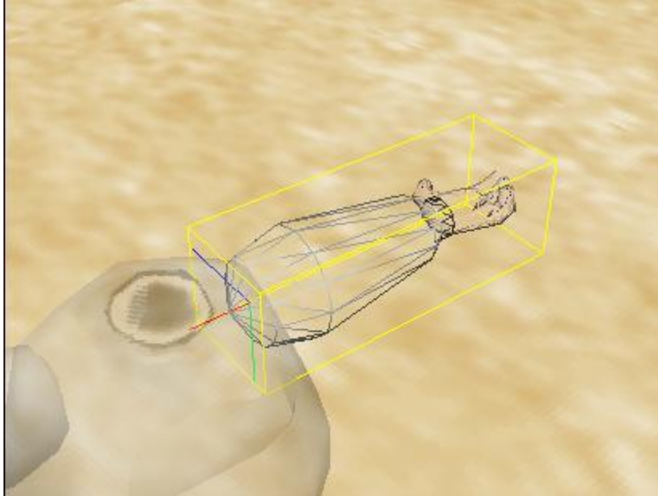
However, if we ***turn*** the arm to the *left* instead of the *right*, we can't turn it very far until we would put it in a position that is not possible for most humans. So, we need to be careful as to the limits if we ***turn*** the arm to the right or to the left.

Rotate around the red axis

Going back to the original pose in Figure 6, another possibility would be to ***turn*** the arm *backwards* so as to rotate it around the red axis by as much as one-half revolution *(180 degrees)*

as shown in Figure 8. However, turning the arm backwards by more than 180 degrees, or turning the arm forward by any amount at all would put the arm in an unnatural state.

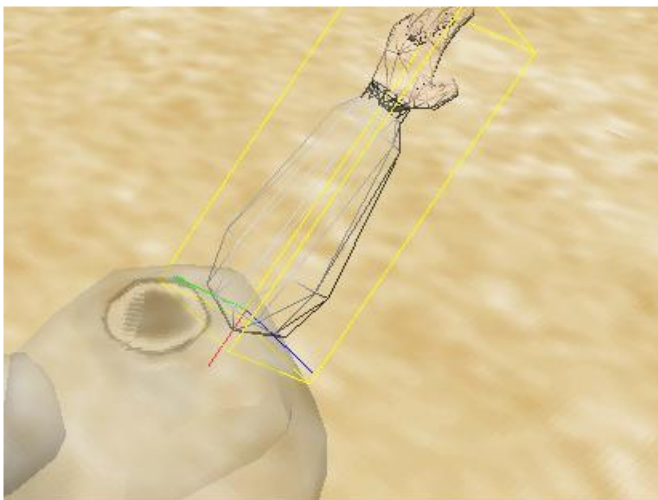
Figure 8. Coach's arm *turned* backwards by 180 degrees.



Rotate the arm around the blue axis

Once again, going back to the original pose in Figure 6, that leaves us with two more possibilities. We can ***roll*** the arm to the *left* or to the *right*, thus causing it to rotate around the blue axis. For example, Figure 9 shows the arm rolled to the right by one-eighth of a revolution *(45 degrees)*.

Figure 9. Coach's arm *rolled* right by 45 degrees.



As you can see, this caused the coach to lift his arm so as to point skyward.

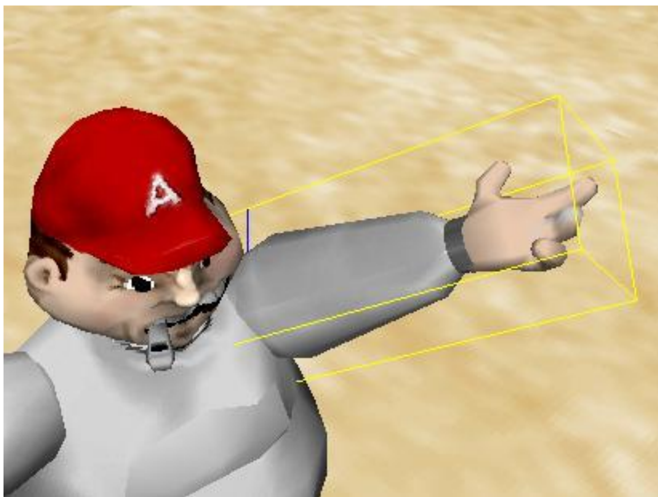
Difference between a right roll and a left roll

The difference between a right roll and a left roll can sometimes be a little confusing. To avoid the confusion, don't think primarily in terms of what happens to the arm proper. Rather, think of what happens to the red axis. For example, a right roll around the blue axis will cause the red axis to tilt downward, just like a right roll in an airplane will cause the right wing to tilt downward. In the case of the coach's left arm, if the red axis tilts downward, then the arm proper, which protrudes in the opposite direction from the red axis will tilt upward.

The grand finale

It is important to note that when one of these turn or roll operations is performed, the arm's 3D axes travel or rotate with the arm. For example, the green axis no longer points straight up in Figure 9. As a result, we could follow the motion in Figure 9 by a turn to the right by 45 degrees *(rotation around the new position of the green axis)* and follow that by a turn backwards by 180 degrees *(rotation around the new position of the red axis)* resulting in the image shown in Figure 10.

Figure 10. Coach's arm pointing to the left and up with palm up.



Body parts are once again visible

Note that I made all of the coach's body parts visible in Figure 10 so that we can see the new position of the left arm in the context of the entire body. In this case, the coach is pointing slightly upward and slightly to his left with his palm turned up.

Calling the turn and roll methods

Each of the motions described above can be executed interactively by manually calling the **turn** and **roll** methods on the arm object during manual setup. The same methods can also be called by program code at runtime to animate the arm.

To learn more about the handling of objects in 3D space by Alice, follow the link to the tutorial titled "Objects in 3D Space" in Resources.

A simple Alice program

It's time for us to take a look at how a simple Alice program might be put together. The goal of the program is to cause the ice skater in Figure 11 to face the parking meter, *(which is used here simply to provide a visible marker for the center of the world)*, rotate the lower portion of her left leg around her knee, and point her toe behind her as shown in Figure 12. Following this, she is to reverse the motion and resume the pose shown in Figure 11.

A case of the jaggies
Note that the requirement to reduce these images to a size that would fit in this narrow publication format caused them to have a slight case of the jaggies along diagonal lines.

Figure 11. Skater's starting and ending pose in program Alice0125f.

Figure 12. Animation shot from program Alice0125f.

Program listing

A complete listing of the source code for the program is presented in Listing 5 near the end of the lesson. The format of the listing shown in Listing 5 is one of three standard output formats that are available from Alice:

1. Alice Style
2. Java Style in Color
3. Java Style in Black & White

The format shown in Listing 5 is Java Style in Color. When a program listing is requested, it is delivered in an HTML file in the selected style.

Will discuss in fragments

I will discuss this program in fragments. The first fragment is shown in Listing 1.

Listing 1. Events for the program named Alice0125f.

Events	
When the world starts	
Do:	**world.main ( );**

Event-driven programming

As I have mentioned earlier, Alice is an event-driven programming language. An Alice program can service the set of event types shown in Table 5.

Table 5. Event types in Alice.

Event types in Alice

1. When the world starts
2. While the world is running
3. When a key is types
4. While a key is pressed
5. When the mouse is clicked on something
6. While the mouse is pressed on something
7. While something is true
8. When something becomes true
9. When a variable changes
10. Let the mouse move <objects>
11. Let the arrow keys move <subject>
12. Let the mouse move the camera
13. Let the mouse orient the camera

An event is fired *"When the world starts"* as well as at many other times during the execution of the program as shown in Table 5. This program handles only one of those event types. The code in Listing 1 instructs that the method named **main** is to be executed when that event is fired. Unlike Java, C++, and C# the Alice programmer can specify the method that is to be executed when the program first starts running by designating the method that is to handle the event shown in Listing 1.

The main method

When you create a new world in Alice, a skeleton for a method named **my_first_method** is automatically created. I don't like that name for a method so I routinely rename it **main**. The **main** method for this program is shown in Listing 2.

Listing 2. The main method for the program named Alice0125f.

Methods

```
public void main ( ) {
  // Program Alice0125f, Copyright 2007,
R.G.Baldwin
  world.setTheStage ( );
  world.playTheShow ( );
}
```

Methods in Alice

Methods in Alice can exist at either the world level or at the class *(object)* level. The closest analogy that I can draw to more conventional programming languages such as Java is that *world-level* methods are sort of like class methods in Java while *class-level* methods are definitely like instance methods in Java.

For this program, I defined two additional methods at the world level named:

1. setTheStage
2. playTheShow

As the names imply, the purpose of the first method is to get everything ready for the show and the second method actually presents the show. As you can see, the code in Listing 2 calls these two methods in sequence.

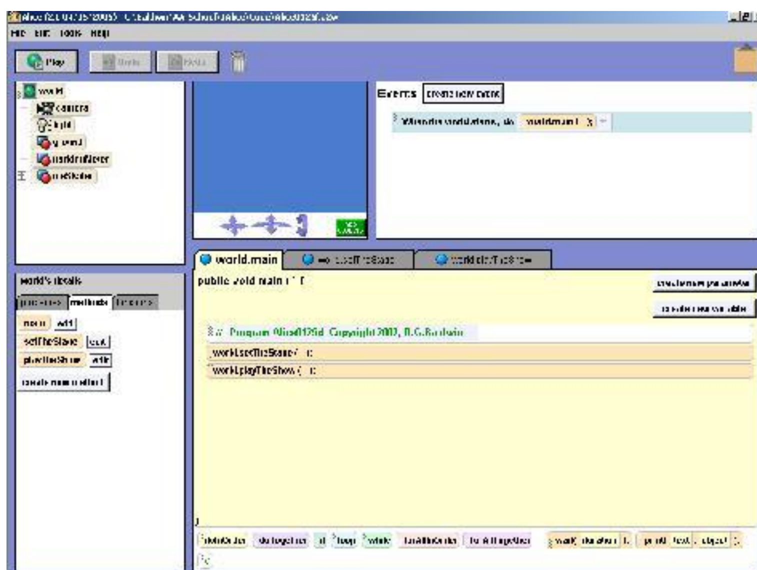
Writing programs in Alice

Before getting into the details of the two methods listed above, I need to explain a little about the physical process of writing programs in Alice. I mentioned earlier that the Alice development screen can be switched between two major edit modes:

1. Scene edit mode
2. Program edit mode

I showed you a picture of the scene edit mode in Figure 3. Figure 13 shows a greatly reduced screen shot of the program edit mode.

Figure 13. A reduced screen shot of the Alice program edit mode.



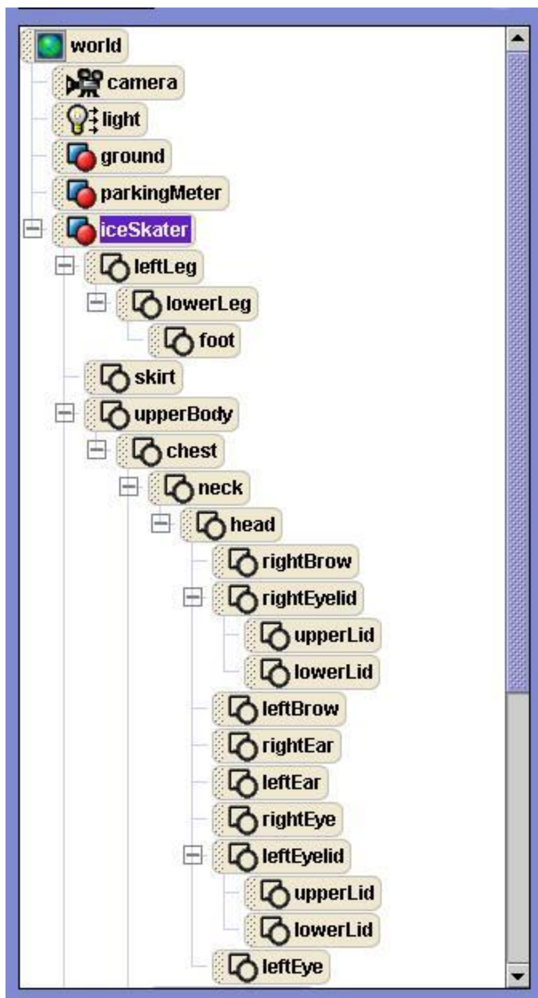
Various parts of the program edit screen

Beginning with the *object tree* in the white rectangle in the upper left, skipping the blue rectangular area *(which is simply a picture of the current state of the world)* and moving clockwise around the screen, Figures 14 through 17 show cropped *(and in some cases reduced)* screen shots of the major areas of the screen in program edit mode. Note that you can drag the blue boundaries between the different sections in Figure 13 to reallocate space on the screen. I did that here in an attempt to get the most informative screen shots.

The object tree

Figure 14 shows the object tree for the completed program named **Alice0125f** with the **iceSkater** object expanded to show as much detail regarding component objects as the screen size would allow.

Figure 14. Object tree with iceSkater object expanded.

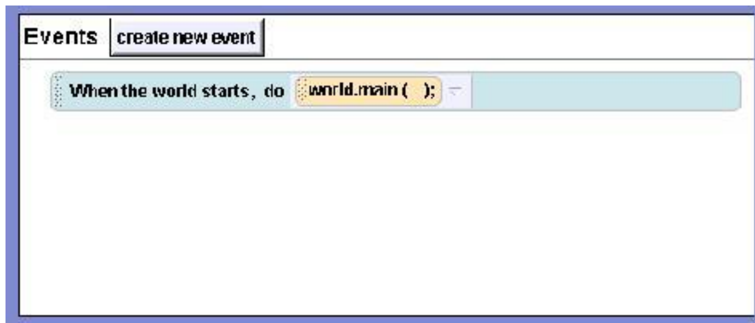


As you can see, the ice skater is a very detailed object with details extending all the way down to the upper and lower eye lids. As a result, this object can be animated to blink her eyes.

The Events area

This is the area of the screen that you work in when writing a program that services other events in addition to the one shown in Figure 15 and Listing 1. *(The item shown in Figure 15, with a different method name, is created automatically when you create the new world.)*

Figure 15. Reduced screen shot of Events area in program edit mode.



The edit pane

Figure 16 shows the part of the screen where you will spend most of your time after you design the program, add the objects to the world, do whatever manual setup you elect to do, and are ready to begin writing code to animate the world.

Figure 16. Reduced screen shot of edit pane in program edit mode.

The drag-and-drop paradigm

You actually write the source code by dragging tiles from the *variables area* at the top of Figure 16, from the *control structures are*a at the bottom of Figure 16, and from one of the tabbed panes shown in Figure 17. You drop those tiles into appropriate locations in Figure 16. You occasionally need to use the mouse with an image of a number-pad to enter a numeric value. You occasionally also need to type the text for a string, a new variable name, a new method name, a new parameter name, etc. into a dialog. Otherwise, no typing is required to write your source code.

Placeholders and other complicating factors

Sometimes you need to specify placeholder values when you drop a tile. You then come back and replace the placeholder with another tile later. Also, sometimes it can be a little tricky constructing a complex expression by dropping tiles in the correct order. All in all, however, once you get used to it, this is a very effective way to write source code for a program and it virtually eliminates the possibility of syntax errors, which is a good thing, especially for new aspiring programmers.

Defining new methods

You create the skeleton for a new method by:

- Selecting an object in the object tree in Figure 14 *(possibly including the world)*.
- Selecting the **methods** tab for that object in Figure 17.
- Clicking the button labeled **create new method** in Figure 17.

This will create a new tabbed pane as shown in Figure 16 that will contain the source code for that method.

Method parameters

If your new method requires parameters, you create them by clicking the button labeled **create new parameter** in Figure 16. This causes a dialog box to appear in which you specify the name and type of the new parameter. *(The allowable types in Alice are shown in Table 6.)* The parameter then appears inside the parentheses in the method signature in Figure 16. *(The **main** method in Figure 16 doesn't have any parameters.)*

Table 6. Allowable types in Alice.

Allowable types in Alice

• Number
• Boolean
• Object
• Other
    ○ String
    ○ Color
    ○ TextureMap
    ○ Sound
    ○ Pose
    ○ Position
    ○ Orientation
    ○ PointOfView
    ○ ReferenceFrame
    ○ Transformable
    ○ Light
    ○ Direction

The variables area

If your new method needs variables, you declare them by clicking the button labeled **create new variable** in Figure 16. This causes a dialog to appear that allows you to specify the name, type, and initial value of the variable. The variable then appears in a new tile directly below the method signature in the edit pane in Figure 16. However, the **main** method showing in Figure 16 doesn't have any variables.

The control structures area

The control structures area at the bottom of Figure 16 contains tiles that you use to specify the overall control structure of the method plus a few tiles that are used for other purposes. A brief description of each of these tiles is given in Table 7.

Table 7. Control structures in Alice.

Control structures in Alice
• **doInOrder** - creates a code block where each item is executed in sequential order. • **doTogether** - creates a code block where each item is executed concurrently. • **if** - creates an **if-else** construct. • **loop** - creates a **for** loop. • **while** - creates a **while** loop. • **forAllInOrder** - iterates an array or list applying the same action to each element in sequential order. • **forAllTogether** - processes an array or list applying the same action to all elements concurrently. • **wait** - inserts a pause in the execution of the program. • **print** - displays text in a special output area on the screen. • // - inserts a comment in the code.

As you can see from Table 7, Alice supports all of the control structures that are required for writing computer programs plus two structures that support concurrent programming. Alice does not support a **switch** structure or a **do-while** structure, but those are generally considered in computer science circles to be simply convenience structures.

The tabbed panes in the details area

Figures 17, 18, and 19 show screen shots of the three tabbed panes in the details area for the iceSkater object in the program named **Alice0125f**. Figure 17 shows the tabbed pane for the **methods** tab.

Figure 17. Screen shot of methods tab in details area for iceSkater object.

Figure 18 shows the tabbed pane for the **properties** tab.

Figure 18. Screen shot of properties tab in details area for iceSkater object.

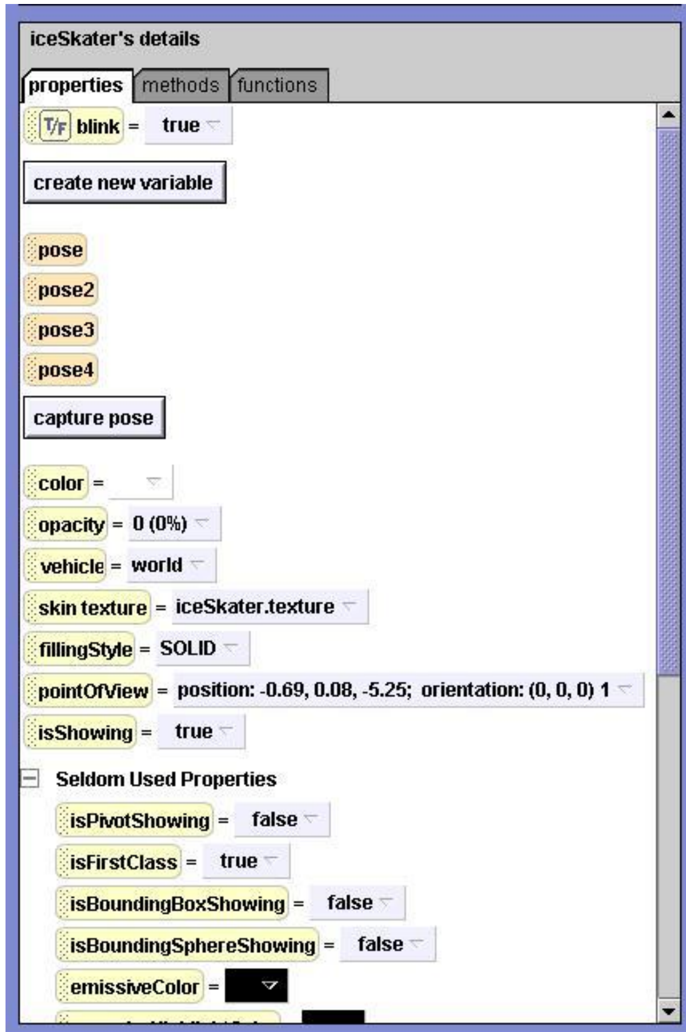
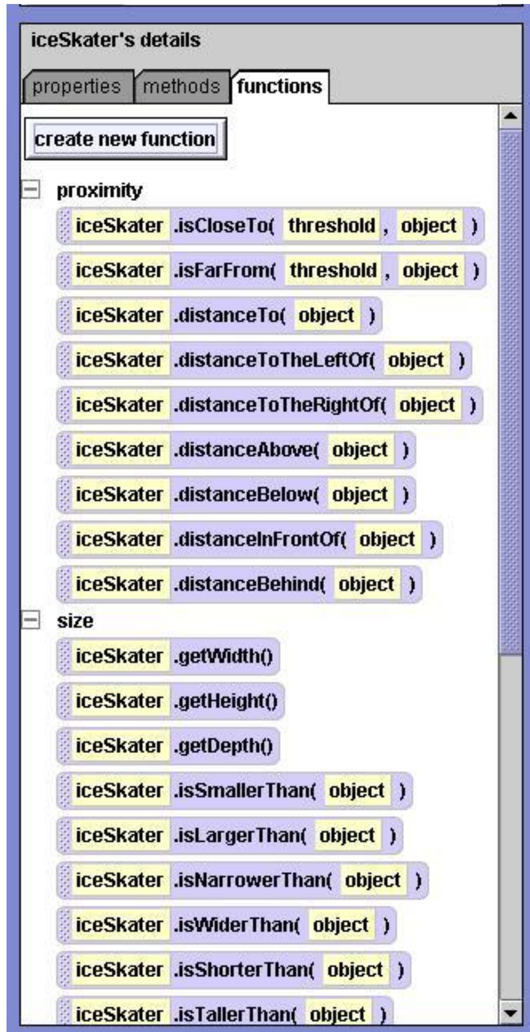


Figure 19 shows the tabbed pane for the **functions** tab.

Figure 19. Screen shot of functions tab in details area for iceSkater object.



The method named setTheStage

Listing 3 contains the source code for the method named **setTheStage**.

Listing 3. Source code for the method named setTheStage.

```
public void setTheStage ( ) {

  doInOrder {
    // Align objects to the world
    // Note: The ground, the parking meter, and
the skater
    // are invisible at startup
    ground .setPointOfView( world ); duration = 0
seconds
```

```
parkingMeter .setPointOfView( world ); duration = 0 seconds
iceSkater .setPointOfView( world ); duration = 0 seconds
// Set the camera to a known viewpoint
camera .setPointOfView( ground ); duration = 0 seconds
camera .turn( RIGHT , 135/360 revolutions ); duration = 0 seconds
camera .moveAwayFrom( target = parkingMeter , amount = 10 meters ); duration = 0 seconds
camera .move( UP , 1.5 meters ); duration = 0 seconds
// Set skater to her starting viewpoint
iceSkater .turn( RIGHT , 0.25 revolutions ); duration = 0 seconds
iceSkater .moveAwayFrom( target = parkingMeter , amount = 2 meters ); duration = 0 seconds
// Cause the ground, the parking meter, and
// the skater to become visible
ground .set( opacity , 1 (100%) ); duration = 0 seconds
parkingMeter .set( opacity , 1 (100%) ); duration = 1 second
iceSkater .set( opacity , 1 (100%) ); duration = 2 seconds
  }
}
```

A feel for the lay of the land

While I don't plan to explain this code in detail in this article, I will make a few comments just to give you a feel for the lay of the land.

The only manual setup required for this program is to:

- Create the world selecting the ice template for the ground.
- Add the ice skater and the parking meter to the world.

- Set the **opacity** properties for the ground, the ice skater, and the world to 0% to make them invisible at startup.

All of the remaining setup requirements are handled by the code in Listing 3.

Align objects to the world

The first three statements following the alignment comment cause the ground, the parking meter, and the ice skater to be positioned so that their centers are aligned with the center of the world and they all to face in the same direction as the world. *(Yes, the world does face in a particular direction.)* In other words, the viewpoints of the three objects are set to match *(aligned to)* the viewpoint of the world.

A duration property value of 0

Because the values of the default **duration** parameters for these three method calls are set to 0, this alignment process happens almost instantaneously.

The **duration** parameters for all of the method calls in Listing 3 *(with the exception of the final two statements)* are set to 0 so as to cause the entire setup to occur almost instantaneously.

Set the camera to a known viewpoint

The statements following the camera comment cause the viewpoint *(position and orientation)* of the camera to be set to a known viewpoint.

Set skater to her starting viewpoint

The code following this comment causes the skater to assume the viewpoint shown in Figure 11 *(but she is still invisible at this point).*

Make the objects visible

The last three statements in Listing 3 cause the ground, the parking meter, and the skater to become visible in that order. The ground becomes visible almost instantaneously. The parking meter requires one second to transition from invisible to totally visible. The skater requires two seconds to make the transition from being invisible to being totally visible. This is a simple transition effect, of which many are possible.

The doInOrder construct

You may be wondering why I included all of the code in Listing 3 in a **doInOrder** construct since all of the statements in Listing 3 would be executed in sequential order with or without that construct. The reason has to do with real estate in the edit pane. The available real estate in the edit pane in Figure 16 is somewhat limited. When the pane becomes full, you can scroll the code up and down in the typical fashion but this is somewhat inconvenient.

The code in the edit pane appears in the form of a tree structure that allows you to collapse and expand the nodes of the tree. The first seven control structures in Table 7 constitute nodes on that tree. When you place code in one of those structures, you can collapse all of the code contained in the structure into a single line of text in the edit pane to make room for other code. Therefore, I usually include sequential code in a **doInOrder** control structure so I can collapse it later to make room for creating and editing other code while minimizing the requirement to scroll the code in the edit pane.

The method named playTheShow

The source code for the method named **playTheShow** is shown in Listing 4.

Listing 4. Source code for method named playTheShow.

```
public void playTheShow ( ) {

  doInOrder {
    // Animate the skater's leg and foot
    doTogether {
      iceSkater.leftLeg.lowerLeg .turn(
      FORWARD , 0.25 revolutions );
      iceSkater.leftLeg.lowerLeg.foot .turn(
      FORWARD , 0.12 revolutions );
    }
    wait( 1 second );
    doTogether {
      iceSkater.leftLeg.lowerLeg.foot .turn(
      BACKWARD , 0.12 revolutions );
      iceSkater.leftLeg.lowerLeg .turn(
      BACKWARD , 0.25 revolutions );
    }
  }
}
```

A relatively short method

As you can see by comparing Listing 4 with Listing 3, the amount of code required to perform this animation is less than the amount of code required to get everything set up to begin with for this simple animation program.

Concurrent execution of program code

This method contains no parameters and no variables, so it is pretty simple as methods go. The code in broken into two blocks each enclosed in a **doTogether** control structure. The two statements in each **doTogether** block are executed concurrently. The two **doTogether** blocks are executed in sequential order with a one-second pause in between.

The code in the first **doTogether** block causes the skater to rotate her lower leg around her knee and rotate her foot around her ankle during a default **duration** of one second. This causes the lower leg and foot objects to transition from the viewpoints that they have in Figure 11 to the new viewpoints that they have in Figure 12.

The code in the second **doTogether** block in Listing 4 is essentially the reverse of the code in the first **doTogether** block, causing the skater to resume the pose shown in Figure 11.

An appetizer, not a full-course meal

Since the purpose of this article is to provide an appetizer and not a full-course meal, this will probably be a good place for me to stop. The full course meal is available at http://www.dickbaldwin.com/tocalice.htm and I invite you to go there and partake of that meal.

The Alice website

I also recommend that you visit and explore the Alice website. In particular, I recommend that you view the two demonstration videos that are available there.

Beyond that, I recommend that you download the Alice software, get it running, and run the tutorial and some of the sample animations that appear on the Welcome page when you start Alice running.

The Alice demonstration videos
Note that I was unable to view the second demonstration video in Firefox, but was successful in downloading and viewing it in Internet Explorer 7 by selecting the link on the page that reads *"If you have difficulty watching these videos, please try to view **this file in Windows Media Player**."* However, it took about six minutes to download the video because it is about 75 mbytes in size.

Visit my website to learn even more

And if you are interested in learning more about how to program using Alice, be sure to visit my website.

Resources

General resources

- Dick Baldwin's website
- Alice v2.0, Learn to Program Interactive 3D Graphics (Alice website)
- When Things Go Wrong (with the Alice program)

Resources from Baldwin's lessons in the series titled "Learn to Program using Alice"

- Table of Contents page see this page for links to lessons that were incomplete at the time this article was written, but were completed later.
- 100 Getting Started
- 105 Setting the Stage
- 110 Objects in 3D Space
- 115 Setting the Stage Manually, Part 1
- 120 Setting the Stage Manually, Part 2
- 125 Your First Alice Program
- 130 The Program Development Cycle
- 135 Functions that return values
- 140 Data Types and Variables
- 145 World-Level Methods
- 150 Class-Level Methods and Inheritance
- 155 Syntax, Runtime, and Logic Errors
- 160 Expressions and Operators
- More lessons will be added between 160 and 900.
- 900 Appendix A, Behavior of Primitive Methods

Downloads

- Download Alice v2.0

Complete program listing

A complete listing of the program discussed in this lesson is shown in Listing 5.

Listing 5. Source code for the program named Alice0125f.

Alice0125f's Code

Created by: Dick Baldwin

world

Events

When the world starts	
Do:	**world.main ( );**

Methods

```
public void main ( ) {

  // Program Alice0125f, Copyright 2007, R.G.Baldwin
  world.setTheStage ( );
  world.playTheShow ( );
}

public void setTheStage ( ) {

  doInOrder {
    // Align objects to the world
    // Note: The ground, the parking meter, and the skater
    // are invisible at startup
    ground .setPointOfView( world ); duration = 0 seconds
    parkingMeter .setPointOfView( world ); duration = 0 seconds
    iceSkater .setPointOfView( world ); duration = 0 seconds
    // Set the camera to a known viewpoint
    camera .setPointOfView( ground ); duration = 0 seconds
    camera .turn( RIGHT , 135/360 revolutions ); duration = 0 seconds
    camera .moveAwayFrom( target = parkingMeter , amount = 10 meters ); duration = 0 seconds
    camera .move( UP , 1.5 meters ); duration = 0 seconds
    // Set skater to her starting viewpoint
    iceSkater .turn( RIGHT , 0.25 revolutions ); duration = 0 seconds
    iceSkater .moveAwayFrom( target = parkingMeter , amount = 2 meters ); duration = 0 seconds
    // Cause the ground, the parking meter, and
    // the skater to become visible
```

```
      ground .set( opacity , 1 (100%) ); duration = 0 seconds
      parkingMeter .set( opacity , 1 (100%) ); duration = 1 second
      iceSkater .set( opacity , 1 (100%) ); duration = 2 seconds
    }
}

public void playTheShow ( ) {

    doInOrder {
      // Animate the skater's leg and foot
      doTogether {
        iceSkater.leftLeg.lowerLeg .turn( FORWARD , 0.25 revolutions );
        iceSkater.leftLeg.lowerLeg.foot .turn( FORWARD , 0.12 revolutions );
      }
      wait( 1 second );
      doTogether {
        iceSkater.leftLeg.lowerLeg.foot .turn( BACKWARD , 0.12 revolutions );
        iceSkater.leftLeg.lowerLeg .turn( BACKWARD , 0.25 revolutions );
      }
    }
}
```

Copyright

Copyright 2007, Richard G. Baldwin. Reproduction in whole or in part in any form or medium without express written permission from Richard Baldwin is prohibited.

About the author

Richard Baldwin *is a college professor (at Austin Community College in Austin, TX) and private consultant whose primary focus is a combination of Java, C#, and XML. In addition to the many platform and/or language independent benefits of Java and C# applications, he believes that a combination of Java, C#, and XML will become the primary driving force in the delivery of structured information on the Web.*

Richard has participated in numerous consulting projects and he frequently provides onsite training at the high-tech companies located in and around Austin, Texas. He is the author of Baldwin's Programming Tutorials, which have gained a worldwide following among experienced and aspiring programmers. He has also published articles in JavaPro magazine.

In addition to his programming expertise, Richard has many years of practical experience in Digital Signal Processing (DSP). His first job after he earned his Bachelor's degree was doing DSP in the Seismic Research Department of Texas Instruments. (TI is still a world leader in DSP.) In the following years, he applied his programming and DSP expertise to other interesting areas including sonar and underwater acoustics.

Richard holds an MSEE degree from Southern Methodist University and has many years of experience in the application of computer technology to real-world problems.

Baldwin@DickBaldwin.com

Keywords
java Alice

-end-